
Phalcon PHP Framework Documentation

Release 3.1.1

Phalcon Team

Nov 13, 2017

1	Qu'est-ce que Phalcon ?	3
2	table des matières	5
2.1	Installation	5
2.1.1	Installation	5
2.1.2	Outils pour développeurs Phalcon	25
2.2	Tutorials	45
2.2.1	Tutoriel 1: Apprenons par l'exemple	45
2.2.2	Tutoriel 2: Présentation d'INVO	55
2.2.3	Tutorial 3: Securing INVO	61
2.2.4	Tutorial 4: Travailler avec le CRUD	71
2.2.5	Tutorial 5: Customizing INVO	86
2.2.6	Tutorial 6: Vökuró	89
2.2.7	Tutorial 7: Créer une application REST API	92
2.2.8	Liste d'exemple de projets	103
2.3	Components	103
2.3.1	Injection de dépendance/Localisation de Service	103
2.3.2	L'architecture MVC	117
2.3.3	Utilisation de Contrôleurs	118
2.3.4	Travailler avec les Modèles	125
2.3.5	Relation de modèle	144
2.3.6	Evénements et Modèles	159
2.3.7	Model Behaviors	165
2.3.8	Models Metadata	170
2.3.9	Model Transactions	174
2.3.10	Validation de modèles	178
2.3.11	Travailler avec des modèles (Avancé)	181
2.3.12	Phalcon Query Language (PHQL)	192
2.3.13	Caching in the ORM	209
2.3.14	ODM (Object-Document Mapper)	222
2.3.15	Using Views	235
2.3.16	View Helpers (Tags)	254
2.3.17	Assets Management	266
2.3.18	Volt: Template Engine	273
2.3.19	MVC Applications	297
2.3.20	Routage	304

2.3.21	Dispatching Controllers	325
2.3.22	Micro Applications	336
2.3.23	Travailler avec les espaces de nom	351
2.3.24	Events Manager	354
2.3.25	Request Environment	360
2.3.26	Returning Responses	363
2.3.27	Cookies Management	367
2.3.28	Generating URLs and Paths	368
2.3.29	Flashing Messages	371
2.3.30	Stocker des données dans une session	375
2.3.31	Filtering and Sanitizing	378
2.3.32	Contextual Escaping	382
2.3.33	Validation	386
2.3.34	Forms	395
2.3.35	Configuration de lecture	406
2.3.36	Pagination	409
2.3.37	Improving Performance with Cache	412
2.3.38	Security	421
2.3.39	Encryption/Decryption	424
2.3.40	Access Control Lists - Listes de Contrôle d'Access (ACL)	427
2.3.41	Multi-lingual Support	436
2.3.42	Class Autoloader	439
2.3.43	Logging	443
2.3.44	Annotations Parser	449
2.3.45	Applications en Ligne de Commande	456
2.3.46	Images	460
2.3.47	File d'attente	465
2.3.48	Database Abstraction Layer	467
2.3.49	Internationalization	482
2.3.50	Database Migrations	484
2.3.51	Debugging Applications	490
2.3.52	Unit testing	496
2.4	In Depth Explanations / Further Reading	500
2.4.1	Améliorer les performances : C'est quoi la suite ?	500
2.4.2	La dépendance d'injection expliquée	508
2.4.3	Understanding How Phalcon Applications Work	514
2.5	API	517
2.5.1	API Indice	517
2.6	Legal	1113
2.6.1	License	1113
3	Previous Versions	1115
4	Autres formats	1117

Bienvenue sur le framework Phalcon, une toute nouvelle approche des frameworks PHP. Notre objectif est de vous donner un outil de pointe pour développer des sites Web et des applications sans se soucier des performances.

CHAPTER 1

Qu'est-ce que Phalcon ?

Phalcon est un framework PHP 5 complet et open source. Il est écrit en C et disponible en tant qu'extension php ce qui en fait un framework optimisé et très performant. Il n'est pas nécessaire d'apprendre ou d'utiliser le langage C puisque les fonctionnalités sont disponibles en tant que classes pour PHP prêtes à l'utilisation. Phalcon est également extrêmement modulable, vous permettant ainsi d'utiliser ses composants comme vous le souhaitez au sein de votre application, ou de n'importe quelle autre application.

En plus des performances, notre but est de faire de Phalcon un framework robuste, riche en fonctionnalités et facile d'utilisation !

2.1 Installation

2.1.1 Installation

Les extensions PHP nécessitent une méthode d'installation un peu différente des librairies ou des framework basés sur PHP. Vous pouvez soit télécharger le paquet binaire pour le système de votre choix soit le construire à partir des sources.

Windows

Pour utiliser Phalcon sur Windows vous pouvez [télécharger](#) une DLL. Ouvrez votre php.ini et ajoutez la ligne suivante:

```
extension=php_phalcon.dll
```

Ensuite redémarrez votre serveur web.

La vidéo qui suit (en anglais) montre pas à pas comment installer Phalcon sur Windows:

Guides relatifs

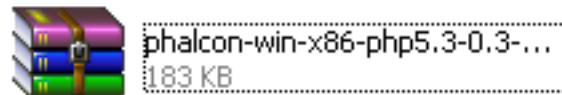
Installation sur XAMPP

XAMPP est une distribution Apache facile à installer qui contient MySQL, PHP et Perl. Une fois téléchargé XAMPP, vous l'extrayez simplement et commencez à vous en servir. Les instructions d'installation de Phalcon sur XAMPP sont détaillées ci-dessous. L'utilisation de la dernière version de XAMPP est fortement recommandée.

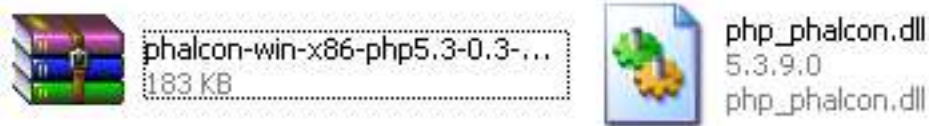
Téléchargement de la bonne version de Phalcon

XAMPP utilise toujours les versions 32 bits d'Apache et de PHP. Vous avez juste besoin de télécharger la version x86 de Phalcon pour Windows dans la section de téléchargement.

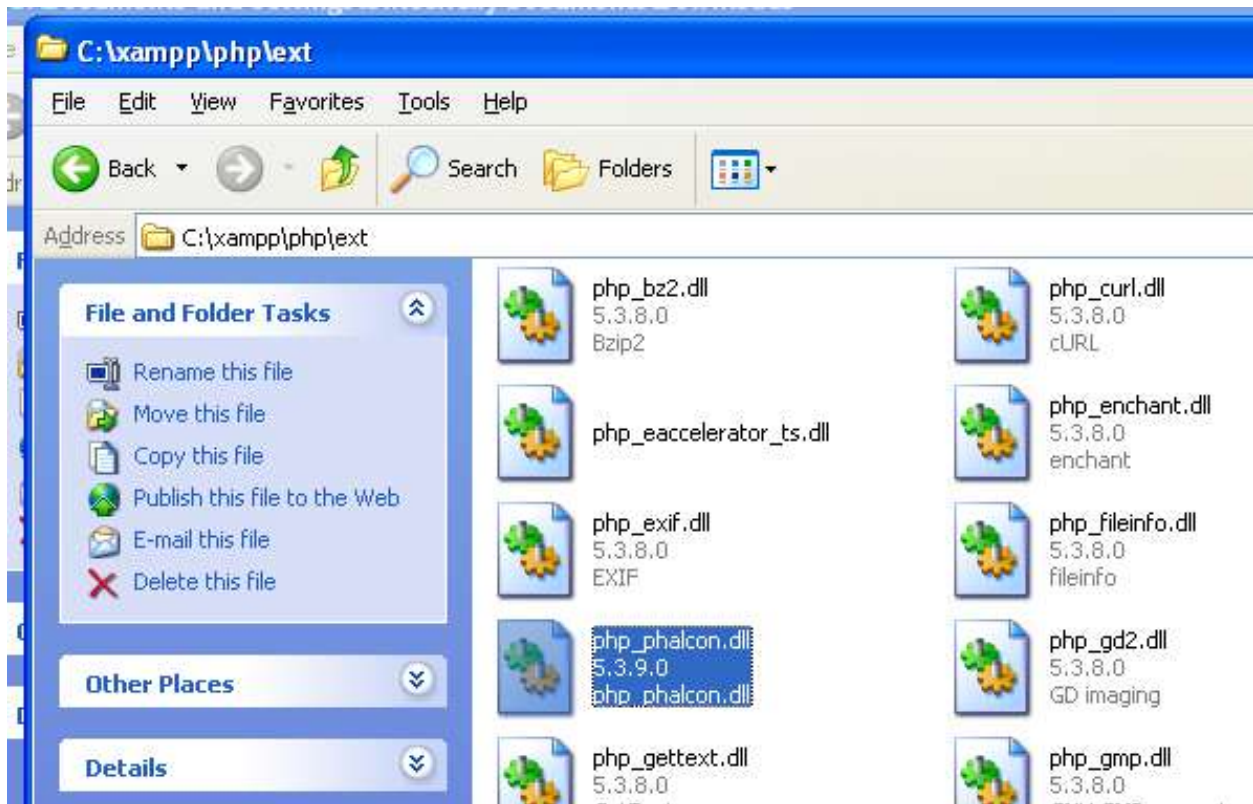
Après le téléchargement de la librairie Phalcon vous obtenez un fichier zip comme celui montré ci-dessous:



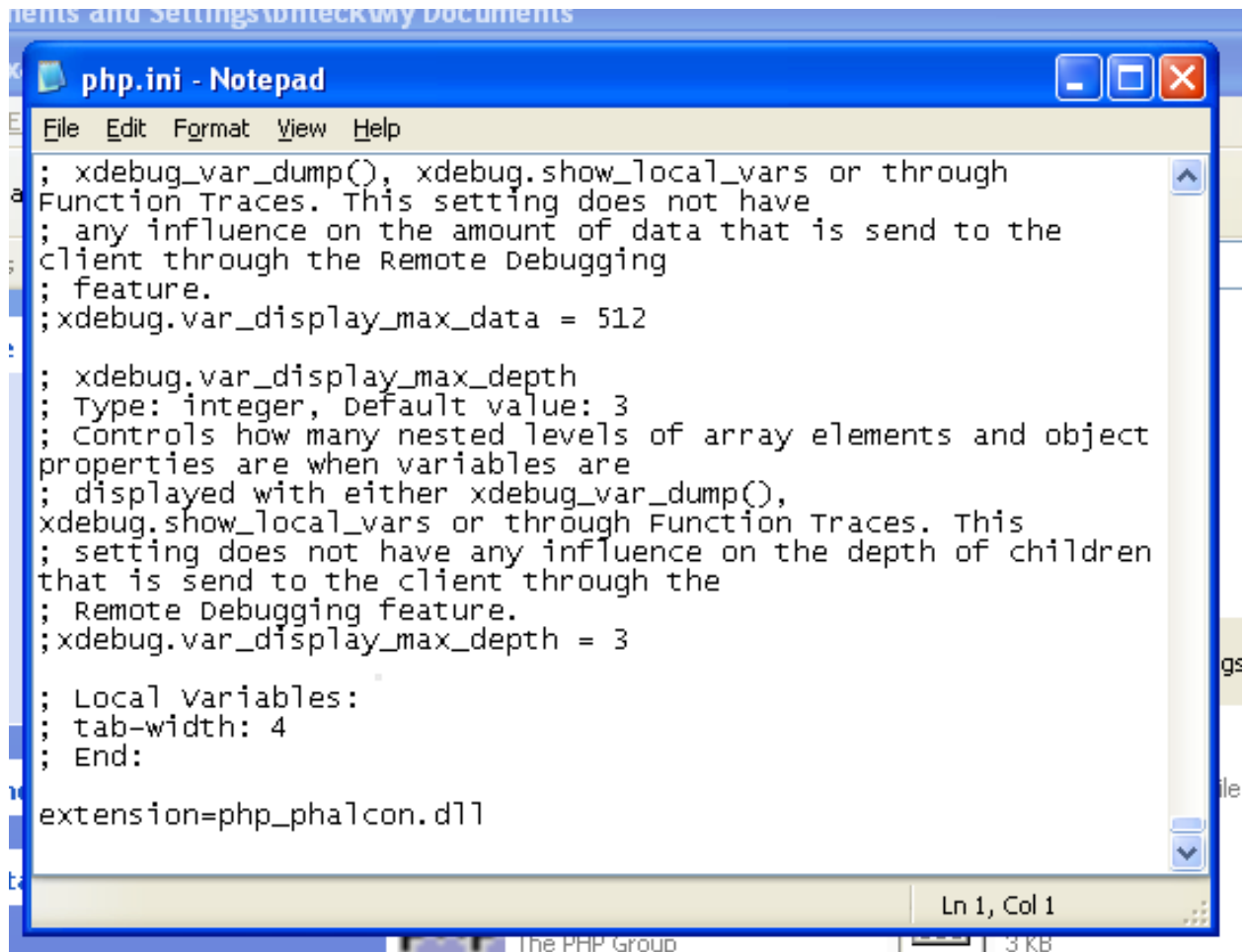
Extrayez la librairie de l'archive et récupérez la DLL Phalcon:



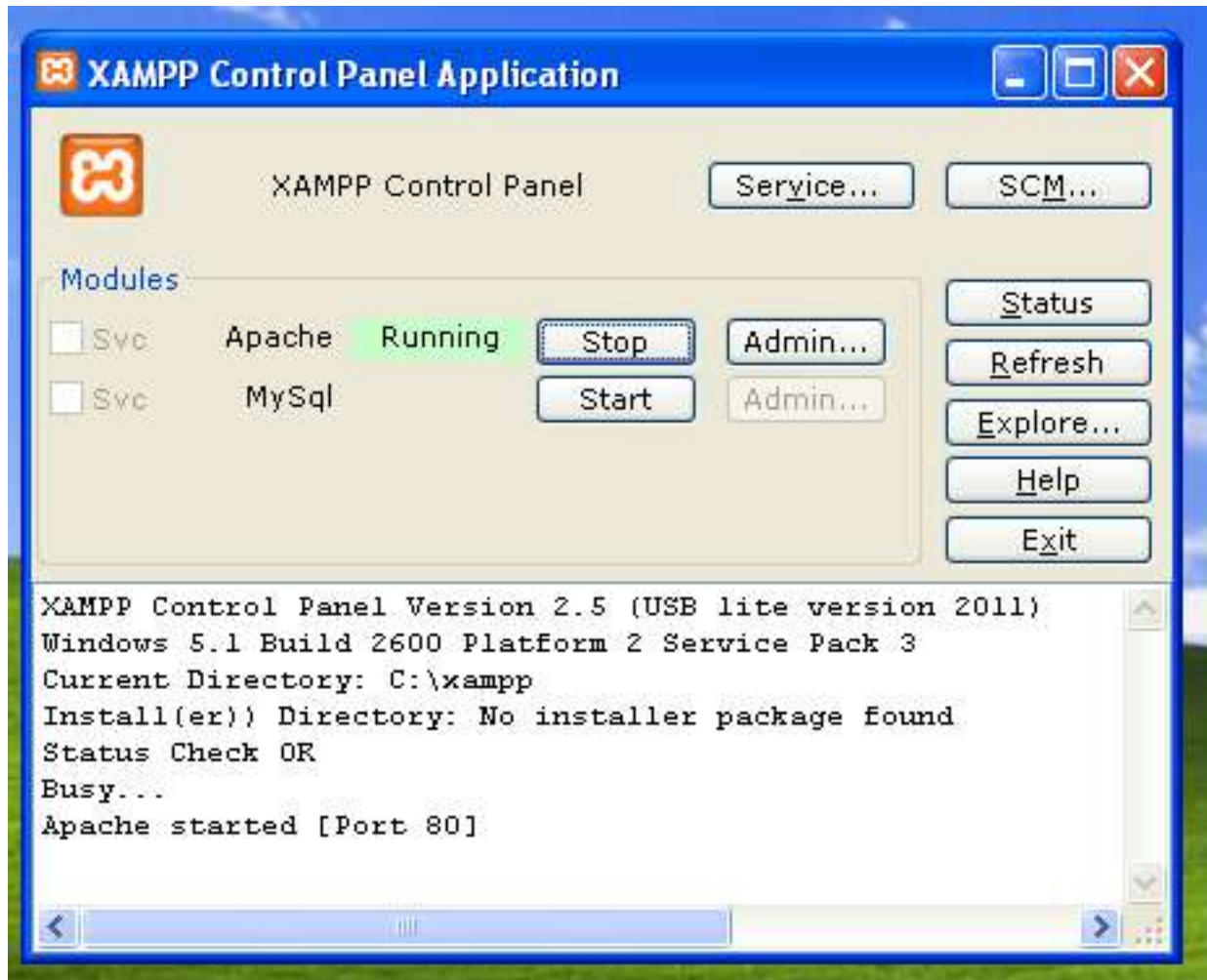
Copiez le fichier `php_phalcon.dll` à côté des autres extensions. Si vous avez installé XAMPP dans le dossier `C:\xampp`, l'extension doit se trouver dans `C:\xampp\php\ext`



Modifiez le fichier `php.ini` qui se trouve à `C:\xampp\php\php.ini`. Il peut être édité avec le bloc-notes (Notepad) ou tout autre programme similaire. Nous recommandons Notepad++ pour éviter les problèmes avec les fins de ligne. Ajoutez à la fin du fichier: `extension=php_phalcon.dll` et enregistrez-le.



Redémarrer le serveur Apache à partir du centre de contrôle XAMPP. La nouvelle configuration PHP devrait être chargée.



Ouvrez votre navigateur à l'adresse <http://localhost>. La page d'accueil de XAMPP devrait apparaître. Cliquez sur le lien `phpinfo()`.

`phpinfo()` génère une quantité importante d'informations à propos de l'état courant de PHP. Faites défiler vers le bas pour vérifier que l'extension Phalcon soit correctement chargée.

Si vous voyez la version de Phalcon au sein de la sortie de `phpinfo()`, alors félicitations ! Vous volez désormais avec Phalcon.²

Vidéo

La vidéo qui suit montre pas à pas l'installation de Phalcon sous Windows:

Voir aussi

- *Installation générale*
- *Installation détaillée avec WAMP pour Windows*

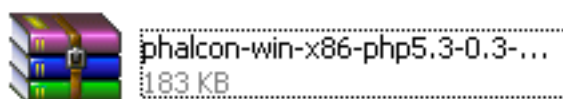
Installation sur WAMP

WampServer est un environnement de développement web pour Windows. Il vous permet de créer des applications web avec Apache2, PHP et des base MySQL. L'installation de Phalcon sur WampServer est détaillée ci-dessous. L'utilisation de la dernière version de WampServer est fortement recommandée.

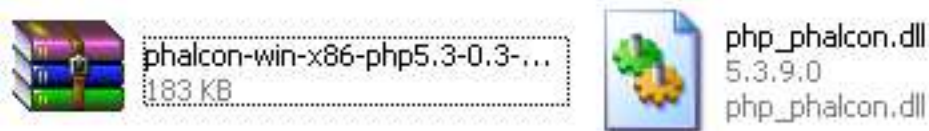
Télécharger la bonne version de Phalcon

WAMP existe en version 32 et 64 bits. Dans la section de téléchargement, choisissez Phalcon pour Windows en fonction de l'architecture utilisée.

Après le téléchargement de la librairie Phalcon, vous obtenez un fichier zip comme celui montré ci-dessous:



Extrayez la librairie depuis l'archive pour obtenir la DLL Phalcon:



Copiez le fichier `php_phalcon.dll` à côté des autres extensions PHP. Si WAMP est installé dans le dossier `C:\wamp`, l'extension devra être dans `C:\wamp\bin\php\php5.5.12\ext`

Modifiez le fichier `php.ini` qui se trouve à `C:\wamp\bin\php\php5.5.12\php.ini`. Il peut être édité avec le bloc-notes (Notepad) ou tout autre programme similaire. Nous recommandons Notepad++ pour éviter les problèmes avec les fins de ligne. Ajoutez à la fin du fichier: `extension=php_phalcon.dll` et enregistrez-le.

Modifiez aussi un autre fichier `php.ini` qui se trouve à `C:\wamp\bin\apache\apache2.4.9\bin\php.ini`. Ajoutez à la fin du fichier: `extension=php_phalcon.dll` et enregistrez-le.

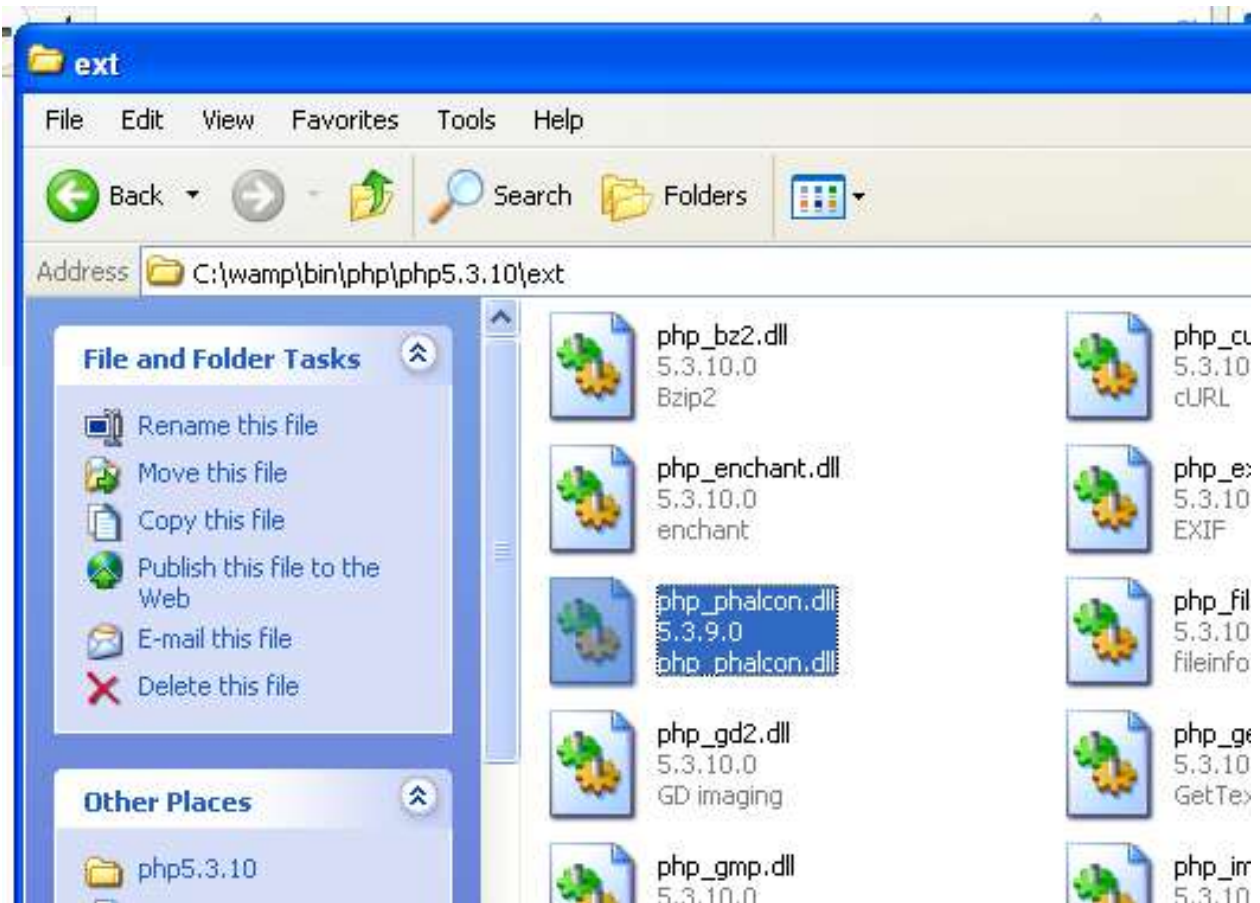
Redémarrer le serveur Apache. Faites un clic simple sur l'icône de WampServer qui se trouve dans la zone de notifications. Choisissez "Redémarrer les services" dans le menu. Vérifier que l'icône de notification redevienne verte.

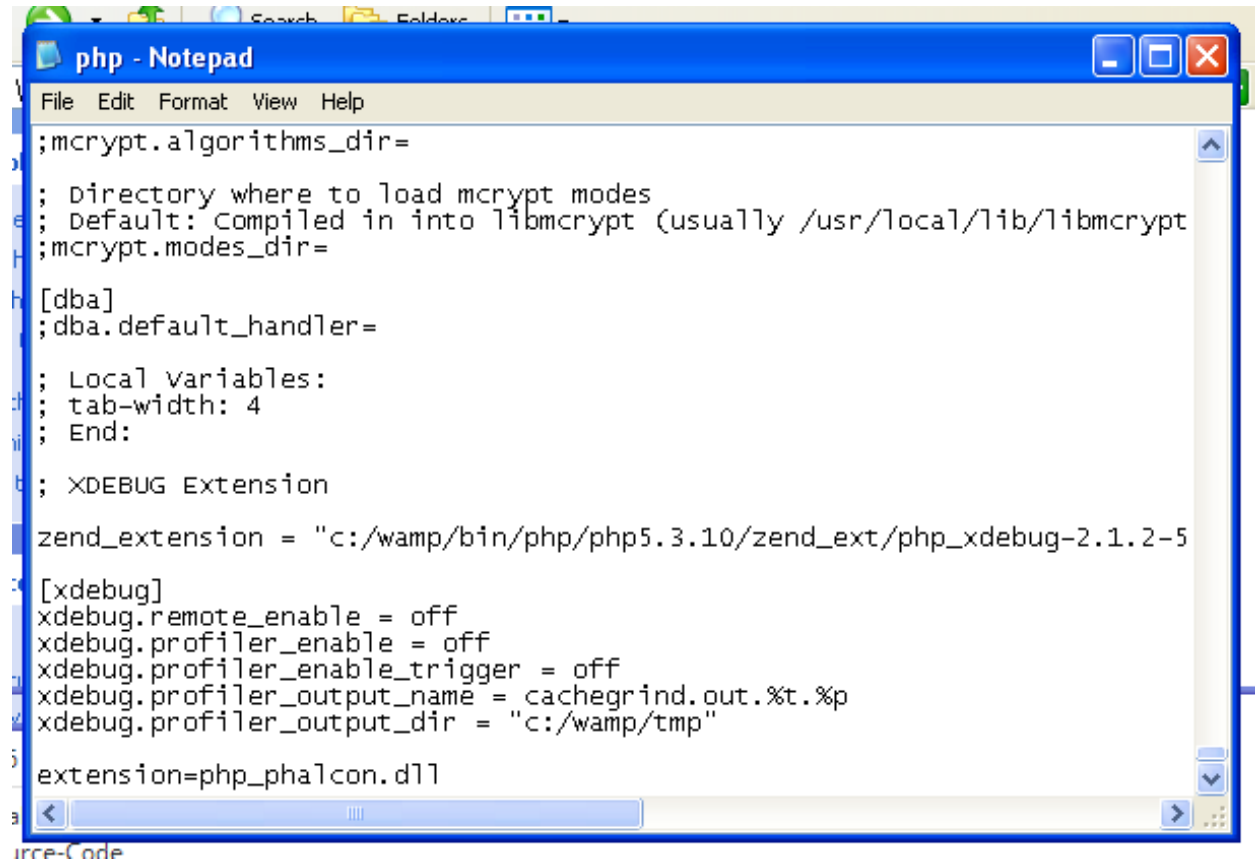
Ouvrez votre navigateur à l'adresse <http://localhost>. La page d'accueil de de WAMP doit apparaître. Consultez la section des extensions chargées pour contrôler que Phalcon soit bien chargée.

Félicitations! Vous volez désormais avec Phalcon.

Voir aussi

- *Installation générale*
- *Installation détaillée avec XAMPP pour Windows*

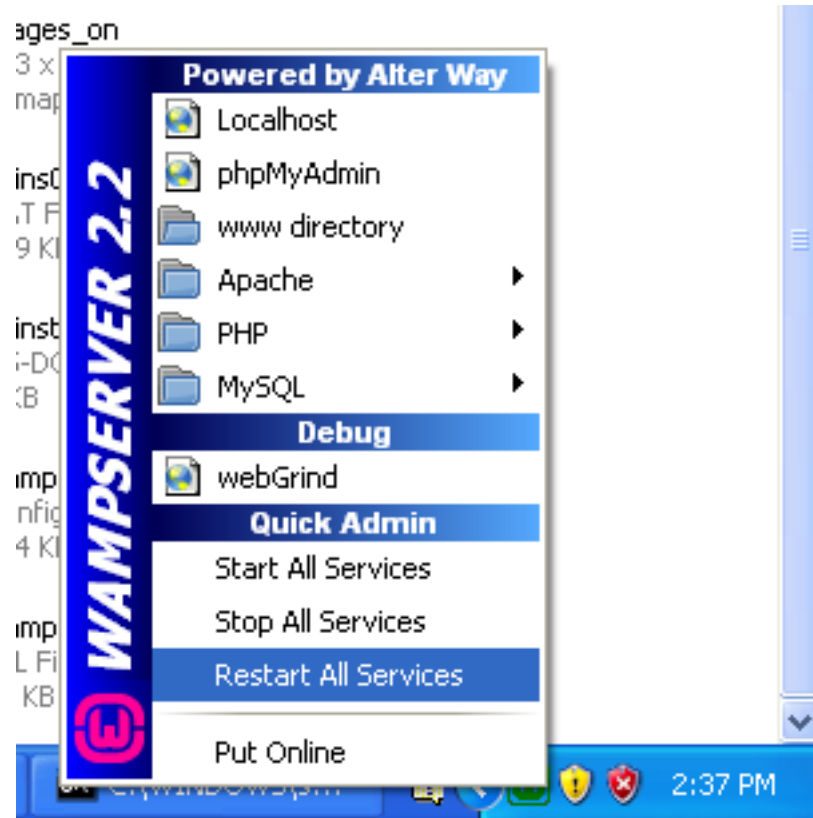




```

File Edit Format View Help
;mcrypt.algorithms_dir=
; Directory where to load mcrypt modes
; Default: Compiled in into libmcrypt (usually /usr/local/lib/libmcrypt
;mcrypt.modes_dir=
[dba]
;dba.default_handler=
; Local variables:
; tab-width: 4
; End:
; XDEBUG Extension
zend_extension = "c:/wamp/bin/php/php5.3.10/zend_ext/php_xdebug-2.1.2-5
[xdebug]
xdebug.remote_enable = off
xdebug.profiler_enable = off
xdebug.profiler_enable_trigger = off
xdebug.profiler_output_name = cachegrind.out.%t.%p
xdebug.profiler_output_dir = "c:/wamp/tmp"
extension=php_phalcon.dll

```



Version 2.2 [Version Française](#)

Server Configuration

Apache Version : 2.2.21**PHP Version :** 5.3.10

Loaded Extensions :

Core	bcmath	calendar	com_dotnet
ctype	date	ereg	filter
ftp	hash	iconv	json
mcrypt	SPL	odbc	pcre
Reflection	session	standard	mysqlnd
tokenizer	zip	zlib	libxml
dom	PDO	Phar	SimpleXML
wddx	xml	xmlreader	xmlwriter
apache2handler	mbstring	gd	mysql
mysqli	pdo_mysql	pdo_sqlite	phalcon
mhash	xdebug		

MySQL Version : 5.5.20

Linux/Solaris

Debian / Ubuntu

Pour ajouter le dépôt à votre distribution:

```
# Versions stables
curl -s https://packagecloud.io/install/repositories/phalcon/stable/script.deb.sh |
sudo bash

# Versions nocturnes
curl -s https://packagecloud.io/install/repositories/phalcon/nightly/script.deb.sh |
sudo bash
```

Ceci ne doit être fait qu'une seule fois, à moins d'un changement de distribution ou que vous souhaitiez basculer vers une construction nocturne.


Pour installer Phalcon:


```
sudo apt-get install php5-phalcon

# ou pour PHP 7
sudo apt-get install php7.0-phalcon
```

Distributions RPM (par ex. CentOS)

Pour ajouter le dépôt à votre distribution:

```
# Versions stables
curl -s https://packagecloud.io/install/repositories/phalcon/stable/script.rpm.sh | 
↪ sudo bash

# Versions nocturnes
curl -s https://packagecloud.io/install/repositories/phalcon/nightly/script.rpm.sh | 
↪ sudo bash
```

Ceci ne doit être fait qu’une seule fois, à moins d’un changement de distribution ou que vous souhaitiez basculer vers une construction nocturne.

Pour installer Phalcon:

```
sudo yum install php56u-phalcon

# or for PHP 7

sudo yum install php70u-phalcon
```

Compiler depuis les sources

Sur un système Linux/Solaris vous pouvez aisément compiler et installer l’extension en partant du code source:

Les paquets nécessaires sont:

- PHP >= 5.5 development resources
- GCC compiler (Linux/Solaris)
- Git (s’il n’est pas déjà installé sur votre système - sinon vous pouvez le télécharger depuis Github et le déposer sur votre serveur via FTP/SFTP)

Paquets spécifique pour les plateformes courantes:

```
# Ubuntu
sudo apt-get install php5-dev libpcre3-dev gcc make php5-mysql

# Suse
sudo yast -i gcc make autoconf php5-devel php5-pear php5-mysql

# CentOS/RedHat/Fedora
sudo yum install php-devel pcre-devel gcc make

# Solaris
pkg install gcc-45 php-56 apache-php56
```

Création de l’extension:

```
git clone git://github.com/phalcon/cphalcon.git

cd cphalcon/build

sudo ./install
```

Ajout de l'extension à votre configuration PHP:

```
# Suse: Ajoutez un fichier nommé phalcon.ini dans /etc/php5/conf.d/ avec ce contenu:
extension=phalcon.so

# CentOS/RedHat/Fedora: Ajoutez un fichier nommé phalcon.ini in /etc/php.d/ avec ce
↳ contenu:
extension=phalcon.so

# Ubuntu/Debian with apache2: Ajoutez un fichier nommé 30-phalcon.ini in /etc/php5/
↳ apache2/conf.d/ avec ce contenu:
extension=phalcon.so

# Ubuntu/Debian with php5-fpm: Ajoutez un fichier nommé 30-phalcon.ini in /etc/php5/
↳ fpm/conf.d/ avec ce contenu:
extension=phalcon.so

# Ubuntu/Debian with php5-cli: Ajoutez un fichier nommé 30-phalcon.ini in /etc/php5/
↳ cli/conf.d/ avec ce contenu:
extension=phalcon.so
```

Redémarrez le serveur web.

Si vous utilisez php5-fpm sur Ubuntu ou Debian, redémarrez le:

```
sudo service php5-fpm restart
```

Phalcon détecte automatiquement votre architecture, cependant vous pouvez forcer la compilation pour une architecture spécifique:

```
cd cphalcon/build

# One of the following:
sudo ./install 32bits
sudo ./install 64bits
sudo ./install safe
```

Si l'installateur automatique échoue, essayez la construction manuelle:

```
cd cphalcon/build/64bits

export CFLAGS="-O2 --fvisibility=hidden"

./configure --enable-phalcon

make && sudo make install
```

Mac OS X

Sur un système Mac OS X vous pouvez compiler et installer l'extension à partir du code source:

Pré-requis

Les paquets nécessaires sont:

- PHP >= 5.5 development resources

- XCode

```
# brew
brew tap homebrew/homebrew-php
brew install php55-phalcon
brew install php56-phalcon

# MacPorts
sudo port install php55-phalcon
sudo port install php56-phalcon
```

Ajoutez l’extension à votre configuration PHP.

FreeBSD

Un portage est disponible pour FreeBSD. Vous devez juste taper cette ligne pour l’installer:

```
pkg_add -r phalcon
```

ou

```
export CFLAGS="-O2 --fvisibility=hidden"

cd /usr/ports/www/phalcon

make install clean
```

Vérification de l’installation

Consultez la sortie de `phpinfo()` à la recherche d’une section nommée “Phalcon” ou bien exécutez ce bout de code ci-dessous:

```
<?php print_r(get_loaded_extensions()); ?>
```

L’extension Phalcon doit apparaître dans la sortie:

```
Array
(
    [0] => Core
    [1] => libxml
    [2] => filter
    [3] => SPL
    [4] => standard
    [5] => phalcon
    [6] => pdo_mysql
)
```

Notes d’installation

Instructions d’installation pour les serveurs Web:

Notes d'installation sur Apache

Apache est un serveur web populaire et bien connu, disponible sur de nombreuses plateformes.

Configurer Apache pour Phalcon

Ce qui suit sont de possibles configurations que vous pouvez utiliser pour configurer Apache pour Phalcon. Ces instructions sont principalement dirigées vers la configuration du module `mod_rewrite` qui permet l'utilisation d'URLs conviviales et du *router component*. Les applications ont habituellement cette structure:

```
test/
  app/
    controllers/
    models/
    views/
  public/
    css/
    img/
    js/
    index.php
```

Répertoire à partir de la racine document

C'est le cas le plus fréquent. L'application est installée dans n'importe quel dossier sous la racine document. Dans ce cas nous utilisons deux fichiers `.htaccess`. Le premier sert à masquer l'application en redirigeant toutes les requêtes vers la racine de l'application (`public/`).

```
# test/.htaccess

<IfModule mod_rewrite.c>
    RewriteEngine on
    RewriteRule ^$ public/ [L]
    RewriteRule ((?s).*) public/$1 [L]
</IfModule>
```

Le second fichier `.htaccess` est placé dans le dossier `public/`. Celui-ci réécrit toutes les URIs vers le fichier `public/index.php`:

```
# test/public/.htaccess

<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^((?s).*)$ index.php?url=/$1 [QSA,L]
</IfModule>
```

Si vous ne souhaitez pas utiliser les fichiers `.htaccess`, vous pouvez déplacer ces configurations dans le fichier principal de configuration d'Apache:

```
<IfModule mod_rewrite.c>

    <Directory "/var/www/test">
        RewriteEngine on
```

```
RewriteRule ^$ public/ [L]
RewriteRule ((?s).*) public/$1 [L]
</Directory>

<Directory "/var/www/test/public">
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^((?s).*)$ index.php?url=/$1 [QSA,L]
</Directory>

</IfModule>
```

Hôtes Virtuels

Cette deuxième configuration vous permet d'installer une application Phalcon dans un hôte virtuel:

```
<VirtualHost *:80>

    ServerAdmin admin@example.host
    DocumentRoot "/var/vhosts/test/public"
    DirectoryIndex index.php
    ServerName example.host
    ServerAlias www.example.host

    <Directory "/var/vhosts/test/public">
        Options All
        AllowOverride All
        Allow from all
    </Directory>

</VirtualHost>
```

Ou si vous utilisez Apache 2.4 et supérieurs:

```
<VirtualHost *:80>

    ServerAdmin admin@example.host
    DocumentRoot "/var/vhosts/test/public"
    DirectoryIndex index.php
    ServerName example.host
    ServerAlias www.example.host

    <Directory "/var/vhosts/test/public">
        Options All
        AllowOverride All
        Require all granted
    </Directory>

</VirtualHost>
```

Notes d'installation pour Nginx

Nginx est un projet libre et open-source qui permet d'avoir un serveur HTTP et un reverse proxy extrêmement performant ainsi qu'un relais IMAP/POP3. Contrairement aux serveurs classiques, Nginx n'exploite pas des threads pour traiter les requêtes. A la place il utilise une architecture plus évolutive basée sur des événements (asynchrones). Cette architecture utilise de petites quantités de mémoire mais plus importantes et prévisibles en cas de charge.

Le PHP-FPM (FastCGI Process Manager) est habituellement utilisé pour permettre à Nginx de traiter les fichiers PHP. Actuellement, PHP-FPM est fourni avec n'importe quelle distribution PHP Unix. La combinaison Phalcon + Nginx + PHP-FPM fournit un puissant ensemble d'outils qui offre un maximum de performance pour vos applications PHP.

Configurer Nginx pour Phalcon

Ce qui suit sont de possibles configurations que vous pouvez utiliser pour configurer Nginx avec Phalcon:

Configuration de base

En utilisant `$_GET['_url']` comme source des URIs:

```
server {
    listen      80;
    server_name localhost.dev;
    root        /var/www/phalcon/public;
    index       index.php index.html index.htm;
    charset     utf-8;

    location / {
        try_files $uri $uri/ /index.php?_url=$uri&$args;
    }

    location ~ /\.php {
        fastcgi_pass   unix:/run/php-fpm/php-fpm.sock;
        fastcgi_index  /index.php;

        include fastcgi_params;
        fastcgi_split_path_info      ^(\.+\.php)(/.+)$;
        fastcgi_param PATH_INFO      $fastcgi_path_info;
        fastcgi_param PATH_TRANSLATED $document_root$fastcgi_path_info;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }

    location ~ /\.ht {
        deny all;
    }
}
```

En utilisant `$_SERVER['REQUEST_URI']` comme source des URIs:

```
server {
    listen      80;
    server_name localhost.dev;
    root        /var/www/phalcon/public;
    index       index.php index.html index.htm;
    charset     utf-8;
```

```
location / {
    try_files $uri $uri/ /index.php;
}

location ~ /\.php$ {
    try_files $uri =404;

    fastcgi_pass 127.0.0.1:9000;
    fastcgi_index /index.php;

    include fastcgi_params;
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_param PATH_INFO $fastcgi_path_info;
    fastcgi_param PATH_TRANSLATED $document_root$fastcgi_path_info;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
}

location ~ /\.ht {
    deny all;
}
}
```

Notes d'installation pour Cherokee

[Cherokee](#) est un serveur web hautement performant. Il est très rapide, flexible et facile à configurer.

Configurer Cherokee pour Phalcon

Cherokee fournit une interface graphique conviviale pour configurer presque tous les réglages disponibles dans le serveur web. Démarrez l'administrateur cherokee avec `/path-to-cherokee/sbin/cherokee-admin` (NDT: On suppose que l'interface d'administration est en anglais)

Créez un nouveau serveur virtuel en cliquant sur “vServers” et renseignez alors votre nouveau serveur virtuel:

Le serveur virtuel nouvellement ajouté doit apparaître dans la barre gauche de l'écran. Dans l'onglet “Behaviors” vous verrez un ensemble de comportements par défaut pour ce serveur virtuel. Cliquez sur le bouton “Rule Management”. Supprimez ceux qui sont identifiés comme “Directory /cherokee_themes” et “Directory /icons”:

A l'aide de l'assistant, ajoutez le comportement “PHP Language”. Ce comportement vous permet d'exécuter des applications PHP:

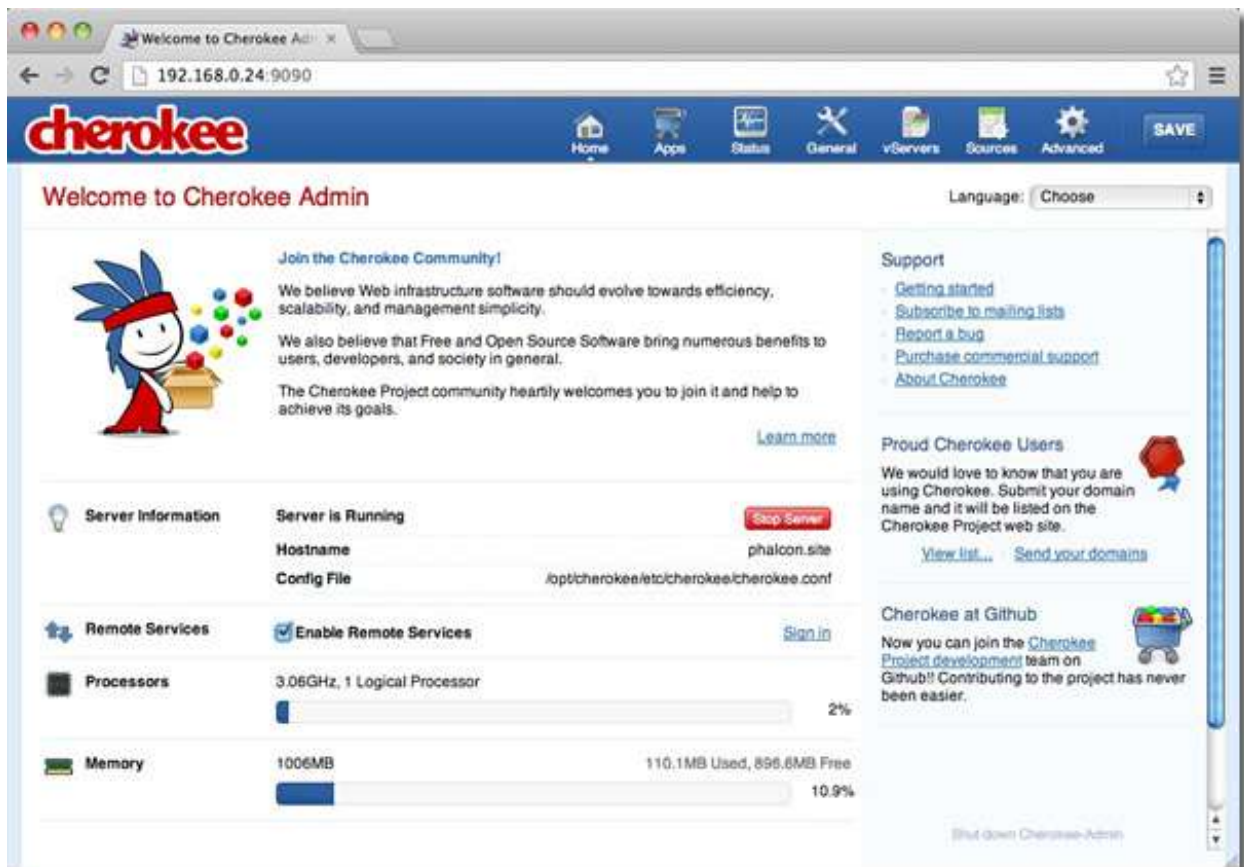
Normalement ce comportement ne nécessite aucun réglage supplémentaire. Ajoutez un autre comportement cette fois-ci en allant dans la section “Manual Configuration”. Dans “Rule Type” choisissez “File Exists”, ensuite assurez-vous que l'option “Match any file” soit activée:

Dans l'onglet “Handler”, choisissez “List & Send” comme gestionnaire:

Editez le comportement “Default” afin d'activer le moteur de réécriture d'URL. Définissez le gestionnaire à “Redirection”, et ajoutez l'expression régulière `^(.*)$` au moteur:

Enfin, assurez-vous que les comportements soient dans l'ordre suivant:

Lancez l'application dans un navigateur:



Add New Virtual Server

Manual

Manual configuration

CMS
Content Management Systems

Application Servers
General Purpose Application Servers

Platforms
Web Development Platforms

Languages
Development Languages and Platforms

Web Applications
General Purpose Applications

Tasks
Common Maintenance Tasks

Nick

Name of the Virtual Server you are about to create. A domain name is alright.

Document Root

Document Root directory of the new Virtual Server.

Add **Cancel**

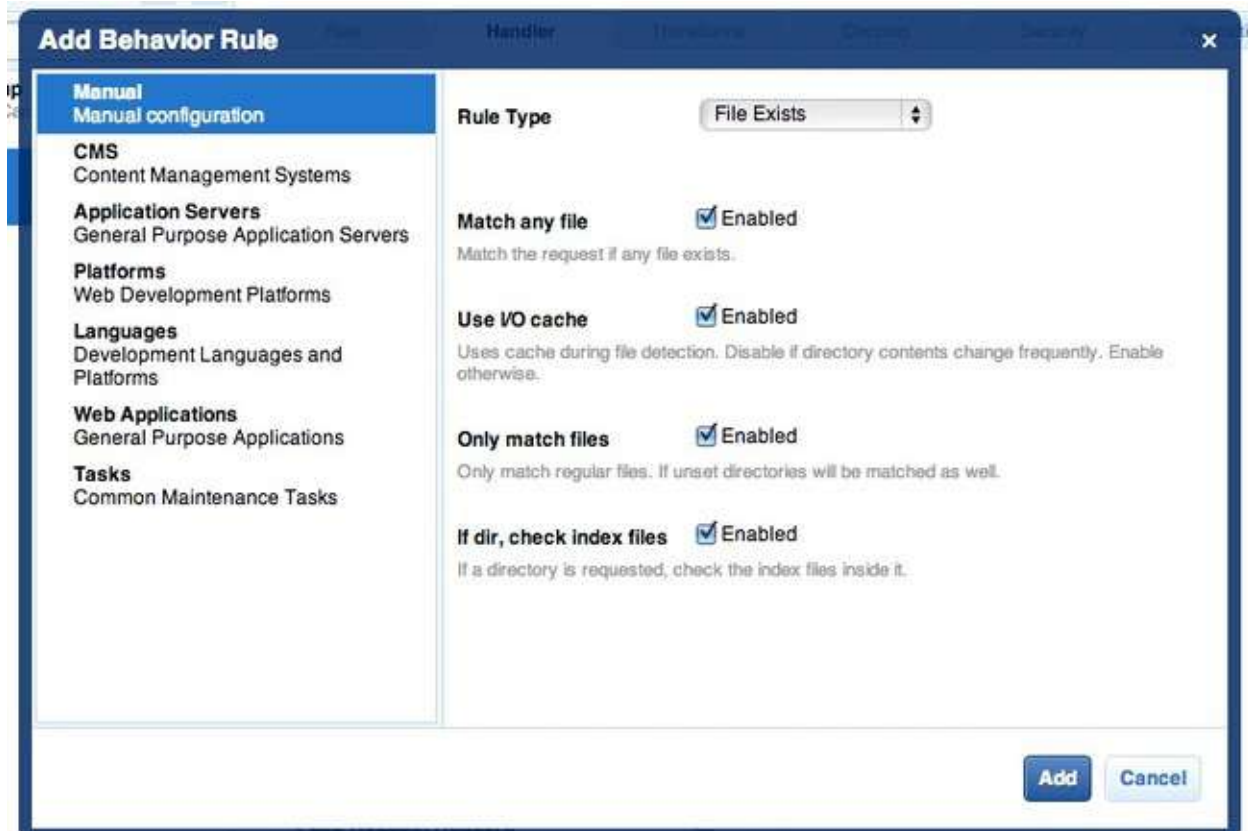
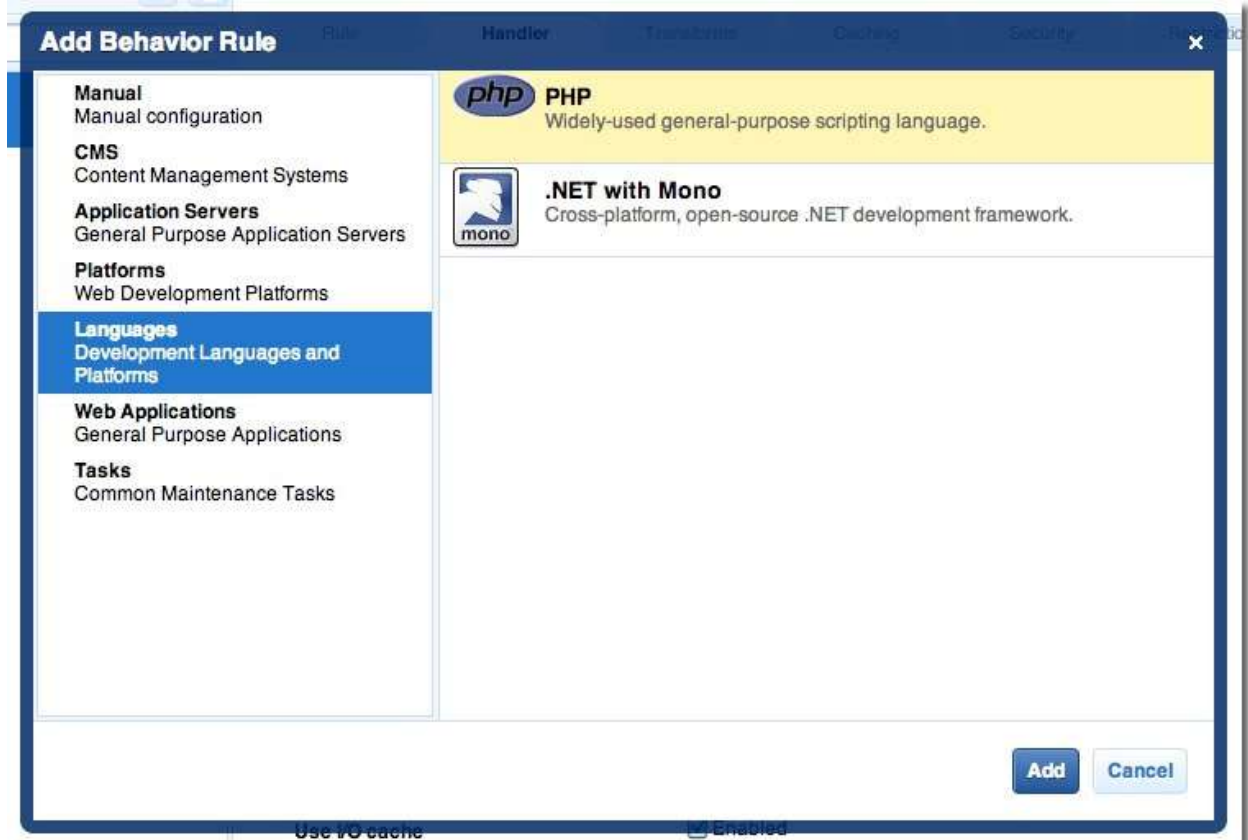
Virtual Server: my-phalcon.app

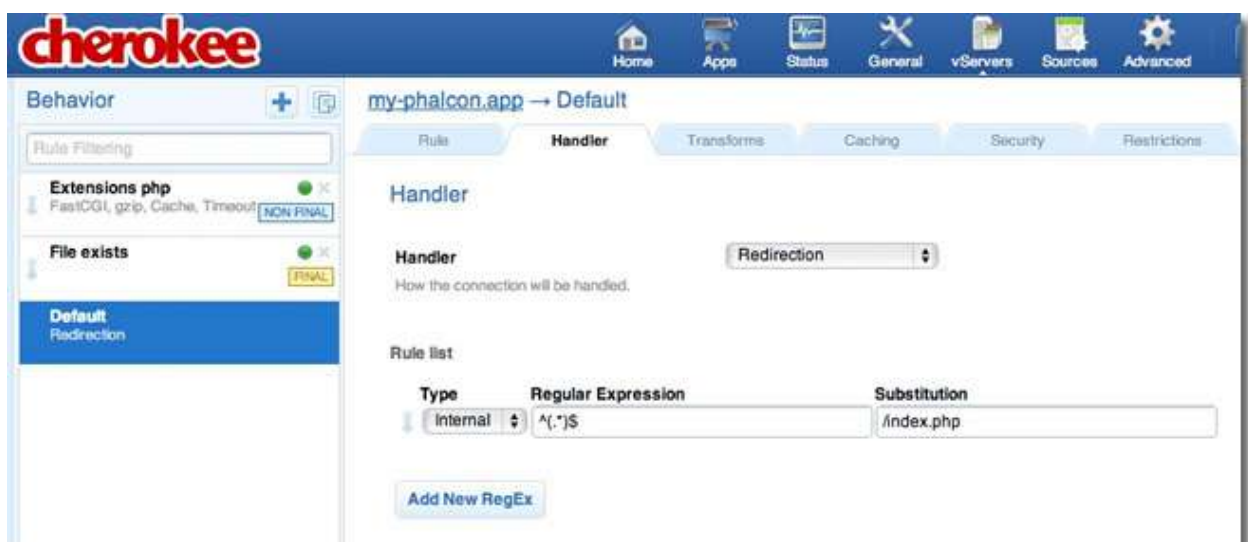
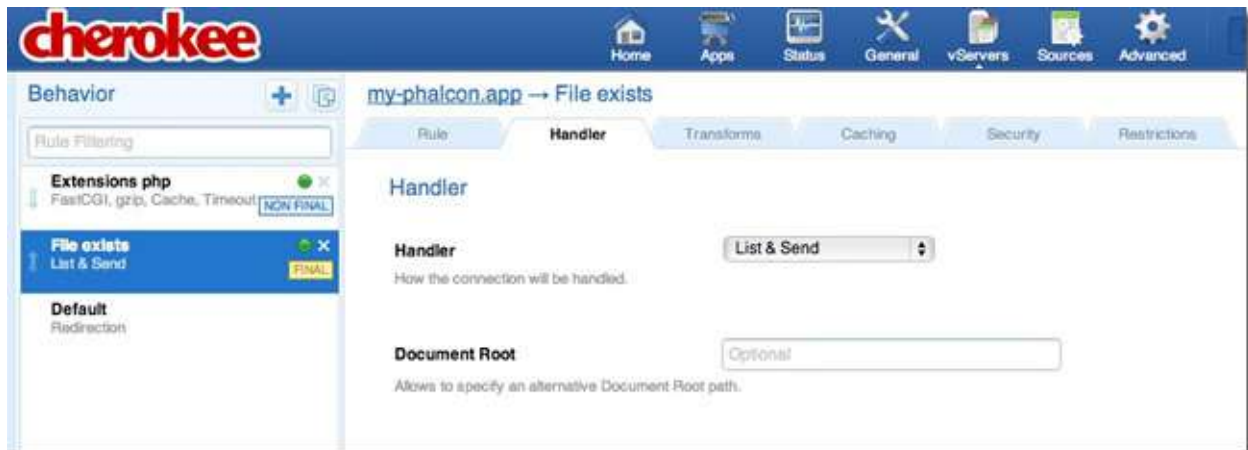
Behavior

Behavior Rules

Match	Handler	Auth	Root	Secure	Enc	Cache	Exp	Timeout	Shaping	Log	Final
Default	List & Send									✓	✓

Rule Management







Utilisation du serveur web interne à PHP

A partir de PHP 5.4.0, vous pouvez utiliser un serveur web [interne](#) pour le développement.

Tapez pour démarrer le serveur:

```
php -S localhost:8000 -t /public
```

Si vous souhaitez réécrire les URIs pour le fichier `index.php` utilisez le fichier routeur suivant (`.htrouter.php`):

```
<?php
if (!file_exists(__DIR__ . '/' . $_SERVER['REQUEST_URI'])) {
    $_GET['_url'] = $_SERVER['REQUEST_URI'];
}
return false;
```

et démarrez le serveur à partir du répertoire de base du projet avec:

```
php -S localhost:8000 -t /public .htrouter.php
```

Et ouvrez votre navigateur à l'adresse <http://localhost:8000/> pour vérifier que cela fonctionne.

2.1.2 Outils pour développeurs Phalcon

Ces outils sont une collection de scripts utiles pour générer un squelette de code. Des composants principaux de votre application peuvent être générés avec une simple commande, vous permettant de développer des applications facilement avec phalcon.

Si vous préférez utiliser la version web plutôt que la console ce [ticket de blog](#) donne plus d'informations.

Téléchargement

Vous pouvez télécharger ou cloner un packet multi-plateforme contenant les outils pour développeurs à partir de [Github](#).

Installation

Vous trouverez des informations détaillés pour chaque plateforme aux adresses suivantes :

Phalcon Developer Tools on Windows

These steps will guide you through the process of installing Phalcon Developer Tools for Windows.

Prerequisites

The Phalcon PHP extension is required to run Phalcon Tools. If you haven't installed it yet, please see the [Installation](#) section for instructions.

Download

You can download a cross platform package containing the developer tools from the [Download](#) section. Also you can clone it from [Github](#).

On the Windows platform, you need to configure the system PATH to include Phalcon tools as well as the PHP executable. If you download the Phalcon tools as a zip archive, extract it on any path of your local drive i.e. *c:\phalcon-tools*. You will need this path in the steps below. Edit the file "phalcon.bat" by right clicking on the file and selecting "Edit":

Change the path to the one you installed the Phalcon tools (set PTOOLSPATH=C:\phalcon-tools):

Save the changes.

Adding PHP and Tools to your system PATH

Because the scripts are written in PHP, you need to install it on your machine. Depending on your PHP installation, the executable can be located in various places. Search for the file php.exe and copy the path it is located in. For instance, if using the latest WAMP stack, PHP is located in: *C:\wampbin\php\php5.3.10\php.exe*.

From the Windows start menu, right mouse click on the "Computer" icon and select "Properties":

Click the "Advanced" tab and then the button "Environment Variables":

At the bottom, look for the section "System variables" and edit the variable "Path":

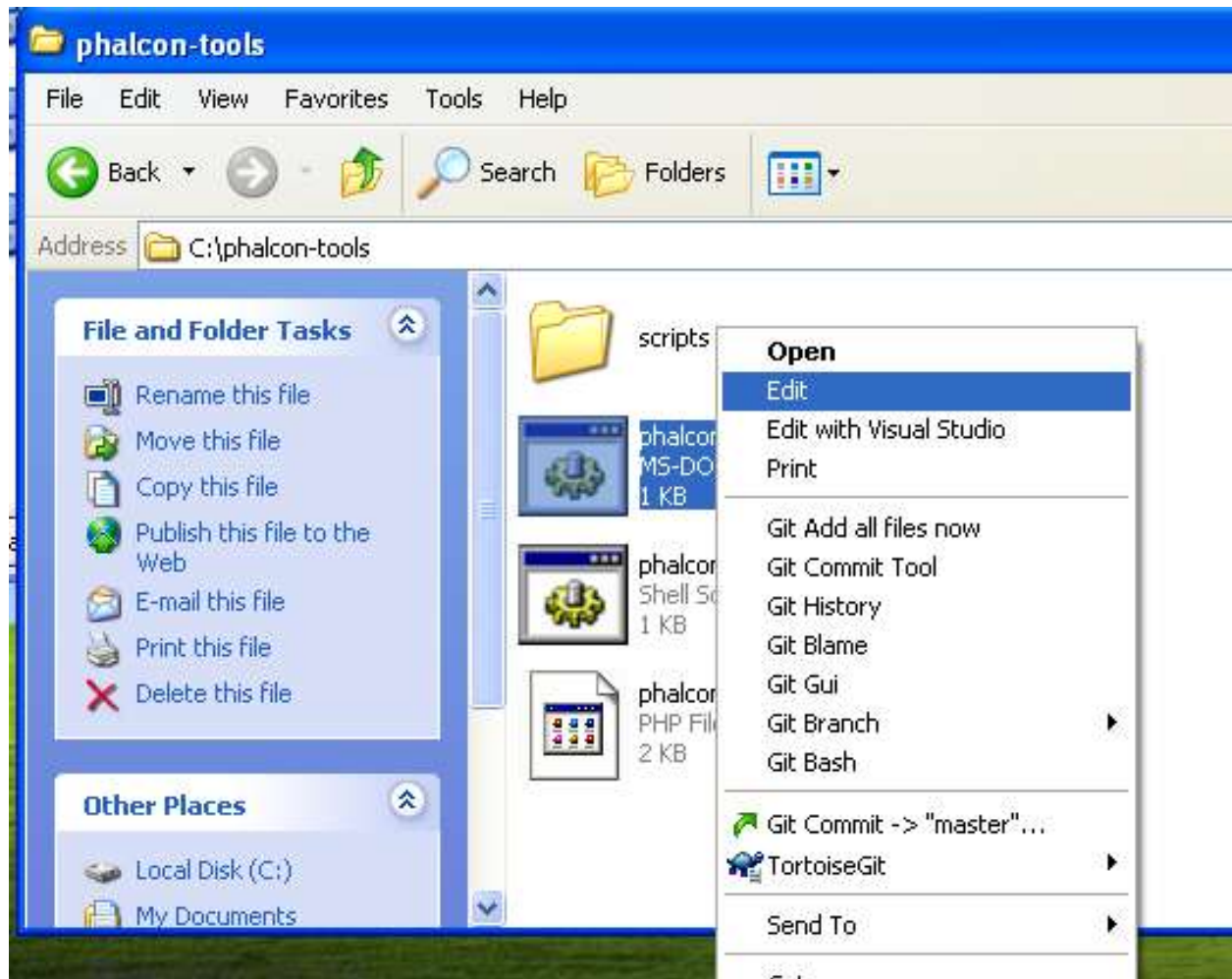
Be very careful on this step! You need to append at the end of the long string the path where your php.exe was located and the path where Phalcon tools are installed. Use the ";" character to separate the different paths in the variable:

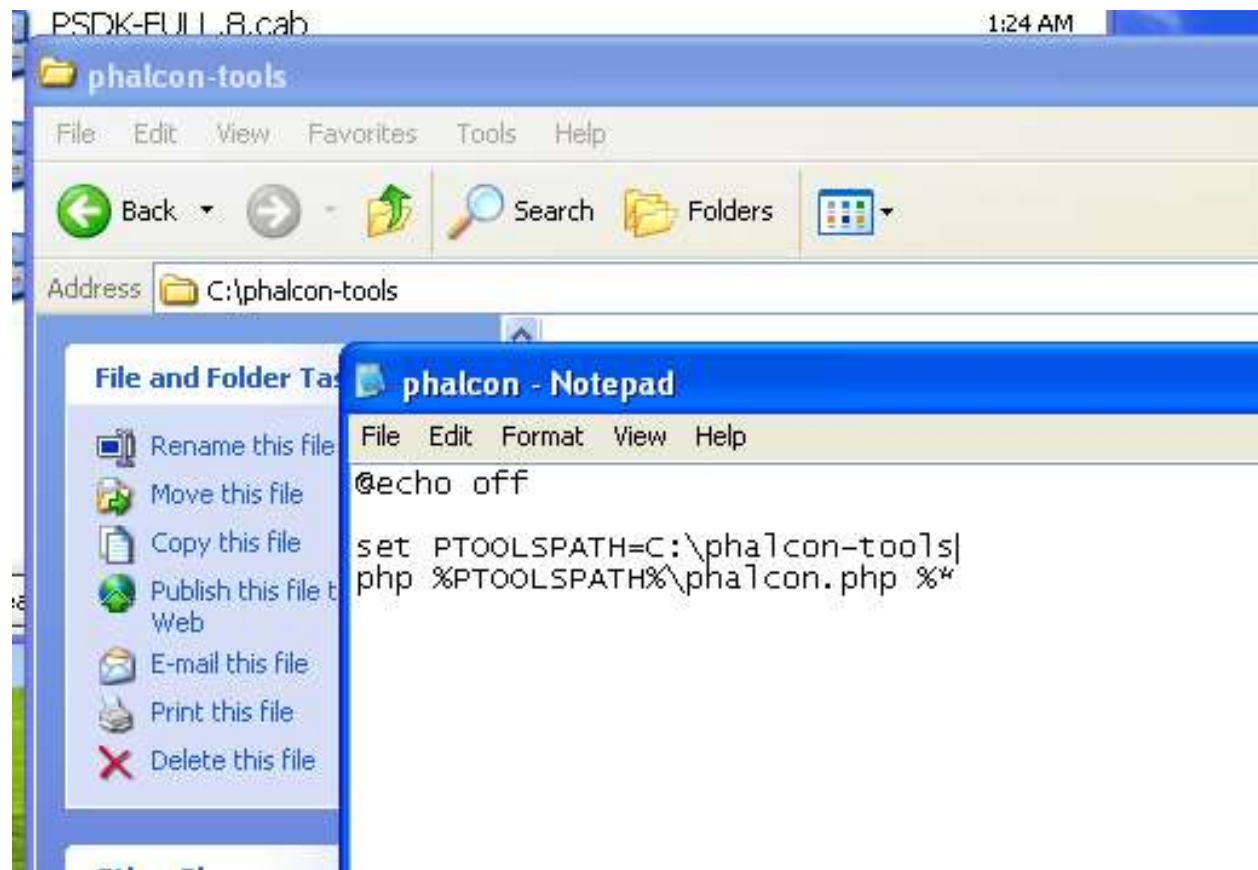
Accept the changes made by clicking "OK" and close the dialogs opened. From the start menu click on the option "Run". If you can't find this option, press "Windows Key" + "R".

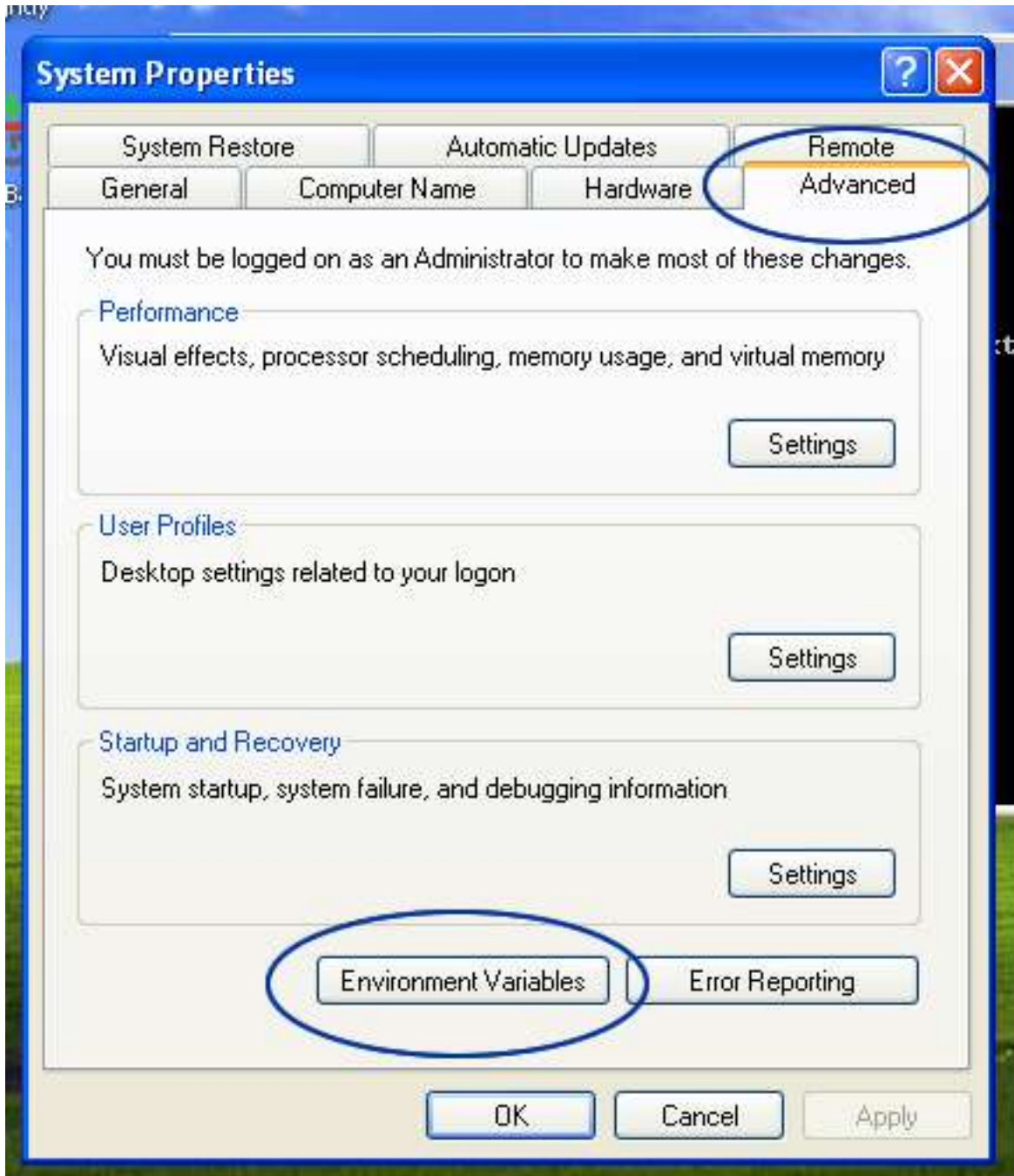
Type "cmd" and press enter to open the windows command line utility:

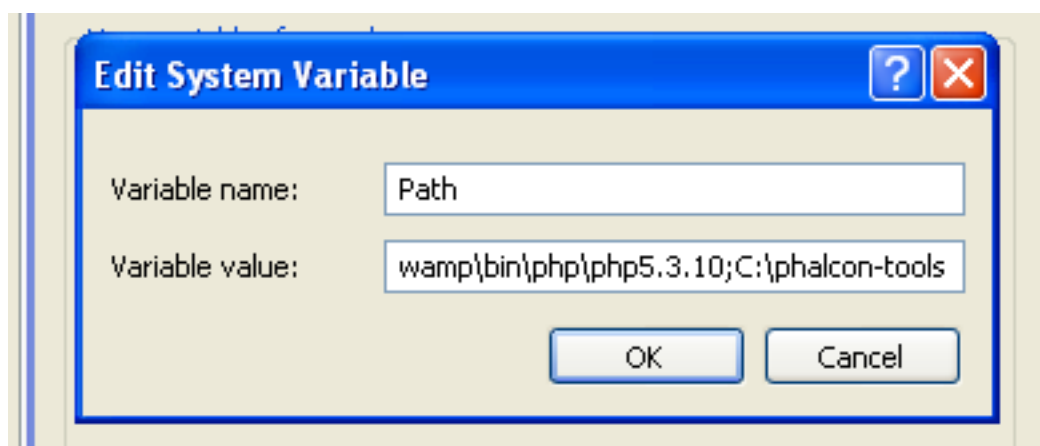
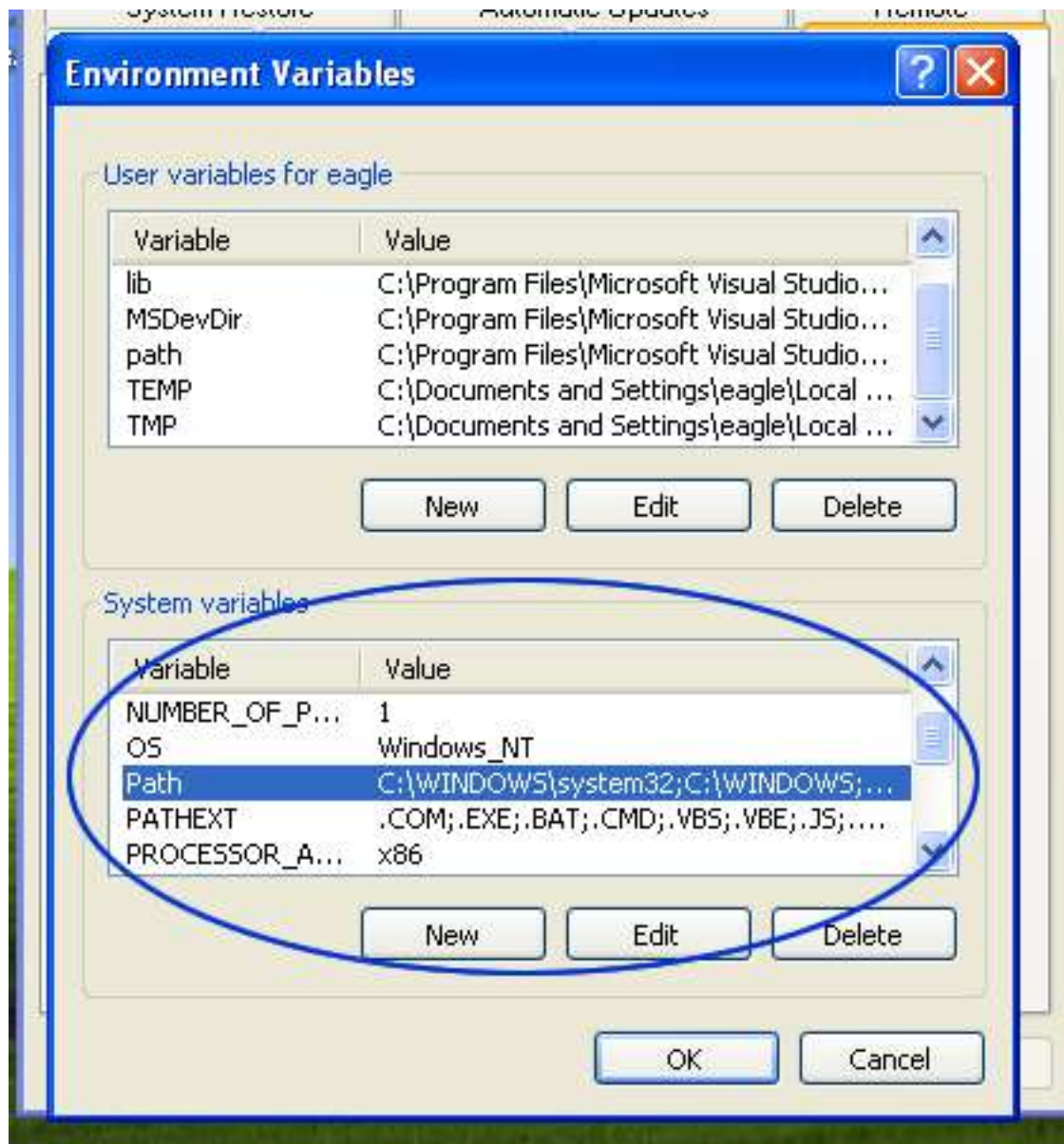
Type the commands "php -v" and "phalcon" and you will see something like this:

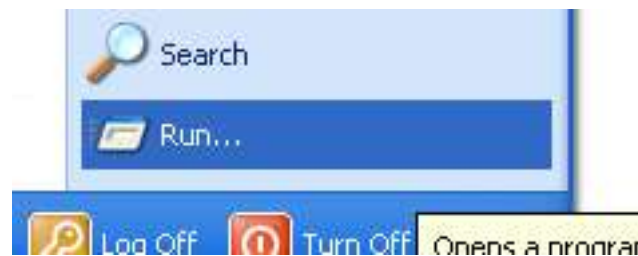
Congratulations you now have Phalcon tools installed!











```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\eagle>php -v
PHP 5.3.10 (cli) (built: Feb  2 2012 20:27:51)
Copyright (c) 1997-2012 The PHP Group
Zend Engine v2.3.0, Copyright (c) 1998-2012 Zend Technologies
    with Xdebug v2.1.2, Copyright (c) 2002-2011, by Derick Rethans

C:\Documents and Settings\eagle>phalcon
Phalcon: incorrect usage

C:\Documents and Settings\eagle>
```

Related Guides

- [Using Developer Tools](#)
- [Installation on OS X](#)
- [Installation on Linux](#)

Phalcon Developer Tools on Mac OS X

These steps will guide you through the process of installing Phalcon Developer Tools for OS/X.

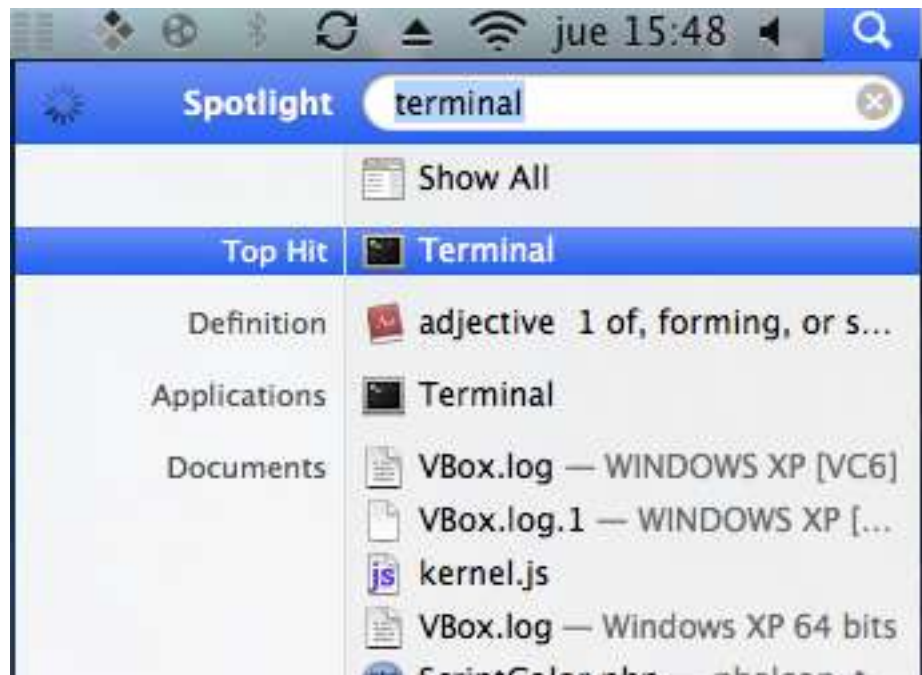
Prerequisites

The Phalcon PHP extension is required to run Phalcon Tools. If you haven't installed it yet, please see the [Installation](#) section for instructions.

Download

You can download a cross platform package containing the developer tools from the [Download](#) section. You can also clone it from [Github](#).

Open the terminal application:



Copy & Paste the commands below in your terminal:

```
git clone git://github.com/phalcon/phalcon-devtools.git
```

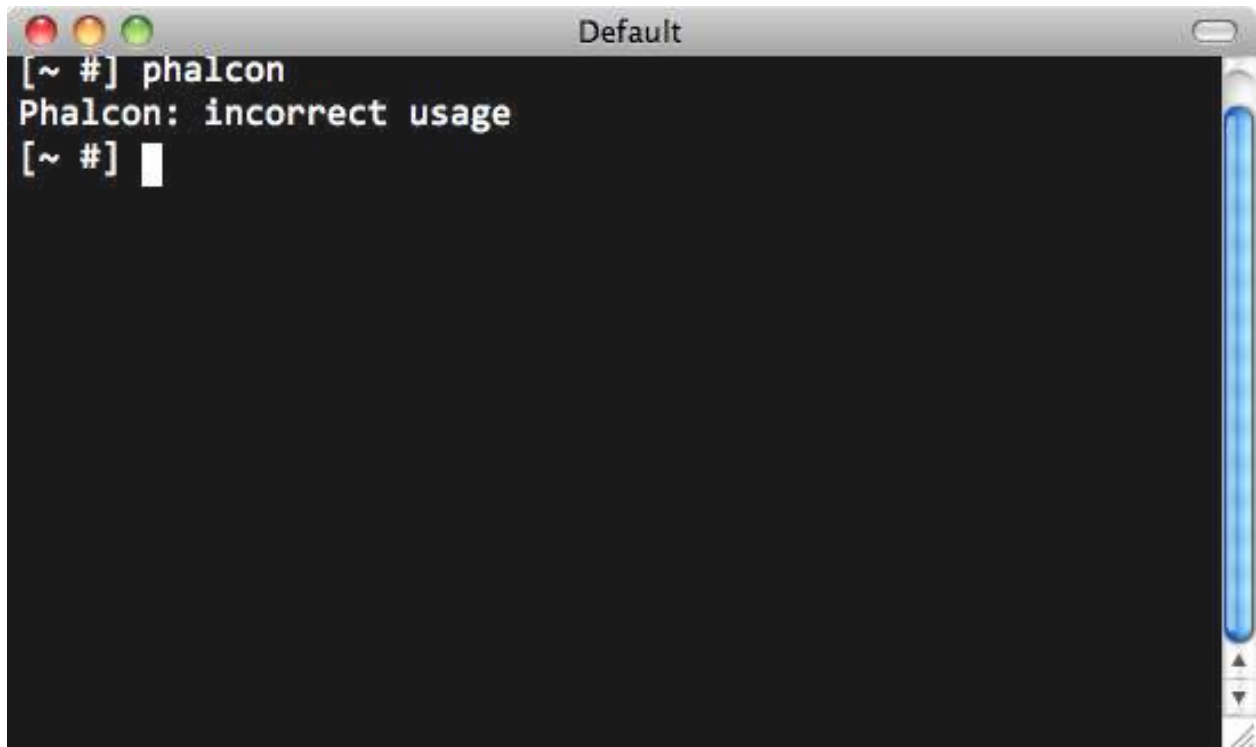
Then enter the folder where the tools were cloned and execute `". /phalcon.sh"`, (don't forget the dot at beginning of the command):

```
cd phalcon-devtools/  
. ./phalcon.sh
```

In the terminal window, type the following commands to create a symbolic link to the phalcon.php script:

```
ln -s ~/phalcon-tools/phalcon.php ~/phalcon-tools/phalcon  
chmod +x ~/phalcon-tools/phalcon
```

Type the command “phalcon” and you will see something like this:



Congratulations you now have Phalcon tools installed!

Related Guides

- *Using Developer Tools*
- *Installation on Windows*
- *Installation on Linux*

Phalcon Developer Tools on Linux

These steps will guide you through the process of installing Phalcon Developer Tools for Linux.

Prerequisites

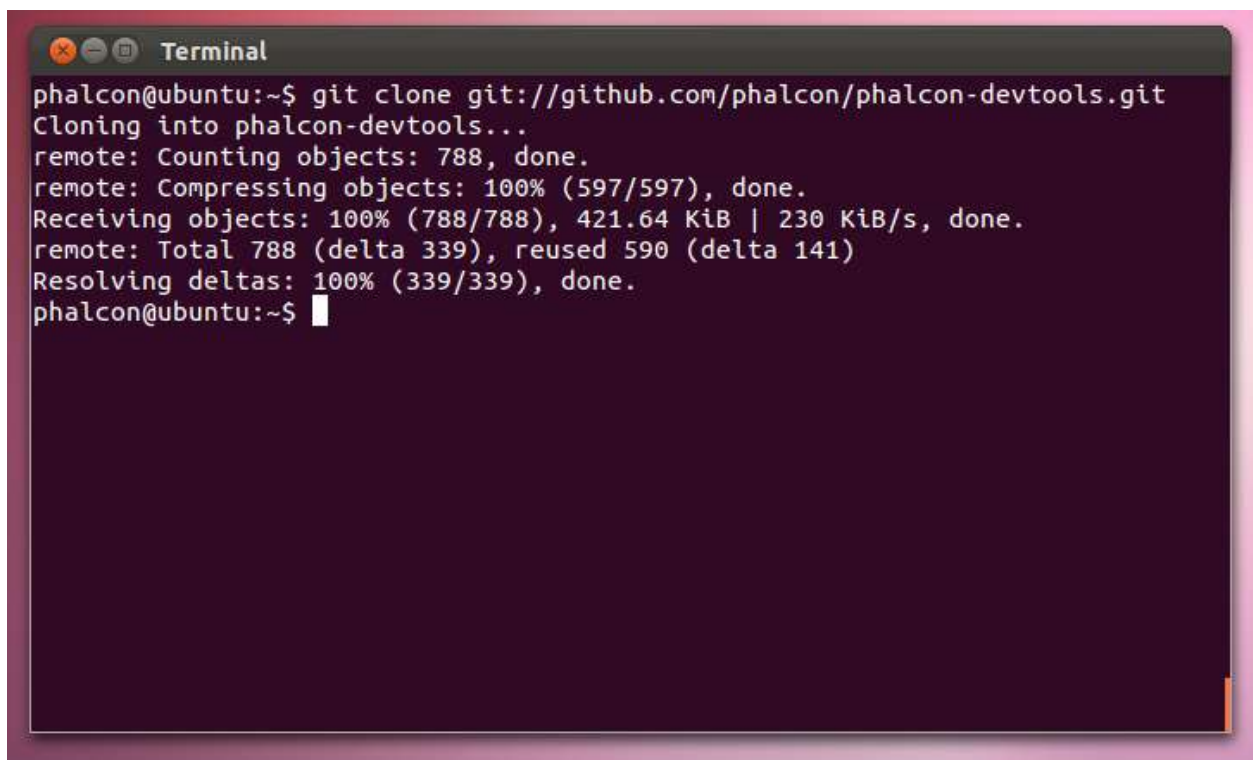
The Phalcon PHP extension is required to run Phalcon Tools. If you haven't installed it yet, please see the [Installation](#) section for instructions.

Download

You can download a cross platform package containing the developer tools from the [Download](#) section. Also you can clone it from [Github](#).

Open a terminal and type the command below:

```
git clone git://github.com/phalcon/phalcon-devtools.git
```



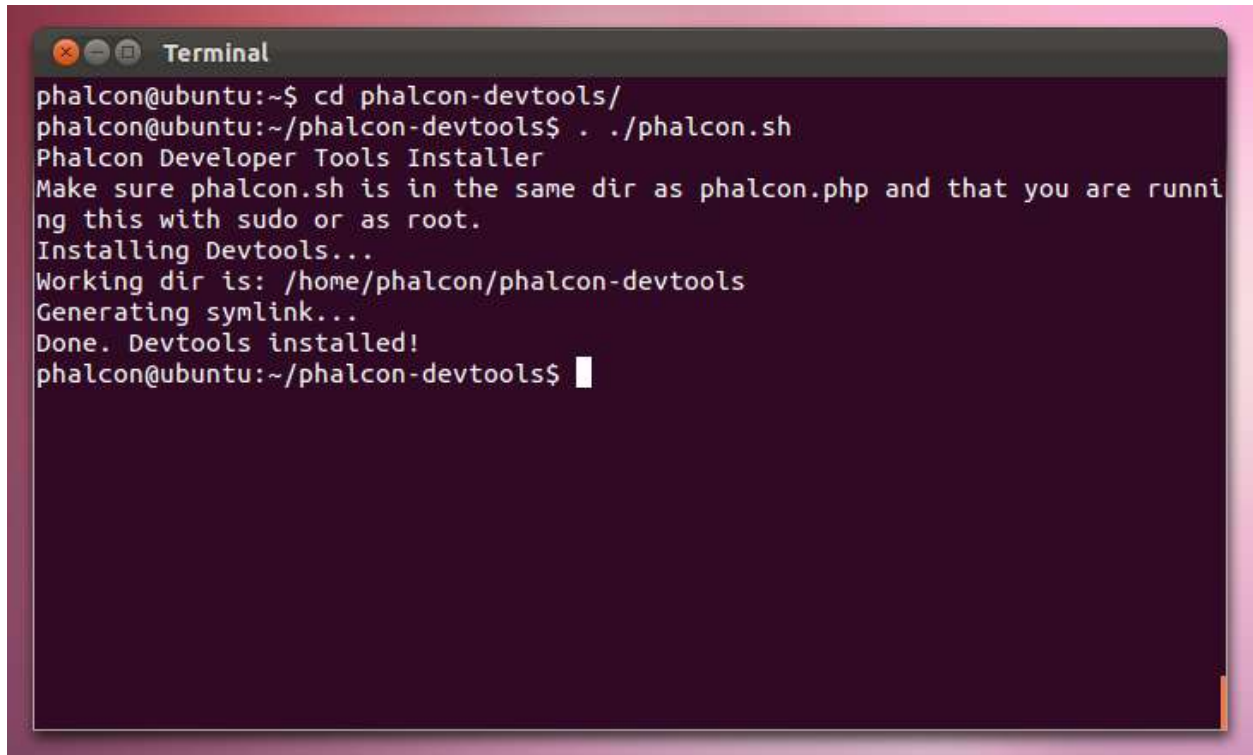
Then enter the folder where the tools were cloned and execute ". ./phalcon.sh", (don't forget the dot at beginning of the command):

```
cd phalcon-devtools/  
. ./phalcon.sh
```

Create a symbolink link to the phalcon.php script:

```
ln -s ~/phalcon-devtools/phalcon.php /usr/bin/phalcon  
chmod ugo+x /usr/bin/phalcon
```

Congratulations you now have Phalcon tools installed!

A terminal window titled "Terminal" with a dark background and light text. The user is at a prompt "phalcon@ubuntu:~\$". They enter "cd phalcon-devtools/" and the prompt changes to "phalcon@ubuntu:~/phalcon-devtools\$". They then enter "./phalcon.sh". The script outputs: "Phalcon Developer Tools Installer", "Make sure phalcon.sh is in the same dir as phalcon.php and that you are running this with sudo or as root.", "Installing Devtools...", "Working dir is: /home/phalcon/phalcon-devtools", "Generating symlink...", "Done. Devtools installed!", and returns to the prompt "phalcon@ubuntu:~/phalcon-devtools\$".

```
phalcon@ubuntu:~$ cd phalcon-devtools/
phalcon@ubuntu:~/phalcon-devtools$ ./phalcon.sh
Phalcon Developer Tools Installer
Make sure phalcon.sh is in the same dir as phalcon.php and that you are running this with sudo or as root.
Installing Devtools...
Working dir is: /home/phalcon/phalcon-devtools
Generating symlink...
Done. Devtools installed!
phalcon@ubuntu:~/phalcon-devtools$
```

Related Guides

- *Using Developer Tools*
- *Installation on Windows*
- *Installation on Mac*

Les commandes disponibles

Vous pouvez obtenir la liste des commandes Phalcon disponibles en tapant : `phalcon commands`

```
$ phalcon commands

Phalcon DevTools (3.0.0)

Available commands:
  commands      (alias of: list, enumerate)
  controller    (alias of: create-controller)
  module        (alias of: create-module)
  model         (alias of: create-model)
  all-models    (alias of: create-all-models)
  project       (alias of: create-project)
  scaffold      (alias of: create-scaffold)
  migration     (alias of: create-migration)
  webtools      (alias of: create-webtools)
```

Générer un squelette de Project

Vous pouvez utiliser les outils Phalcon pour générer un squelette de projet prédéfini pour vos applications. Par défaut le générateur de squelette de projet utilise le module de réécriture d'url (mod_rewrite) d'Apache. Ecrivez la ligne de commande suivante à l'endroit où vous désirez créer votre projet :

```
$ pwd

/Applications/MAMP/htdocs

$ phalcon create-project store
```

La structure suivante sera générée :

Vous pouvez ajouter le paramètre `--help` pour obtenir de l'aide sur l'utilisation de certains scripts:

```
$ phalcon project --help

Phalcon DevTools (3.0.0)

Help:
  Creates a project

Usage:
  project [name] [type] [directory] [enable-webtools]

Arguments:
  help      Shows this help text

Example
  phalcon project store simple

Options:
  --name                Name of the new project
  --enable-webtools     Determines if webtools should be enabled [optional]
  --directory=s        Base path on which project will be created [optional]
  --type=s              Type of the application to be generated (cli, micro, simple, ↵
↵modules)
  --template-path=s    Specify a template path [optional]
  --use-config-ini      Use a ini file as configuration file [optional]
  --trace               Shows the trace of the framework in case of exception. ↵
↵[optional]
  --help               Shows this help
```

Accédez à l'url de votre projet et vous obtiendrez ceci :

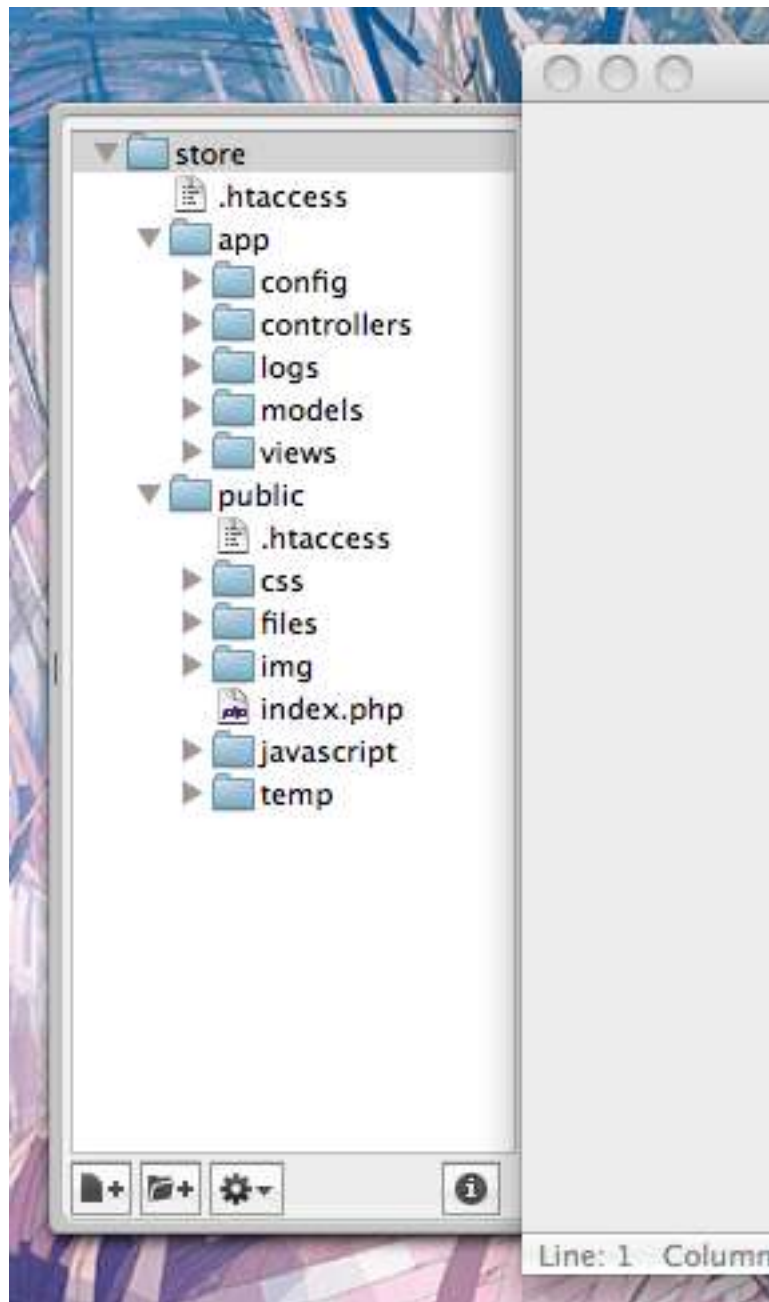
Générer des contrôleurs

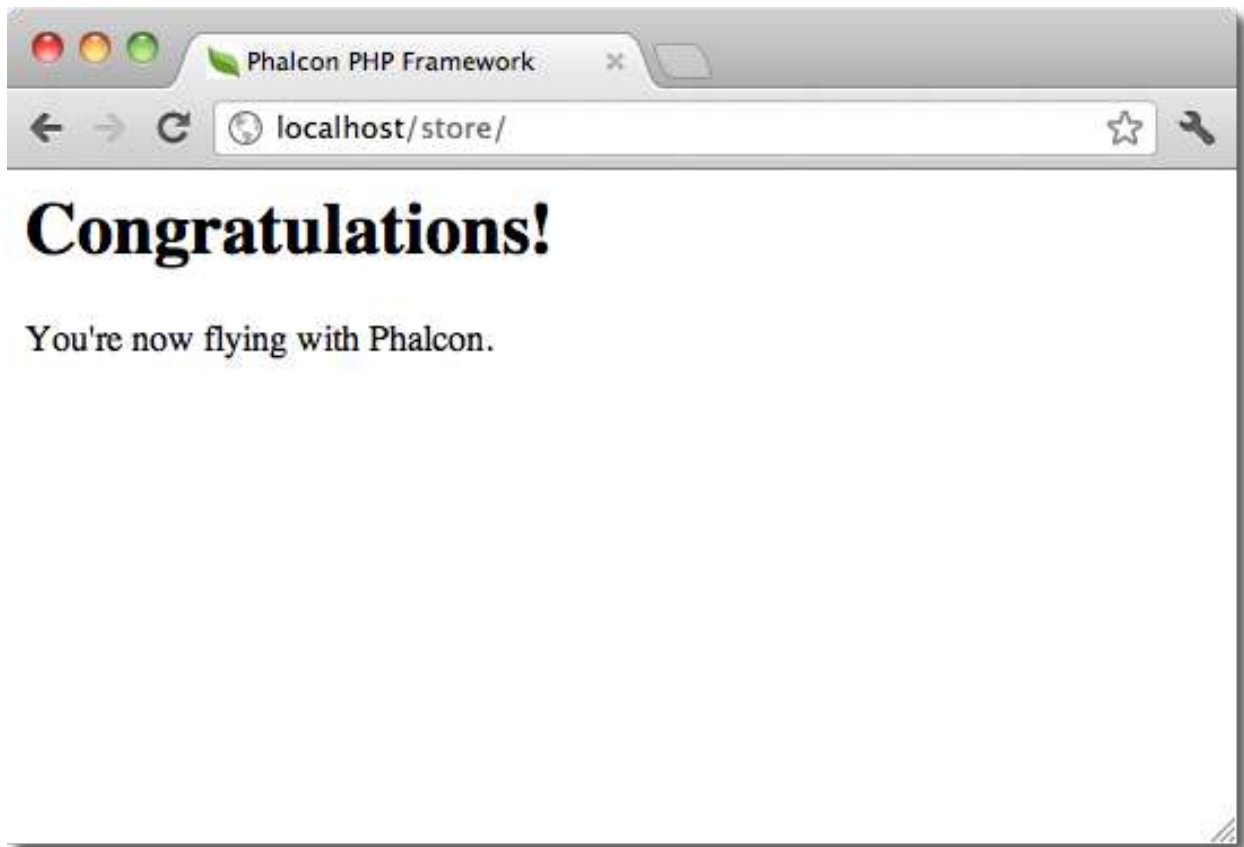
La commande "create-controller" génère un contrôleur type. Il est important de faire cette commande à l'intérieur du dossier qui contient le projet Phalcon.

```
$ phalcon create-controller --name test
```

Le code suivant sera généré par le script :

```
<?php
```



```
use Phalcon\Mvc\Controller;

class TestController extends Controller
{
    public function indexAction()
    {

    }
}
```

Préparez les paramètres de base de données

Quand un projet est généré avec les outils pour développeurs, un fichier de configuration sera disponible dans *app/config/config.ini*. Pour générer un modèle vous devrez changer les paramètres utilisés pour se connecter à la base de données.

Modifiez la section “database” dans votre fichier config.ini:

```
[database]
adapter  = Mysql
host     = "127.0.0.1"
username = "root"
password = "secret"
dbname   = "store_db"

[phalcon]
```

```

controllersDir = "../app/controllers/"
modelsDir      = "../app/models/"
viewsDir       = "../app/views/"
baseUri        = "/store/"

```

Générer des modèles

Il y a plusieurs manières de générer des modèles. Vous pouvez créer tous les modèles à partir de la connexion par défaut à la base de données ou de manière plus sélective. Les modèles peuvent avoir des attributs public pour la représentation des champs ou des accesseurs peuvent être utilisés.

Options:

--name=s	Table name
--schema=s	Name of the schema. [optional]
--namespace=s	Model's namespace [optional]
--get-set	Attributes will be protected and have setters/getters. [optional]
--extends=s	Model extends the class name supplied [optional]
--excludefields=l	Excludes fields defined in a comma separated list [optional]
--doc	Helps to improve code completion on IDEs [optional]
--directory=s	Base path on which project will be created [optional]
--force	Rewrite the model. [optional]
--trace	Shows the trace of the framework in case of exception. [optional]
--mapcolumn	Get some code for map columns. [optional]
--abstract	Abstract Model [optional]

La manière la plus simple de générer un modèle est d'écrire ceci:

```
$ phalcon model products
```

```
$ phalcon model --name tablename
```

Tous les champs de la table seront déclarés public pour un accès direct.

```

<?php
use Phalcon\Mvc\Model;

class Products extends Model
{
    /**
     * @var integer
     */
    public $id;

    /**
     * @var integer
     */
    public $typesId;
}

```

```
/**
 * @var string
 */
public $name;

/**
 * @var string
 */
public $price;

/**
 * @var integer
 */
public $quantity;

/**
 * @var string
 */
public $status;
}
```

En ajoutant le paramètre `-get-set`, vous pouvez générer les champs avec des variables protégés et y accéder avec les accesseurs. Ces méthodes peuvent aider à la mise en œuvre de la logique métier à l'intérieur des accesseurs.

```
<?php

use Phalcon\Mvc\Model;

class Products extends Model
{
    /**
     * @var integer
     */
    protected $id;

    /**
     * @var integer
     */
    protected $typesId;

    /**
     * @var string
     */
    protected $name;

    /**
     * @var string
     */
    protected $price;

    /**
     * @var integer
     */
    protected $quantity;

    /**
     * @var string
     */
}
```

```

protected $status;

/**
 * Method to set the value of field id
 *
 * @param integer $id
 */
public function setId($id)
{
    $this->id = $id;
}

/**
 * Method to set the value of field typeId
 *
 * @param integer $typeId
 */
public function setTypeId($typeId)
{
    $this->typeId = $typeId;
}

// ...

/**
 * Returns the value of field status
 *
 * @return string
 */
public function getStatus()
{
    return $this->status;
}
}

```

Une fonctionnalité intéressante de la génération de modèle est qu’il conserve les changements fait par les développeurs entre les générations de code. Cela permet d’ajouter ou de supprimer des champs ou des propriétés sans craindre de perdre les changements déjà apportés au modèle. La vidéo suivante vous montre comment cela fonctionne :

L’échaffaudage d’un CRUD

“L’échaffaudage” est un moyen rapide de générer la plupart des parties importante d’une application. Si vous voulez créer les modèles, vues et les contrôleurs pour une nouvelle ressource en une seule action, l’échaffaudage est l’outil qu’il vous faut.

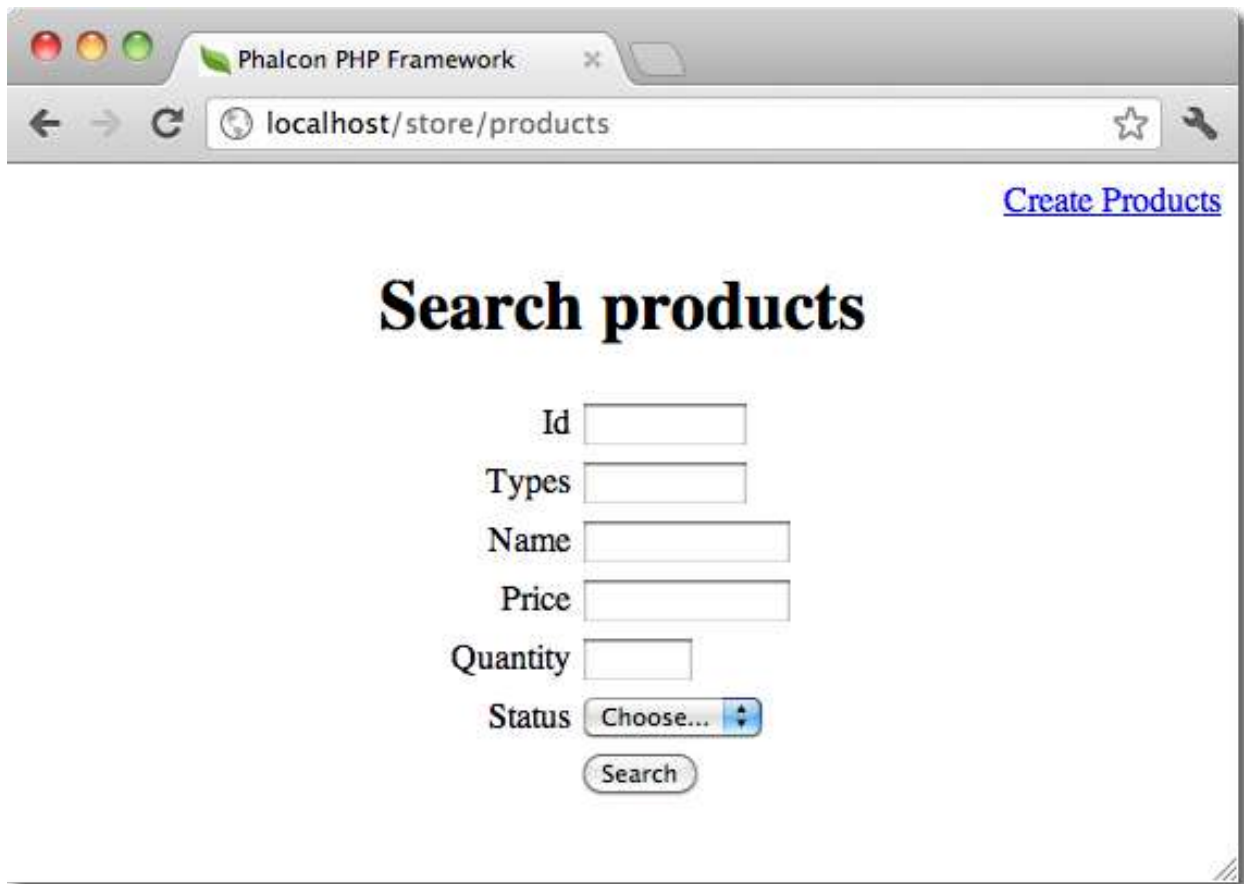
Une fois le code généré, il lui faudra être modifier pour répondre à vos besoins. Beaucoup de développeurs évitent l’échaffaudage complètement, choisissant de tout écrire eux-même. La génération de code peut aider à mieux comprendre comment le framework fonctionne ou bien pour développer des prototypes. Sur l’exemple de code suivant illustre le liner sur la base de la table “products”:

```
$ phalcon scaffold --table-name products
```

Le générateur d’échaffaudage va créer plusieurs fichiers dans votre application ainsi que quelques dossiers. Voici un aperçu rapide de ce qui sera généré:

Fichier	Objectif
app/controllers/ProductsController.php	Le Controller de Products
app/models/Products.php	Le model Products Products
app/views/layout/products.phtml	Controller layout for Products
app/views/products/new.phtml	Vue pour l'action "new"
app/views/products/edit.phtml	Vue pour l'action "edit"
app/views/products/search.phtml	Vue pour l'action "search"

En naviguant sur la page du contrôleur récemment généré, on voit un formulaire de recherche et un lien pour créer un nouveau produit:

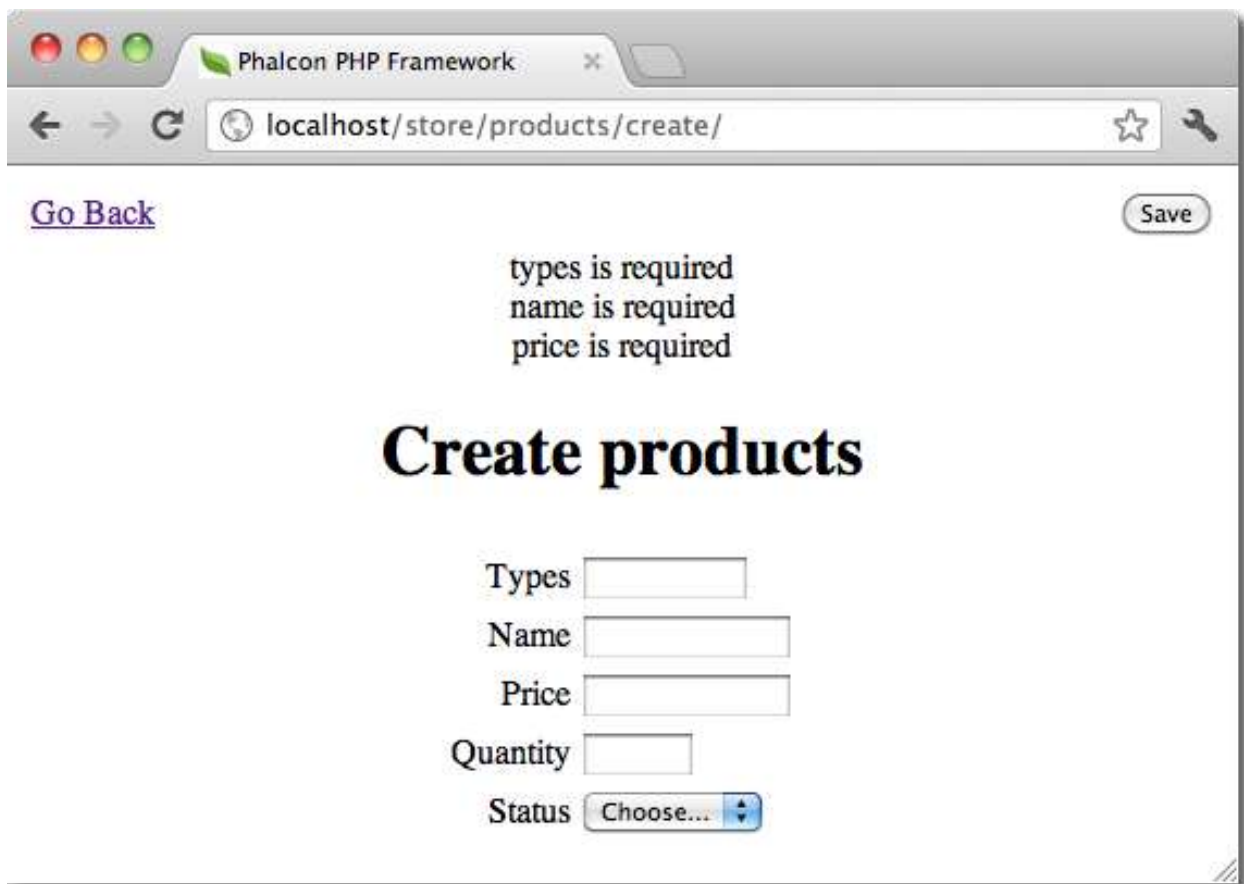


La création de page vous permet de créer des produits en appliquant les validations du modèle Products. Phalcon va automatiquement vérifier les champs nuls et générer des avertissements pour ceux qui sont requis.

Après avoir effectué une recherche, un composant de pagination est disponible pour voir les résultats. Utilisez les liens "Edit" ou "Delete" sur chaque ligne pour effectuer l'action d'édition ou de suppression.

L'interface web des outils

Si vous préférez il est tout à fait possible d'utiliser les outils de développeur Phalcon à partir d'une interface web. Regardez la vidéo suivante pour voir comment faire :



A screenshot of a web browser window. The title bar says 'Phalcon PHP Framework'. The address bar shows 'localhost/store/products/create/'. The page has a 'Go Back' link on the left and a 'Save' button on the right. In the center, there are three lines of text: 'types is required', 'name is required', and 'price is required'. Below this is a large heading 'Create products'. Under the heading are five form fields: 'Types' (text input), 'Name' (text input), 'Price' (text input), 'Quantity' (text input), and 'Status' (dropdown menu with 'Choose...' selected).

[Go Back](#) Save

types is required
name is required
price is required

Create products

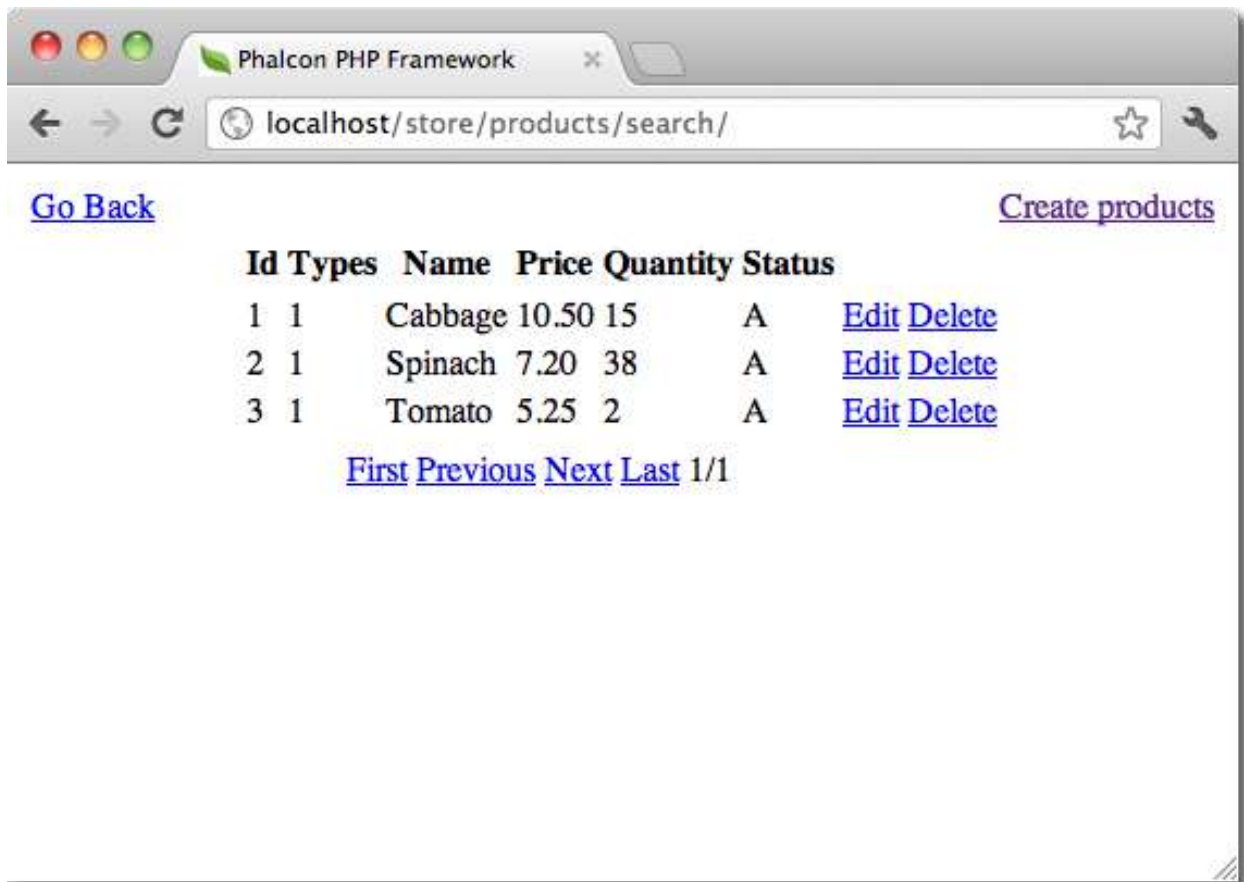
Types

Name

Price

Quantity

Status Choose...



Intégration des outils sur l'IDE PhpStorm

La vidéo suivante vous montre comment intégrer les outils de développeur avec l'IDE PhpStorm. La configuration peut facilement être adaptée à d'autres IDE pour PHP.

Conclusion

Les outils pour développeurs Phalcon fournissent un moyen simple de générer du code pour votre application. Cela réduit le temps de développement et diminue le nombre potentiel d'erreur de code.

2.2 Tutorials

2.2.1 Tutoriel 1: Apprenons par l'exemple

Au travers de ce premier tutoriel nous allons vous emmener dans la création d'une application avec un simple formulaire d'inscription en partant de zéro. Nous expliquerons les aspects élémentaire du comportement du framework. Si vous êtes intéressés par des outils de génération de code pour Phalcon, allez voir *outils pour développeur*.

La meilleure façon d'utiliser ce guide est de le suivre étape par étape. Vous pouvez récupérer le code complet *ici*.

Structure des fichiers

Phalcon n'impose pas une structure particulière des fichiers pour le développement d'application. Comme il est faiblement couplé vous pouvez réaliser de puissantes applications avec une structure de fichiers qui vous convienne.

Pour les besoins de ce tutoriel et comme point de départ, nous suggérons cette simple structure:

```
tutorial/  
  app/  
    controllers/  
    models/  
    views/  
  public/  
    css/  
    img/  
    js/
```

Notez que nous n'avons pas besoin d'un dossier "library" en rapport avec Phalcon. Ce framework est disponible en mémoire prêt à l'emploi.

Avant de poursuivre, soyez certains d'avoir installé Phalcon avec succès et configuré un serveur parmi *Nginx*, *Apache* ou *Cherokee*.

Amorce

Le premier fichier que nous devons créer est le fichier d'amorce. Ce fichier est vraiment important; comme il sert de base à votre application, il vous donne le contrôle sur tous ses aspects. Dans ce fichier vous pouvez y mettre l'initialisation des composants ainsi que définir le comportement de l'application.

Il est responsable de trois choses:

1. Préparer le chargeur automatique
2. Configurer l'injecteur de dépendances (DI).

3. Gérer les requête à l'application.

Chargement automatique

La première chose que nous trouvons dans l'amorce est l'inscription d'un chargeur automatique. Ceci permet de charger les classes des contrôleurs et des modèles de l'application. Par exemple vous pouvez inscrire un ou plusieurs répertoire de contrôleurs améliorant ainsi la flexibilité de l'application. Dans notre exemple nous avons utilisé le composant *Phalcon\Loader*.

Grâce à ceci, nous pouvons charger les classes selon différentes stratégies, mais pour cet exemple, nous avons choisi de placer les classes dans des dossiers prédéfinis:

```
<?php
use Phalcon\Loader;

// ...

$loader = new Loader();

$loader->registerDirs(
    [
        "../app/controllers/",
        "../app/models/",
    ]
);

$loader->register();
```

Gestion de dépendance

Un concept très important qui doit être compris avec Phalcon est son *conteneur d'injection de dépendances*. Cela peut sembler compliqué mais il est en réalité très simple et pratique.

Un conteneur de service est un sac où nous stockons généralement les services que votre application doit utiliser pour fonctionner. A chaque fois que le framework a besoin d'un composant, il interroge le conteneur en utilisant une convention de nommage pour le service. Comme Phalcon est un framework fortement découplé, agit comme un ciment facilitant l'intégration des différents composants en parvenant à les faire travailler ensemble d'une façon transparente.

```
<?php
use Phalcon\Di\FactoryDefault;

// ...

// Create a DI
$di = new FactoryDefault();
```

Phalcon\Di\FactoryDefault est une variante de *Phalcon\Di*. Afin de faciliter les choses, la plupart des composants fournis avec Phalcon sont inscrits. Ainsi nous n'aurons pas à les inscrire un par un. Nous verrons plus tard qu'il n'y a aucun problème à remplacer un service d'usine.

Dans la partie suivante, nous inscrivons le service "view" en indiquant au framework le répertoire où il trouvera les définitions de vues. Comme les vues ne correspondent pas à des classes elles ne peuvent pas prises en compte par le chargeur automatique.

Les service peuvent être inscrits de plusieurs façon, mais dans ce tutoriel nous utiliserons une [fonction anonyme](#):

```
<?php

use Phalcon\Mvc\View;

// ...

// Configuration du composant vue
$di->set(
    "view",
    function () {
        $view = new View();

        $view->setViewsDir("../app/views/");

        return $view;
    }
);
```

Ensuite nous inscrivons une URI de base afin que toutes les URIs générées par Phalcon incluent le dossier “tutorial” que nous avons défini préalablement. Ceci deviendra important plus loin dans ce tutoriel lorsque nous utiliserons la classe *Phalcon\Tag* pour créer des hyperliens.

```
<?php

use Phalcon\Mvc\Url as UrlProvider;

// ...

// Définition d'une URI de base afin que les URIs générées incluent le dossier
↳ "tutorial"
$di->set(
    "url",
    function () {
        $url = new UrlProvider();

        $url->setBaseUri("/tutorial/");

        return $url;
    }
);
```

Traitement des requêtes

Dans la dernière partie de ce fichier nous trouvons *Phalcon\Mvc\Application*. Son rôle est de préparer l’environnement pour les requêtes, de router les requêtes entrante et de répartir entre les différentes actions trouvées; il assemble les réponses et les retourne dès que le processus est complet.

```
<?php

use Phalcon\Mvc\Application;

// ...

$application = new Application($di);
```

```
$response = $application->handle();  
  
$response->send();
```

Tout mettre ensemble

Le fichier `tutorial/public/index.php` doit ressembler à ceci:

```
<?php  
  
use Phalcon\Loader;  
use Phalcon\Mvc\View;  
use Phalcon\Mvc\Application;  
use Phalcon\Di\FactoryDefault;  
use Phalcon\Mvc\Url as UrlProvider;  
use Phalcon\Db\Adapter\Pdo\Mysql as DbAdapter;  
  
  
// Inscription du chargeur automatique  
$loader = new Loader();  
  
$loader->registerDirs(  
    [  
        "../app/controllers/",  
        "../app/models/",  
    ]  
);  
  
$loader->register();  
  
  
// Création du DI  
$di = new FactoryDefault();  
  
// Configuration du composant vue  
$di->set(  
    "view",  
    function () {  
        $view = new View();  
  
        $view->setViewsDir("../app/views/");  
  
        return $view;  
    }  
);  
  
// Définition d'une URI de base afin que les URIs générées incluent le dossier  
↪ "tutorial"  
$di->set(  
    "url",  
    function () {  
        $url = new UrlProvider();
```

```

        $url->setBaseUri("/tutorial/");

        return $url;
    }
);

$application = new Application($di);

try {
    // Gestion de la requête
    $response = $application->handle();

    $response->send();
} catch (\Exception $e) {
    echo "Exception: ", $e->getMessage();
}

```

Comme vous pouvez le voir, le fichier d’amorce est vraiment court et ne nécessite pas l’inclusion de fichier supplémentaire. Nous avons réalisé une application MVC en moins de 30 lignes de code.

Création d’un contrôleur

Par défaut Phalcon recherche un contrôleur nommé “Index”. Ceci est le point de départ lorsqu’aucun contrôleur ou action est transmise dans la requête. Ce contrôleur index (app/controllers/IndexController.php) ressemble à:

```

<?php

use Phalcon\Mvc\Controller;

class IndexController extends Controller
{
    public function indexAction()
    {
        echo "<h1>Hello!</h1>";
    }
}

```

Les classes contrôleur doit avoir le suffixe “Controller” et les actions du contrôleur doivent avoir le suffixe “Action”. Si vous accédez à l’application depuis votre navigateur, vous devez quelque chose comme:

Félicitations ! Vous volez avec Phalcon !

Sortie vers une vue

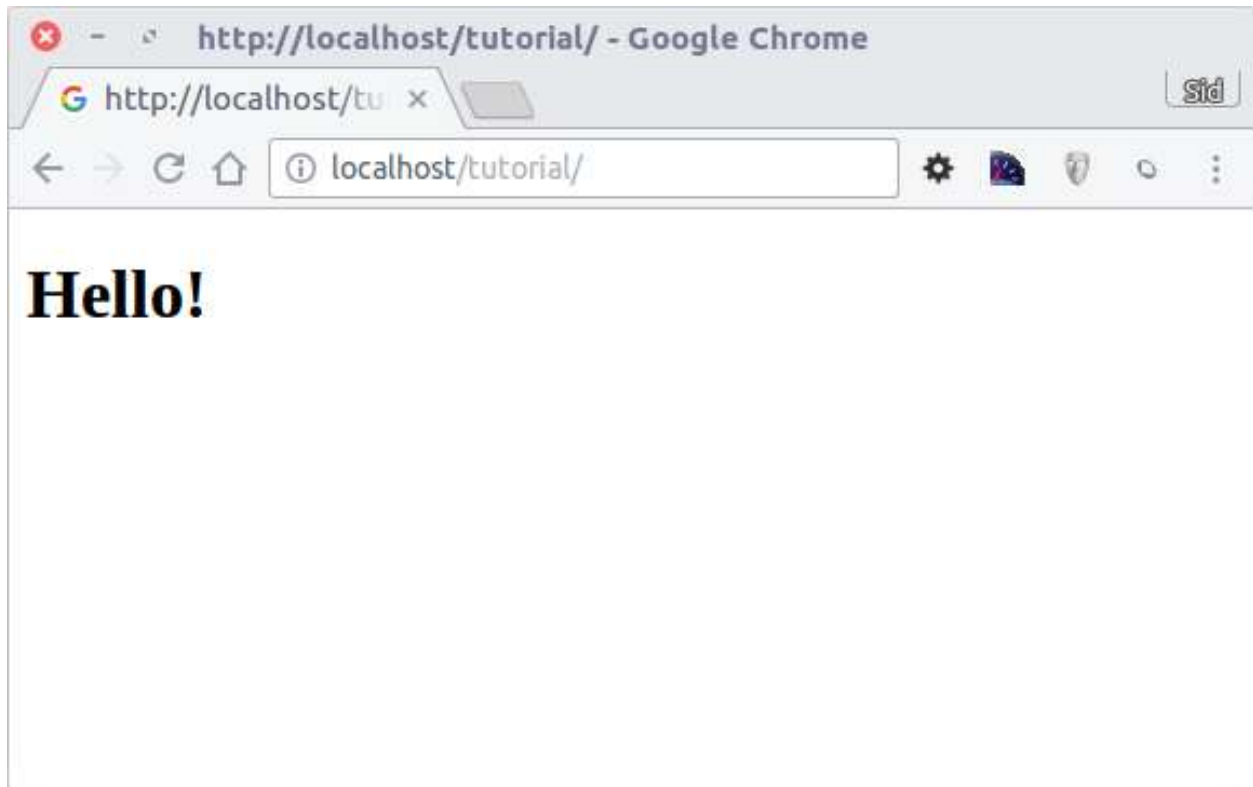
Les sorties à l’écran depuis le contrôleur est parfois nécessaire mais indésirable comme l’attestent la plupart des puristes de la communauté MVC. Tout doit être transmis à la vue qui est responsable de l’affichage des données à l’écran. Phalcon recherche une vue qui porte le même nom que la dernière action exécutée dans un répertoire qui porte le nom du dernier contrôleur exécuté. Dans notre cas (app/views/index/index.phtml):

```

<?php echo "<h1>Hello!</h1>";

```

Notre contrôleur (app/controllers/IndexController.php) contient maintenant une définition d’action vide:



```
<?php
use Phalcon\Mvc\Controller;

class IndexController extends Controller
{
    public function indexAction()
    {
    }
}
```

La sortie dans le navigateur doit rester la même. Le composant statique *Phalcon\Mvc\View* est automatiquement créé à la fin de l'exécution de l'action. Apprenez plus sur *l'utilisation des vues ici*.

Conception du formulaire d'inscription

Modifions maintenant le fichier vue `index.phtml` afin d'ajouter un lien vers un nouveau contrôleur appelé "signup". L'objectif est de permettre aux utilisateurs de s'inscrire dans notre application.

```
<?php
echo "<h1>Hello!</h1>";

echo PHP_EOL;

echo PHP_EOL;
```

```
echo $this->tag->linkTo(
    "signup",
    "Sign Up Here!"
);
```

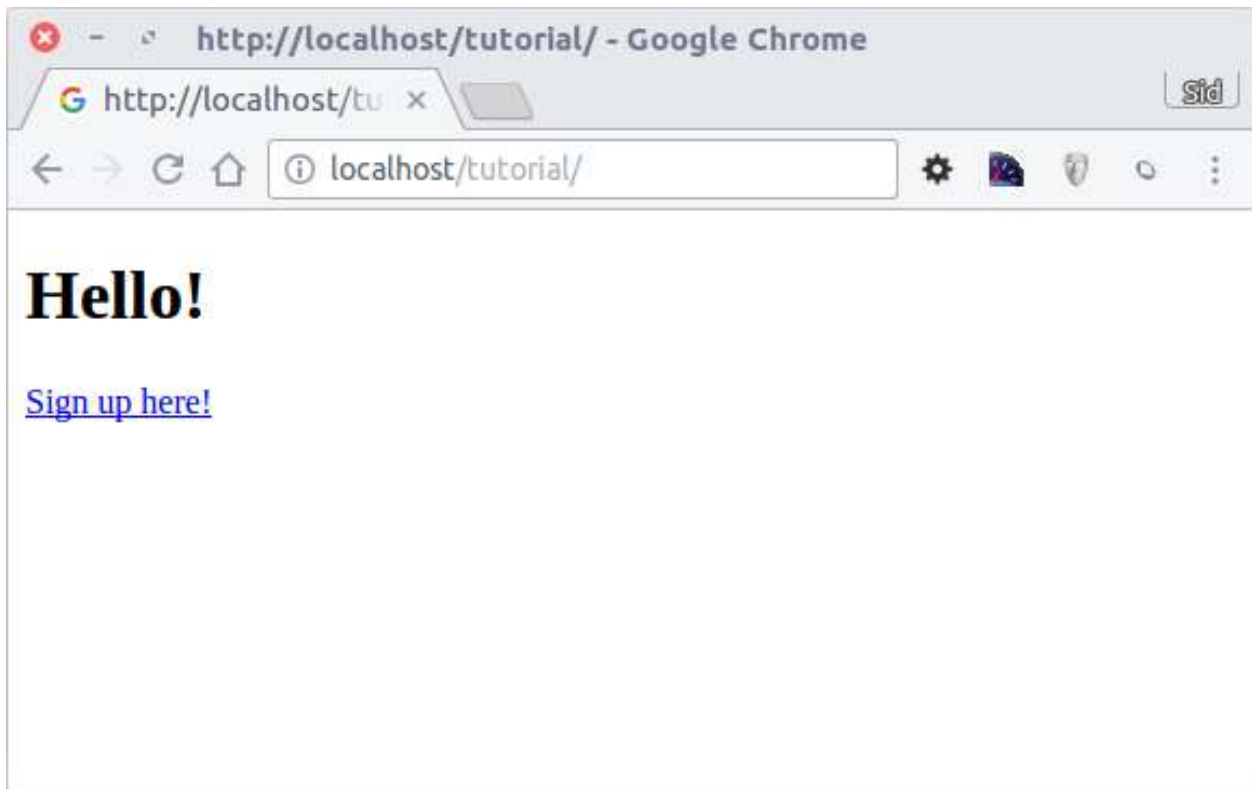
Le code HTML généré affiche une balise ancre HTML (“a”) désignant un nouveau contrôleur:

```
<h1>Hello!</h1>

<a href="/tutorial/signup">Sign Up Here!</a>
```

Pour générer la balise nous utilisons la classe *Phalcon\Tag*. C’est une classe utilitaire qui nous permet de construire des balises HTML en respectant les conventions du framework. Comme cette classe est également un service inscrite dans le DI nous utilisons `$this->tag` pour y accéder.

Un article plus détaillé concernant la génération HTML peut être *trouvée ici*



Voici le contrôleur Signup (`app/controllers/SignupController.php`):

```
<?php

use Phalcon\Mvc\Controller;

class SignupController extends Controller
{
    public function indexAction()
    {
    }
}
```

L'action `index` vide permet un passage propre à la vue qui contient la définition du formulaire (`app/views/signup/index.phtml`):

```
<h2>
    Sign up using this form
</h2>

<?php echo $this->tag->form("signup/register"); ?>

    <p>
        <label for="name">
            Name
        </label>

        <?php echo $this->tag->textField("name"); ?>
    </p>

    <p>
        <label for="email">
            E-Mail
        </label>

        <?php echo $this->tag->textField("email"); ?>
    </p>

    <p>
        <?php echo $this->tag->submitButton("Register"); ?>
    </p>

</form>
```

L’affichage du formulaire dans votre navigateur devrait montrer quelque chose comme:

Phalcon\Tag fournit des méthodes utiles à la constructions des élément de formulaire.

La méthode `Phalcon\Tag::form()` ne reçoit qu’un seul paramètre par instance qui est une URI relative vers un contrôleur/action dans l’application.

En cliquant sur le bouton “Send” vous remarquerez une exception levée par le framework, indiquant que nous avons oublié l’action “register” dans le contrôleur “signup”. Notre fichier `public/index.php` lève cette exception:

Exception: Action “register” was not found on handler “signup”

Le fait de réaliser cette méthode supprimera cette exception:

```
<?php

use Phalcon\Mvc\Controller;

class SignupController extends Controller
{
    public function indexAction()
    {

    }
}
```


The screenshot shows a Google Chrome browser window with the address bar displaying `http://localhost/tutorial/signup`. The page content includes the heading "Sign up using this form", followed by two input fields labeled "Name" and "E-Mail", and a "Register" button below them.

```
public function registerAction()
{
    // ...
}

```

Si vous cliquez à nouveau sur le bouton “Send”, vous tomberez sur une page blanche. Le nom et l’email fournis en entrée par l’utilisateur devrait être stocké en base. Selon les conventions MVC, les interactions avec la base de données sont faites dans les modèles afin d’assurer un code orienté objet propre.

Création d’un modèle

Phalcon apporte le premier ORM pour PHP entièrement écrit en langage C. Au lieu d’augmenter la complexité du développement, il le simplifie.

Avant de créer notre premier modèle, nous avons besoin de créer une table dans une base de données de la rattacher. Un simple table pour stocker les utilisateurs inscrits peut être définie ainsi:

```
CREATE TABLE `users` (
  `id`      int(10)      unsigned NOT NULL AUTO_INCREMENT,
  `name`    varchar(70)  NOT NULL,
  `email`   varchar(70)  NOT NULL,

  PRIMARY KEY (`id`)
);

```

Le modèle doit être placé dans le répertoire `app/models` (`app/models/Users.php`). Le modèle est rattaché à la table “users”:

```
<?php

use Phalcon\Mvc\Model;

class Users extends Model
{
    public $id;

    public $name;

    public $email;
}
```

Définition de la connexion à la base de données

Afin de pouvoir utiliser une connexion à une base de donnée et d'accéder aux données grâce aux modèles, nous devons le spécifier dans notre processus d'amorçage. Une connexion à la base de données est juste un autre service de notre application qui peut être utilisé par de nombreux composants:

```
<?php

use Phalcon\Db\Adapter\Pdo\Mysql as DbAdapter;

// Définition du service de base de données
$di->set(
    "db",
    function () {
        return new DbAdapter(
            [
                "host"      => "localhost",
                "username" => "root",
                "password" => "secret",
                "dbname"   => "test_db",
            ]
        );
    }
);
```

Avec les bons paramètres de base, notre modèle est prêt à fonctionner et à interagir avec le reste de l'application.

Stockage de données avec les modèles

La prochaine étape est la réception des données provenant du formulaire et le stockage dans la table.

```
<?php

use Phalcon\Mvc\Controller;

class SignupController extends Controller
{
    public function indexAction()
    {
    }
}
```

```

public function registerAction()
{
    $user = new Users();

    // Stocke et vérifie les erreurs
    $success = $user->save(
        $this->request->getPost(),
        [
            "name",
            "email",
        ]
    );

    if ($success) {
        echo "Thanks for registering!";
    } else {
        echo "Sorry, the following problems were generated: ";

        $messages = $user->getMessages();

        foreach ($messages as $message) {
            echo $message->getMessage(), "<br/>";
        }
    }

    $this->view->disable();
}
}

```

Nous créons une instance de la classe Users qui correspond à un enregistrement User. Les propriétés publiques de la classe sont reliés aux champs de l'enregistrement dans la table users. Le fait de définir les valeurs appropriées dans le nouvel enregistrement et d'invoquer `save()` enregistrera les données dans la base pour cet enregistrement. La méthode `save()` retourne un booléen qui indique si le stockage est réussi ou non.

L'ORM échappe automatiquement les entrées pour prévenir des injections SQL, ainsi nous avons juste besoin de transmettre la requête à la méthode `save()`.

Une validation supplémentaire est réalisée automatiquement pour les champs qui sont définis comme non null (requis). Si nous ne renseignons aucun des champs requis dans le formulaire d'inscription notre écran devrait ressembler à ceci:

Conclusion

Ceci est un tutoriel très simple et, comme vous pouvez le voir, il est facile de commencer la construction d'une application avec Phalcon. Le fait que Phalcon soit une extension de votre serveur web n'a pas entravé la facilité de développement ou la disponibilité des fonctionnalités. Nous vous invitons à continuer de lire le manuel afin que vous puissiez découvrir d'autres fonctionnalités offertes par Phalcon !

2.2.2 Tutoriel 2: Présentation d'INVO

Dans ce second tutoriel, nous allons découvrir une application plus complète de manière à approfondir le développement de Phalcon. INVO est l'une des applications que nous avons créé en tant qu'exemple. INVO est un petit site web qui permet aux utilisateurs de générer des factures et faire d'autres tâches comme gérer les clients et ses produits. Vous pouvez cloner son code à partir de [Github](#).



INVO utilise aussi [Bootstrap](#) comme framework côté client. Même si l'application ne génère pas de factures, cela donne un exemple pour aider à comprendre comment le framework fonctionne.

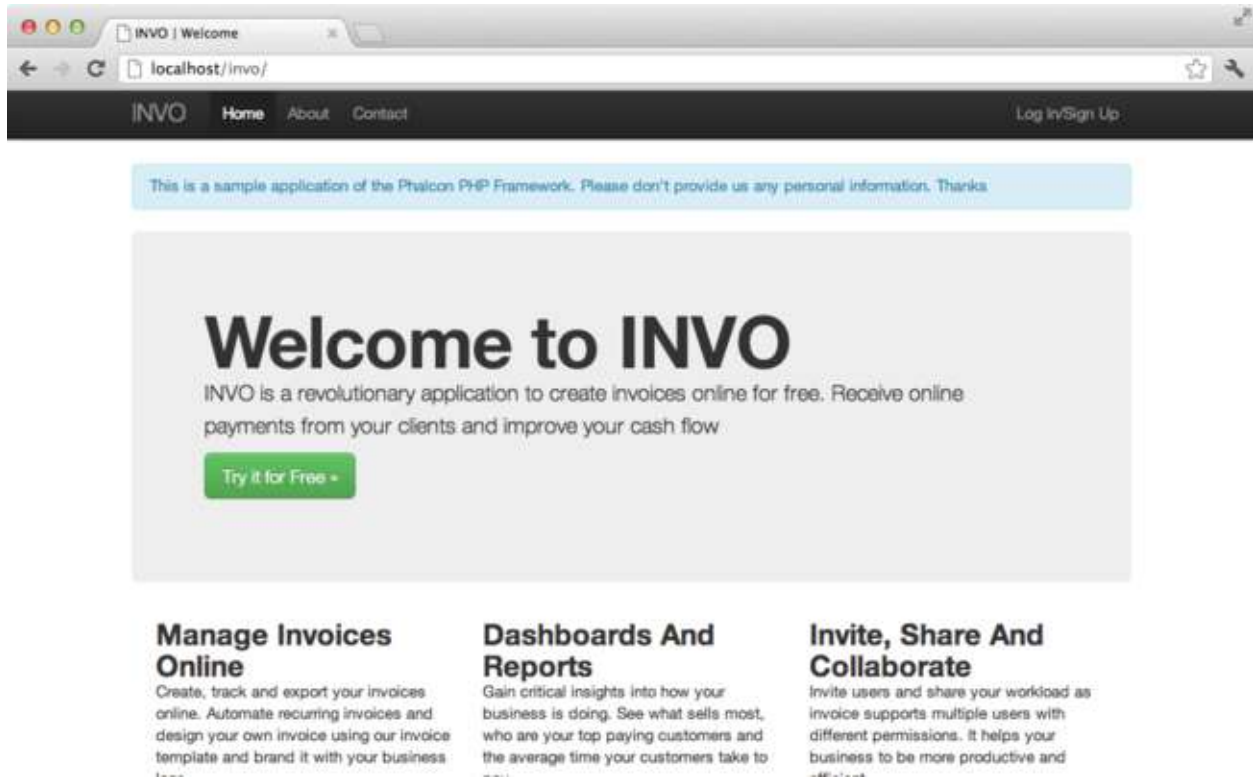
Structure du projet

Une fois que vous avez cloné le projet, à partir de la racine, vous verrez la structure suivante :

```
invo/  
  app/  
    config/  
    controllers/  
    forms/  
    library/  
    logs/  
    models/  
    plugins/  
    views/  
  cache/  
    volt/  
  docs/  
  public/  
    css/  
    fonts/  
    js/  
  schemas/
```

Comme vous l'avez vu précédemment, Phalcon n'impose pas de structure particulière pour développer une application. Ce projet fournit une simple structure MVC et un document racine public.

Une fois l'application ouverte dans votre navigateur à l'adresse : <http://localhost/invo> vous verrez quelque chose comme ceci :



Cette application est divisée en deux parties, un frontal, qui est une partie publique où les visiteurs peuvent obtenir des informations à propos d'INVO et des informations de contact. La seconde partie est le backend, une zone administrative où un utilisateur enregistré peut gérer ses produits et ses clients.

Routing

INVO utilise le routage standard qui est construit avec le composant Router. Ces routes correspondent au motif suivant : `/:controller/:action/:params`. Ceci signifie que la première partie de l'URI est le contrôleur, la seconde est l'action et ensuite viennent les paramètres.

La route suivante `/session/register` exécute le contrôleur `SessionController` et son action `registerAction`.

Configuration

INVO a un fichier de configuration qui définit les paramètres généraux de l'application. Ce fichier est lu par les premières lignes du fichier d'amorce (`public/index.php`) :

```
<?php
use Phalcon\Config\Adapter\Ini as ConfigIni;

// ...

// Lecture de la configuration
$config = new ConfigIni(
```

```
APP_PATH . "app/config/config.ini"
);
```

Phalcon\Config nous permet de manipuler le fichier comme un objet. Dans cet exemple, nous utilisons un fichier ini pour la configuration, cependant ils existe d’autres adaptateurs pour les fichiers de configuration. Le fichier de configuration contient les paramètres suivants :

```
[database]
host      = localhost
username  = root
password  = secret
name      = invo

[application]
controllersDir = app/controllers/
modelsDir      = app/models/
viewsDir       = app/views/
pluginsDir     = app/plugins/
formsDir       = app/forms/
libraryDir     = app/library/
baseUri        = /invo/
```

Phalcon n’a pas de convention de codage prédéfinie. Les sections nous permettent d’organiser les options de manière appropriée. Dans ce fichier il y a deux sections “application” et “database” que nous utiliserons plus tard.

Chargeurs automatiques

La seconde partie du fichier d’amorce (public/index.php) est le chargeur automatique:

```
<?php

/**
 * Configuration de chargement automatique
 */
require APP_PATH . "app/config/loader.php";
```

Le chargeur automatique consigne un ensemble de dossiers dans lesquels l’application cherchera les classes dont il aura éventuellement besoin.

```
<?php

$loader = new Phalcon\Loader();

// Nous consignons un ensemble de répertoire pris dans le fichier de configuration
$loader->registerDirs(
    [
        APP_PATH . $config->application->controllersDir,
        APP_PATH . $config->application->pluginsDir,
        APP_PATH . $config->application->libraryDir,
        APP_PATH . $config->application->modelsDir,
        APP_PATH . $config->application->formsDir,
    ]
);

$loader->register();
```

Notez que le code ci-dessous consigne des dossiers qui sont définis dans le fichier de configuration. Le seul dossier qui n'est pas enregistré est `viewsDir` parce qu'il ne contient pas de classes mais des fichiers de type HTML + PHP. Notez aussi que nous avons utilisé une constante nommée `APP_PATH`. Cette constante est définie dans l'amorce (`public/index.php`) ce qui nous permet de garder une référence sur la racine de notre projet:

```
<?php

// ...

define(
    "APP_PATH",
    realpath("../") . "/"
);
```

Inscription de services

Un autre fichier qui est requis dans l'amorce est (`app/config/services.php`). Ce fichier nous permet d'organiser les services que INVO doit utiliser.

```
<?php

/**
 * Chargement des services de l'application
 */
require APP_PATH . "app/config/services.php";
```

L'inscription de service est réalisée comme dans le tutoriel précédents, avec l'utilisation de closures pour un chargement paresseux des composants requis:

```
<?php

use Phalcon\Mvc\Url as UrlProvider;

// ...

/**
 * Le composant URL sert à générer toutes sortes d'URL dans l'application
 */
$di->set(
    "url",
    function () use ($config) {
        $url = new UrlProvider();

        $url->setBaseUri(
            $config->application->baseUri
        );

        return $url;
    }
);
```

Nous approfondirons l'étude de ce fichier plus tard.

Gestion de la requête

Si nous sautons à la fin du fichier (public/index.php), la requête est finalement gérée par *Phalcon\Mvc\Application*, qui initialise et exécute tout ce qui est nécessaire pour faire tourner l'application:

```
<?php

use Phalcon\Mvc\Application;

// ...

$application = new Application($di);

$response = $application->handle();

$response->send();
```

Injection de dépendances

Regardez à la première ligne du code ci-dessus, le constructeur de la classe *Application* reçoit la variable *\$di* en argument. Quel est le rôle de cette variable ? Phalcon est un framework fortement découplé, donc on a besoin d'un composant qui agit comme un ciment pour que tout fonctionne ensemble. Ce composant est *Phalcon\Di*. C'est un conteneur de services qui réalise aussi des injections de dépendance et la localisation de service, instanciant tous les composants nécessaires à l'application.

Il y a différents moyens d'inscrire des services dans le conteneur. Dans INVO la plupart des services sont enregistrés en utilisant des fonctions anonymes. Grâce à cela, les objets sont instanciés paresseusement (= uniquement lorsque nécessaire), ce qui réduit les ressources requises par l'application.

Par exemple, dans l'extrait suivant, le service de session est inscrit, la fonction anonyme sera appelée uniquement lorsque l'application aura besoin d'accéder aux données de la session:

```
<?php

use Phalcon\Session\Adapter\Files as Session;

// ...

// Démarre la session à la première demande au composant
$di->set(
    "session",
    function () {
        $session = new Session();

        $session->start();

        return $session;
    }
);
```

Ici, nous avons la possibilité de changer l'adaptateur, de faire des initialisations supplémentaires ainsi que beaucoup d'autres choses. Notez que le service est inscrit avec le nom "session", c'est une convention qui va permettre au framework d'identifier le service actif dans le conteneur de service.

Une requête peut utiliser plusieurs services et inscrire chaque service un par un peut être une tâche pénible. Pour cette raison le framework fournit une variante à *Phalcon\Di* appelée *Phalcon\DiFactoryDefault* qui a pour mission d'enregistrer tous les services, fournissant ainsi un framework complet.


```
<?php

use Phalcon\Di\FactoryDefault;

// ...

// The FactoryDefault Dependency Injector automatically registers the
// right services providing a full-stack framework
$di = new FactoryDefault();
```

Cet extrait inscrit la majorité des services avec les composants fournis par le framework. Si on a besoin de surcharger la définition de certains services on pourrait le redéfinir comme on l’a fait pour “session” ou “url”. C’est la raison d’être de la variable \$di.

Dans le chapitre suivant, nous verrons comment l’authentification et l’autorisations sont mis en œuvre dans INVO.

2.2.3 Tutorial 3: Securing INVO

In this chapter, we continue explaining how INVO is structured, we’ll talk about the implementation of authentication, authorization using events and plugins and an access control list (ACL) managed by Phalcon.

Se connecter à l’application

Se connecter va nous permettre de travailler sur les contrôleurs du backend. La séparation entre les contrôleurs du backend et du frontend sont purement d’ordre logique, car tous les contrôleurs sont localisés dans le même dossier (app/controllers/).

Pour se connecter il faut un nom d’utilisateur et un mot de passe valide. Les utilisateurs sont stockés dans la table “users” de la base de données “invo”.

Avant de pouvoir commencer une session, nous devons configurer la connexion à la base de données. Un service appelé “db” est utilisé dans le conteneur de service avec cette information. Pour ce qui est de l’autoloader, on prends en paramètres les informations du fichier de configuration de manière à configurer le service :

```
<?php

use Phalcon\Db\Adapter\Pdo\Mysql as DbAdapter;

// ...

// Database connection is created based on parameters defined in the configuration_
↪file
$di->set(
    "db",
    function () use ($config) {
        return new DbAdapter(
            [
                "host"      => $config->database->host,
                "username" => $config->database->username,
                "password" => $config->database->password,
                "dbname"   => $config->database->name,
            ]
        );
    }
);
```

Ici on retourne une instance de l'adaptateur de connexion à MySQL. Si nécessaire on pourrait faire des actions supplémentaire tel qu'ajouter un logger, un profileur ou changer l'adaptateur, ...

Le formulaire (app/views/session/index.volt) demande les informations de connexion. Certaines lignes HTML ont été supprimés dans l'extrait suivant pour rendre l'exemple plus concis:

```
{{ form("session/start") }}
<fieldset>
  <div>
    <label for="email">
      Username/Email
    </label>

    <div>
      {{ text_field("email") }}
    </div>
  </div>

  <div>
    <label for="password">
      Password
    </label>

    <div>
      {{ password_field("password") }}
    </div>
  </div>

  <div>
    {{ submit_button("Login") }}
  </div>
</fieldset>
{{ endForm() }}
```

Instead of using raw PHP as the previous tutorial, we started to use *Volt*. This is a built-in template engine inspired in *Jinja* providing a simpler and friendly syntax to create templates. It will not take too long before you become familiar with Volt.

Le `SessionController::startAction` (app/controllers/SessionController.php) a pour tâche de valider les données entrées à la recherche d'un utilisateur valide dans la base de données :

```
<?php

class SessionController extends ControllerBase
{
    // ...

    private function _registerSession($user)
    {
        $this->session->set(
            "auth",
            [
                "id" => $user->id,
                "name" => $user->name,
            ]
        );
    }
}
```

```

/**
 * This action authenticate and logs a user into the application
 */
public function startAction()
{
    if ($this->request->isPost()) {
        // Get the data from the user
        $email    = $this->request->getPost("email");
        $password = $this->request->getPost("password");

        // Find the user in the database
        $user = Users::findFirst(
            [
                "(email = :email: OR username = :email:) AND password = _
↪:password: AND active = 'Y'",
                "bind" => [
                    "email"    => $email,
                    "password" => sha1($password),
                ]
            ]
        );

        if ($user !== false) {
            $this->_registerSession($user);

            $this->flash->success(
                "Welcome " . $user->name
            );

            // Forward to the 'invoices' controller if the user is valid
            return $this->dispatcher->forward(
                [
                    "controller" => "invoices",
                    "action"      => "index",
                ]
            );
        }

        $this->flash->error(
            "Wrong email/password"
        );
    }

    // Forward to the login form again
    return $this->dispatcher->forward(
        [
            "controller" => "session",
            "action"      => "index",
        ]
    );
}
}

```

Pour des raisons de simplicité, nous avons utilisé “sha1” pour stocker le mot de passe hashé dans la base de données, cependant cet algorithme n’est pas recommandé pour une vraie application, il est préférable d’utiliser “bcrypt” à la place.

Veillez noter que plusieurs attributs public sont accessibles dans le contrôleur avec `$this->flash`, `$this->request` ou `$this->session`. Ceux-ci sont des services défini dans le conteneur de service de tout à l'heure (`app/config/services.php`). Quand ils sont accédés pour la première fois, ils sont insérés dans le contrôleur.

Ces services sont partagés, ce qui signifie qu'on accède à la même instance sans tenir compte de l'endroit où on les a créés.

Par exemple, ici on créé le service de sessions et on enregistre l'identité de utilisateur dans la variable "auth":

```
<?php

$this->session->set(
    "auth",
    [
        "id"    => $user->id,
        "name"  => $user->name,
    ]
);
```

Another important aspect of this section is how the user is validated as a valid one, first we validate whether the request has been made using method POST:

```
<?php

if ($this->request->isPost()) {
```

Then, we receive the parameters from the form:

```
<?php

$email    = $this->request->getPost("email");
$password = $this->request->getPost("password");
```

Now, we have to check if there is one user with the same username or email and password:

```
<?php

$user = Users::findFirst(
    [
        "(email = :email: OR username = :email:) AND password = :password: AND active_
↪ = 'Y'",
        "bind" => [
            "email"    => $email,
            "password" => sha1($password),
        ]
    ]
);
```

Note, the use of 'bound parameters', placeholders `:email:` and `:password:` are placed where values should be, then the values are 'bound' using the parameter 'bind'. This safely replaces the values for those columns without having the risk of a SQL injection.

If the user is valid we register it in session and forwards him/her to the dashboard:

```
<?php

if ($user !== false) {
    $this->_registerSession($user);
```

```

$this->flash->success(
    "Welcome " . $user->name
);

return $this->dispatcher->forward(
    [
        "controller" => "invoices",
        "action"      => "index",
    ]
);
}

```

If the user does not exist we forward the user back again to action where the form is displayed:

```

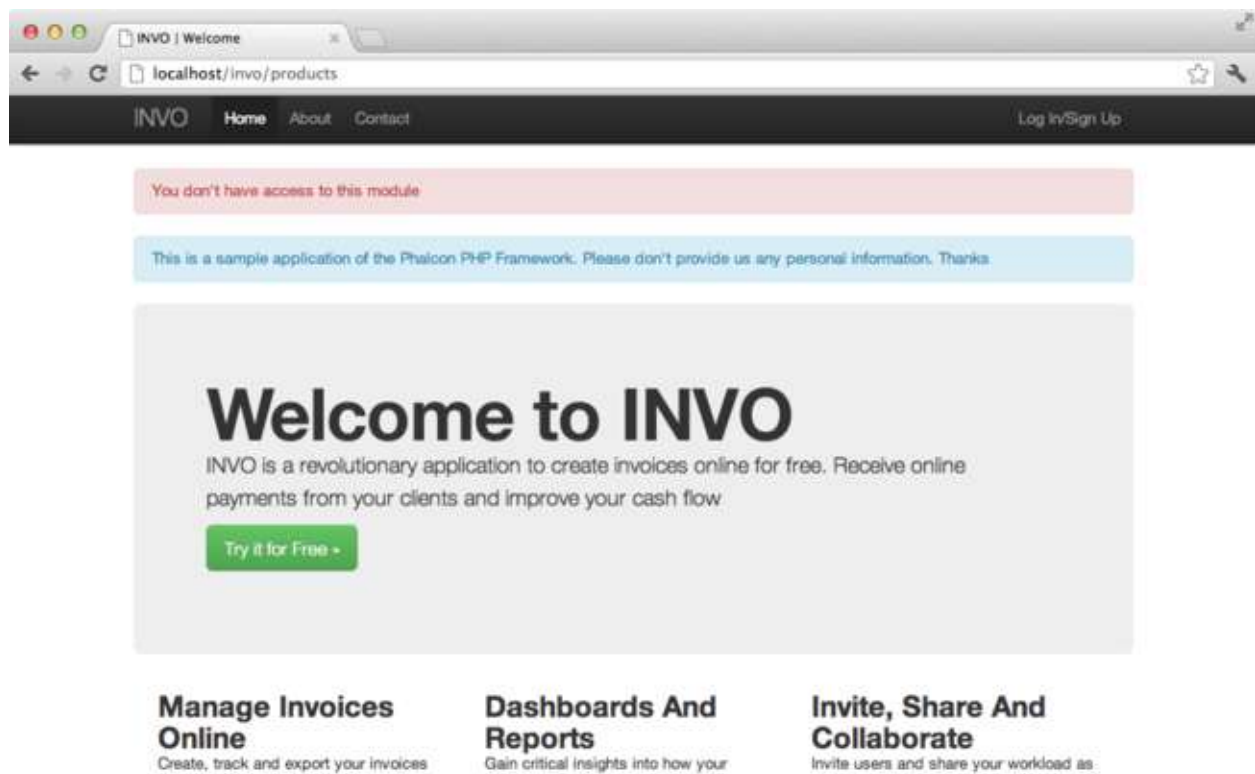
<?php

return $this->dispatcher->forward(
    [
        "controller" => "session",
        "action"      => "index",
    ]
);

```

Sécuriser le Backend

Le backend est une zone privé où seul les personnes enregistrés ont accès. Par conséquent il est nécessaire de vérifier que seul les utilisateurs enregistrés ont accès à ces contrôleurs. Si vous n'êtes pas connectés à l'application et que vous essayez d'accéder au contrôleur product, par exemple, vous verrez le message suivant :



A chaque fois que quelqu'un essaye d'accéder à n'importe quel contrôleur/action, l'application va vérifier que le rôle de l'utilisateur (en session) lui permet d'y accéder, sinon il affiche un message comme celui du dessus et transfère le flux à la page d'accueil.

Maintenant, découvrons comment l'application fait cela. La première chose à savoir est qu'il y a un composant appelé *Dispatcher*. Il est informé de la route trouvée par le composant *Routing*. Puis, il est responsable de charger le contrôleur approprié et d'exécuter l'action correspondante.

En temps normal, le framework crée le dispatcher automatiquement. Dans notre cas, nous voulons faire une vérification avant d'exécuter l'action requise, vérifier si l'utilisateur y a accès ou pas. Pour faire cela, nous avons remplacé le composant en créant une fonction dans le bootstrap (public/index.php):

```
<?php

use Phalcon\Mvc\Dispatcher;

// ...

/**
 * MVC dispatcher
 */
$di->set(
    "dispatcher",
    function () {
        // ...

        $dispatcher = new Dispatcher();

        return $dispatcher;
    }
);
```

Nous avons maintenant un contrôle complet sur le dispatcher utilisé dans notre application. Plusieurs composants du framework déclenchent des événements qui nous autorisent à modifier le flux interne des opérations. Comme l'injecteur de dépendances agit comme une “colle” pour composants, un nouveau composant appelé *EventManager* nous aide à intercepter les événements produits par un composant routant les événements aux listeners.

Gestion des événements

Un *EventManager* (gestionnaire d'évènement) nous permet d'attacher un ou plusieurs listeners à un type particulier d'évènement. Le type d'évènement qui nous intéresse actuellement est le “dispatch”, la code suivant filtre tous les événements produits par le dispatcher :

```
<?php

use Phalcon\Mvc\Dispatcher;
use Phalcon\Events\Manager as EventsManager;

$di->set(
    "dispatcher",
    function () {
        // Create an events manager
        $eventsManager = new EventsManager();

        // Listen for events produced in the dispatcher using the Security plugin
        $eventsManager->attach(
            "dispatch:beforeExecuteRoute",
```

```

        new SecurityPlugin()
    );

    // Handle exceptions and not-found exceptions using NotFoundPlugin
    $eventsManager->attach(
        "dispatch:beforeException",
        new NotFoundPlugin()
    );

    $dispatcher = new Dispatcher();

    // Assign the events manager to the dispatcher
    $dispatcher->setEventsManager($eventsManager);

    return $dispatcher;
}
);

```

When an event called “beforeExecuteRoute” is triggered the following plugin will be notified:

```

<?php

/**
 * Check if the user is allowed to access certain action using the SecurityPlugin
 */
$eventsManager->attach(
    "dispatch:beforeExecuteRoute",
    new SecurityPlugin()
);

```

When a “beforeException” is triggered then other plugin is notified:

```

<?php

/**
 * Handle exceptions and not-found exceptions using NotFoundPlugin
 */
$eventsManager->attach(
    "dispatch:beforeException",
    new NotFoundPlugin()
);

```

Le plugin de sécurité est une classe situé dans (app/plugins/SecurityPlugin.php). Cette classe implémente une méthode “beforeExecuteRoute”. C’est le même nom qu’un des évènement produit dans le dispatcher :

```

<?php

use Phalcon\Events\Event;
use Phalcon\Mvc\User\Plugin;
use Phalcon\Mvc\Dispatcher;

class SecurityPlugin extends Plugin
{
    // ...

    public function beforeExecuteRoute(Event $event, Dispatcher $dispatcher)
    {
        // ...
    }
}

```

```
}  
}
```

Les événements “hooks” reçoivent toujours un premier paramètre qui contient le contexte de l’information de l’évènement produit (`$event`) et un second paramètre qui est l’objet produit par l’évènement lui-même (`$dispatcher`). Il n’est pas obligatoire de faire étendre le plugin de la classe *Phalcon\Mvc\User\Plugin*, mais en faisant ainsi on a un accès facilité aux services disponibles de l’application.

Maintenant nous allons vérifier le rôle de la session courante, vérifier si l’utilisateur a accès en utilisant les listes ACL (access control list). S’il/elle n’a pas accès, il/elle sera redirigé(e) vers la page d’accueil comme expliqué précédemment.

```
<?php  
  
use Phalcon\Acl;  
use Phalcon\Events\Event;  
use Phalcon\Mvc\User\Plugin;  
use Phalcon\Mvc\Dispatcher;  
  
class SecurityPlugin extends Plugin  
{  
    // ...  
  
    public function beforeExecuteRoute(Event $event, Dispatcher $dispatcher)  
    {  
        // Check whether the "auth" variable exists in session to define the active_  
↪role  
        $auth = $this->session->get("auth");  
  
        if (!$auth) {  
            $role = "Guests";  
        } else {  
            $role = "Users";  
        }  
  
        // Take the active controller/action from the dispatcher  
        $controller = $dispatcher->getControllerName();  
        $action      = $dispatcher->getActionName();  
  
        // Obtain the ACL list  
        $acl = $this->getAcl();  
  
        // Check if the Role have access to the controller (resource)  
        $allowed = $acl->isAllowed($role, $controller, $action);  
  
        if (!$allowed) {  
            // If he doesn't have access forward him to the index controller  
            $this->flash->error(  
                "You don't have access to this module"  
            );  
  
            $dispatcher->forward(  
                [  
                    "controller" => "index",  
                    "action"      => "index",  
                ]  
            );  
        }  
    }  
}
```



```

        // Returning "false" we tell to the dispatcher to stop the current_
↪operation
        return false;
    }
}
}

```

Fournir une liste ACL

Dans l'exemple précédent, nous avons obtenu les ACL en utilisant la méthode `$this->getAcl()`. Cette méthode est aussi implémentée dans Plugin. Maintenant nous allons expliquer étape par étape comment nous avons construit les ACL (access control list) :

```

<?php

use Phalcon\Acl;
use Phalcon\Acl\Role;
use Phalcon\Acl\Adapter\Memory as AclList;

// Create the ACL
$acl = new AclList();

// The default action is DENY access
$acl->setDefaultAction(
    Acl::DENY
);

// Register two roles, Users is registered users
// and guests are users without a defined identity
$roles = [
    "users" => new Role("Users"),
    "guests" => new Role("Guests"),
];

foreach ($roles as $role) {
    $acl->addRole($role);
}

```

On définit les ressources pour chaque zone. Le nom des contrôleurs sont des ressources et leurs actions sont accédées pour les ressources :

```

<?php

use Phalcon\Acl\Resource;

// ...

// Private area resources (backend)
$privateResources = [
    "companies" => ["index", "search", "new", "edit", "save", "create", "delete"],
    "products"  => ["index", "search", "new", "edit", "save", "create", "delete"],
    "producttypes" => ["index", "search", "new", "edit", "save", "create", "delete"],
    "invoices"  => ["index", "profile"],
];

foreach ($privateResources as $resourceName => $actions) {

```

```
$acl->addResource(  
    new Resource($resourceName),  
    $actions  
);  
}  
  
// Public area resources (frontend)  
$publicResources = [  
    "index"    => ["index"],  
    "about"    => ["index"],  
    "register" => ["index"],  
    "errors"   => ["show404", "show500"],  
    "session"  => ["index", "register", "start", "end"],  
    "contact"  => ["index", "send"],  
];  
  
foreach ($publicResources as $resourceName => $actions) {  
    $acl->addResource(  
        new Resource($resourceName),  
        $actions  
    );  
}
```

Les ACL ont maintenant connaissance des contrôleurs et de leurs actions. Le rôle “Users” a accès à toutes les ressources du backend et du frontend. Le rôle “Guest” en revanche n’a accès qu’à la partie publique :

```
<?php  
  
// Grant access to public areas to both users and guests  
foreach ($roles as $role) {  
    foreach ($publicResources as $resource => $actions) {  
        $acl->allow(  
            $role->getName(),  
            $resource,  
            "*"   
        );  
    }  
}  
  
// Grant access to private area only to role Users  
foreach ($privateResources as $resource => $actions) {  
    foreach ($actions as $action) {  
        $acl->allow(  
            "Users",  
            $resource,  
            $action  
        );  
    }  
}
```

Hooray!, les ACL sont maintenant terminés. In next chapter, we will see how a CRUD is implemented in Phalcon and how you can customize it.

2.2.4 Tutorial 4: Travailler avec le CRUD

Backends usually provide forms to allow users to manipulate data. Continuing the explanation of INVO, we now address the creation of CRUDs, a very common task that Phalcon will facilitate you using forms, validations, paginators and more.

La plupart des options qui manipulent des données (companies, products et types de products), ont été développés en utilisant un **CRUD** (create/read/update/delete) basique et commun. Chaque CRUD contient les fichiers suivants :

```

invo/
  app/
    controllers/
      ProductsController.php
    models/
      Products.php
    forms/
      ProductsForm.php
    views/
      products/
        edit.volt
        index.volt
        new.volt
        search.volt

```

Chaque contrôleur a les actions suivantes :

```

<?php

class ProductsController extends ControllerBase
{
    /**
     * The start action, it shows the "search" view
     */
    public function indexAction()
    {
        // ...
    }

    /**
     * Execute the "search" based on the criteria sent from the "index"
     * Returning a paginator for the results
     */
    public function searchAction()
    {
        // ...
    }

    /**
     * Shows the view to create a "new" product
     */
    public function newAction()
    {
        // ...
    }

    /**
     * Shows the view to "edit" an existing product
     */
    public function editAction()

```

```
{
    // ...
}

/**
 * Creates a product based on the data entered in the "new" action
 */
public function createAction()
{
    // ...
}

/**
 * Updates a product based on the data entered in the "edit" action
 */
public function saveAction()
{
    // ...
}

/**
 * Deletes an existing product
 */
public function deleteAction($id)
{
    // ...
}
}
```

Formulaire de recherche

Tous les CRUD commencent avec le formulaire de recherche. Ce formulaire montre tous les champs que la table `products` possède, permettant à l'utilisateur de filtrer ses recherches. La tâche `products` est liée à la table `products_types`. Dans notre cas, nous avons déjà demandé des enregistrements de cette table, afin de faciliter la recherche dans ce champ :

```
<?php

/**
 * The start action, it shows the "search" view
 */
public function indexAction()
{
    $this->persistent->searchParams = null;

    $this->view->form = new ProductsForm();
}
```

An instance of the `ProductsForm` form (`app/forms/ProductsForm.php`) is passed to the view. This form defines the fields that are visible to the user:

```
<?php

use Phalcon\Forms\Form;
use Phalcon\Forms\Element\Text;
use Phalcon\Forms\Element\Hidden;
```

```

use Phalcon\Forms\Element\Select;
use Phalcon\Validation\Validator\Email;
use Phalcon\Validation\Validator\PresenceOf;
use Phalcon\Validation\Validator\Numericality;

class ProductsForm extends Form
{
    /**
     * Initialize the products form
     */
    public function initialize($entity = null, $options = [])
    {
        if (!isset($options["edit"])) {
            $element = new Text("id");

            $element->setLabel("Id");

            $this->add(
                $element
            );
        } else {
            $this->add(
                new Hidden("id")
            );
        }

        $name = new Text("name");

        $name->setLabel("Name");

        $name->setFilters(
            [
                "striptags",
                "string",
            ]
        );

        $name->addValidators(
            [
                new PresenceOf(
                    [
                        "message" => "Name is required",
                    ]
                )
            ]
        );

        $this->add($name);

        $type = new Select(
            "profilesId",
            ProductTypes::find(),
            [
                "using" => [

```

```
        "id",
        "name",
    ],
    "useEmpty"    => true,
    "emptyText"   => "...",
    "emptyValue"  => "",
]
);

$this->add($type);

$price = new Text("price");
$price->setLabel("Price");
$price->setFilters([
    "float",
]);

$price->addValidators([
    new PresenceOf([
        "message" => "Price is required",
    ]),
    new Numericality([
        "message" => "Price is required",
    ]),
]);

$this->add($price);
}
```

The form is declared using an object-oriented scheme based on the elements provided by the *forms* component. Every element follows almost the same structure:

```
<?php

// Create the element
$name = new Text("name");

// Set its label
$name->setLabel("Name");

// Before validating the element apply these filters
$name->setFilters([
    "striptags",
    "string",
```

```

    ]
);

// Apply this validators
$name->addValidators(
    [
        new PresenceOf(
            [
                "message" => "Name is required",
            ]
        )
    ]
);

// Add the element to the form
$this->add($name);

```

Other elements are also used in this form:

```

<?php

// Add a hidden input to the form
$this->add(
    new Hidden("id")
);

// ...

$productTypes = ProductTypes::find();

// Add a HTML Select (list) to the form
// and fill it with data from "product_types"
$type = new Select(
    "profilesId",
    $productTypes,
    [
        "using" => [
            "id",
            "name",
        ],
        "useEmpty" => true,
        "emptyText" => "...",
        "emptyValue" => "",
    ]
);

```

Note that `ProductTypes::find()` contains the data necessary to fill the `SELECT` tag using `Phalcon\Tag::select()`. Once the form is passed to the view, it can be rendered and presented to the user:

```

{{ form("products/search") }}

<h2>
    Search products
</h2>

<fieldset>

```

```
{% for element in form %}
    <div class="control-group">
        {{ element.label(["class": "control-label"]) }}

        <div class="controls">
            {{ element }}
        </div>
    </div>
{% endfor %}

<div class="control-group">
    {{ submit_button("Search", "class": "btn btn-primary") }}
</div>

</fieldset>

{{ endForm() }}
```

This produces the following HTML:

```
<form action="/invo/products/search" method="post">

    <h2>
        Search products
    </h2>

    <fieldset>

        <div class="control-group">
            <label for="id" class="control-label">Id</label>

            <div class="controls">
                <input type="text" id="id" name="id" />
            </div>
        </div>

        <div class="control-group">
            <label for="name" class="control-label">Name</label>

            <div class="controls">
                <input type="text" id="name" name="name" />
            </div>
        </div>

        <div class="control-group">
            <label for="profilesId" class="control-label">profilesId</label>

            <div class="controls">
                <select id="profilesId" name="profilesId">
                    <option value="">...</option>
                    <option value="1">Vegetables</option>
                    <option value="2">Fruits</option>
                </select>
            </div>
        </div>

    </div>

</form>
```



```

        <div class="control-group">
            <label for="price" class="control-label">Price</label>

            <div class="controls">
                <input type="text" id="price" name="price" />
            </div>
        </div>

        <div class="control-group">
            <input type="submit" value="Search" class="btn btn-primary" />
        </div>

    </fieldset>

</form>

```

When the form is submitted, the “search” action is executed in the controller performing the search based on the data entered by the user.

Exécuter une recherche

L'action de recherche a un double comportement. Quand on y accède avec POST, cela fait une recherche basé sur les données que l'on a envoyé à partir du formulaire. Mais quand on y accède via GET cela change la page courante dans le paginateur. Pour différencier la méthode (GET ou POST), nous utilisons le composant *Request* :

```

<?php

/**
 * Execute the "search" based on the criteria sent from the "index"
 * Returning a paginator for the results
 */
public function searchAction()
{
    if ($this->request->isPost()) {
        // Create the query conditions
    } else {
        // Paginate using the existing conditions
    }

    // ...
}

```

Avec l'aide de *Phalcon\Mvc\Model\Criteria*, nous pouvons créer les conditions de recherche basé sur les types de données envoyé via le formulaire :

```

<?php

$query = Criteria::fromInput(
    $this->di,
    "Products",
    $this->request->getPost()
);

```

Cette méthode vérifie quelle valeur est différente de “” (chaine vide) et “null” et les prends en compte pour créer les critères de recherche :

- Si le champs de données est “text” ou similaire (char, varchar, text, etc.). L’opérateur “like” sera utilisé pour filtrer les résultats.
- Si le type de donnée est différent, l’opérateur “=” sera utilisé.

De plus, “Criteria” ignore toutes les variables \$_POST qui ne correspondent à aucun champs de la table. Les valeurs seront automatiquement échappées en utilisant les paramètres liés (bond parameters).

Maintenant, on va stocker les paramètres dans le “sac” de session du contrôleur :

```
<?php
$this->persistent->searchParams = $query->getParams();
```

Un sac de session est un attribut particulier dans un contrôleur qui est sauvegardé entre les requêtes. Quand on y accède, cet attribut injecte un service *Phalcon\Session\Bag* qui est indépendant de chaque contrôleur.

Puis, basé sur les paramètres passé, on génère la requête :

```
<?php

$products = Products::find($parameters);

if (count($products) === 0) {
    $this->flash->notice(
        "The search did not found any products"
    );

    return $this->dispatcher->forward([
        "controller" => "products",
        "action"      => "index",
    ]);
}
```

Si la recherche ne retourne aucun produit, on transfère l’utilisateur à l’action index. Si la recherche retourne des résultats, on crée un paginateur pour se déplacer à travers les pages facilement :

```
<?php

use Phalcon\Paginator\Adapter\Model as Paginator;

// ...

$paginator = new Paginator([
    "data" => $products, // Data to paginate
    "limit" => 5,         // Rows per page
    "page" => $numberPage, // Active page
]);

// Get active page in the paginator
$page = $paginator->getPaginate();
```

Enfin, on passe la page retournée à la vue:

```
<?php
$this->view->page = $page;
```

Dans la vue (app/views/products/search.volt), on affiche le résultat correspondant à la page actuelle :

```
{% for product in page.items %}
    {% if loop.first %}
        <table>
            <thead>
                <tr>
                    <th>Id</th>
                    <th>Product Type</th>
                    <th>Name</th>
                    <th>Price</th>
                    <th>Active</th>
                </tr>
            </thead>
            <tbody>
{% endif %}

        <tr>
            <td>
                {{ product.id }}
            </td>

            <td>
                {{ product.getProductTypes().name }}
            </td>

            <td>
                {{ product.name }}
            </td>

            <td>
                {{ "%.2f"|format(product.price) }}
            </td>

            <td>
                {{ product.getActiveDetail() }}
            </td>

            <td width="7%">
                {{ link_to("products/edit/" ~ product.id, "Edit") }}
            </td>

            <td width="7%">
                {{ link_to("products/delete/" ~ product.id, "Delete") }}
            </td>
        </tr>

    {% if loop.last %}
        </tbody>
        <tbody>
            <tr>
                <td colspan="7">
                    <div>
                        {{ link_to("products/search", "First") }}
```

```
                                {{ link_to("products/search?page=" ~ page.before,
↪ "Previous") }}

                                {{ link_to("products/search?page=" ~ page.next, "Next") }}
                                {{ link_to("products/search?page=" ~ page.last, "Last") }}
                                <span class="help-inline">{{ page.current }} of {{ page.
↪ total_pages }}</span>
                                </div>
                                </td>
                                </tr>
                                </tbody>
                                </table>
                                {% endif %}
{% else %}
    No products are recorded
{% endfor %}
```

There are many things in the above example that worth detailing. First of all, active items in the current page are traversed using a Volt's 'for'. Volt provides a simpler syntax for a PHP 'foreach'.

```
{% for product in page.items %}
```

Which in PHP is the same as:

```
<?php foreach ($page->items as $product) { ?>
```

The whole 'for' block provides the following:

```
{% for product in page.items %}
    {% if loop.first %}
        Executed before the first product in the loop
    {% endif %}

    Executed for every product of page.items

    {% if loop.last %}
        Executed after the last product is loop
    {% endif %}
{% else %}
    Executed if page.items does not have any products
{% endfor %}
```

Now you can go back to the view and find out what every block is doing. Every field in "product" is printed accordingly:

```
<tr>
    <td>
        {{ product.id }}
    </td>

    <td>
        {{ product.productTypes.name }}
    </td>

    <td>
        {{ product.name }}
    </td>

    <td>
```

```

        {{ "%.2f"|format(product.price) }}
    </td>

    <td>
        {{ product.getActiveDetail() }}
    </td>

    <td width="7%">
        {{ link_to("products/edit/" ~ product.id, "Edit") }}
    </td>

    <td width="7%">
        {{ link_to("products/delete/" ~ product.id, "Delete") }}
    </td>
</tr>

```

As we seen before using `product.id` is the same as in PHP as doing: `$product->id`, we made the same with `product.name` and so on. Other fields are rendered differently, for instance, let's focus in `product.productTypes.name`. To understand this part, we have to check the Products model (`app/models/Products.php`):

```

<?php

use Phalcon\Mvc\Model;

/**
 * Products
 */
class Products extends Model
{
    // ...

    /**
     * Products initializer
     */
    public function initialize()
    {
        $this->belongsTo(
            "product_types_id",
            "ProductTypes",
            "id",
            [
                "reusable" => true,
            ]
        );
    }

    // ...
}

```

A model can have a method called `initialize()`, this method is called once per request and it serves the ORM to initialize a model. In this case, “Products” is initialized by defining that this model has a one-to-many relationship to another model called “ProductTypes”.

```

<?php

$this->belongsTo(
    "product_types_id",
    "ProductTypes",

```

```
"id",
[
    "reusable" => true,
]
);
```

Which means, the local attribute “product_types_id” in “Products” has an one-to-many relation to the “ProductTypes” model in its attribute “id”. By defining this relationship we can access the name of the product type by using:

```
<td>{{ product.productTypes.name }}</td>
```

The field “price” is printed by its formatted using a Volt filter:

```
<td>{{ "%.2f"|format(product.price) }}</td>
```

In plain PHP, this would be:

```
<?php echo sprintf("%.2f", $product->price) ?>
```

Printing whether the product is active or not uses a helper implemented in the model:

```
<td>{{ product.getActiveDetail() }}</td>
```

This method is defined in the model.

Créer et modifier des entrées

Voyons comment le CRUD crée et modifie des entrées. A partir des vues “new” et “edit”, la donnée entrée par l'utilisateur est envoyé à l'action “create” et “save” qui exécute l'action de créer ou de modifier les produits.

Dans la page de création, on récupère les données envoyés et on leur assigne une nouvelle instance de produit :

```
<?php

/**
 * Creates a product based on the data entered in the "new" action
 */
public function createAction()
{
    if (!$this->request->isPost()) {
        return $this->dispatcher->forward([
            "controller" => "products",
            "action"      => "index",
        ]);
    }

    $form = new ProductsForm();

    $product = new Products();

    $product->id           = $this->request->getPost("id", "int");
    $product->product_types_id = $this->request->getPost("product_types_id", "int");
    $product->name          = $this->request->getPost("name", "striptags");
    $product->price          = $this->request->getPost("price", "double");
    $product->active          = $this->request->getPost("active");
```

```
// ...
}
```

Les données sont filtrées avant d'être assignées à l'objet `$product`. Ce filtrage est optionnel, l'ORM échappe les données entrées et caste les données en fonction des types des champs:

```
<?php
// ...

$name = new Text("name");

$name->setLabel("Name");

// Filters for name
$name->setFilters(
    [
        "striptags",
        "string",
    ]
);

// Validators for name
$name->addValidators(
    [
        new PresenceOf(
            [
                "message" => "Name is required",
            ]
        )
    ]
);

$this->add($name);
```

Quand on sauvegarde, nous saurons si la donnée est conforme aux règles et validations implémentés dans le form `ProductsForm` (`app/forms/ProductsForm.php`):

```
<?php
// ...

$form = new ProductsForm();

$product = new Products();

// Validate the input
$data = $this->request->getPost();

if (!$form->isValid($data, $product)) {
    $messages = $form->getMessages();

    foreach ($messages as $message) {
        $this->flash->error($message);
    }

    return $this->dispatcher->forward(
```

```
        [
            "controller" => "products",
            "action"      => "new",
        ]
    );
}
```

Finally, if the form does not return any validation message we can save the product instance:

```
<?php
// ...

if ($product->save() === false) {
    $messages = $product->getMessages();

    foreach ($messages as $message) {
        $this->flash->error($message);
    }

    return $this->dispatcher->forward(
        [
            "controller" => "products",
            "action"      => "new",
        ]
    );
}

$form->clear();

$this->flash->success(
    "Product was created successfully"
);

return $this->dispatcher->forward(
    [
        "controller" => "products",
        "action"      => "index",
    ]
);
```

Maintenant, dans le cas de la modification de produit, on doit présenter les données à éditer à l'utilisateur en pré-remplissant les champs:

```
<?php

/**
 * Edits a product based on its id
 */
public function editAction($id)
{
    if (!$this->request->isPost()) {
        $product = Products::findFirstById($id);

        if (!$product) {
            $this->flash->error(
                "Product was not found"
            );
        }
    }
}
```



```

        return $this->dispatcher->forward(
            [
                "controller" => "products",
                "action"      => "index",
            ]
        );
    }

    $this->view->form = new ProductsForm(
        $product,
        [
            "edit" => true,
        ]
    );
}
}

```

L’helper “setDefault” entre les valeurs du produit dans les champs qui portent le même nom comme valeur par défaut. Grâce à cela, l’utilisateur peut changer n’importe quelle valeur et ensuite envoyer ses modifications à la base de données avec l’action “save”:

```

<?php

/**
 * Updates a product based on the data entered in the "edit" action
 */
public function saveAction()
{
    if (!$this->request->isPost()) {
        return $this->dispatcher->forward(
            [
                "controller" => "products",
                "action"      => "index",
            ]
        );
    }

    $id = $this->request->getPost("id", "int");

    $product = Products::findFirstById($id);

    if (!$product) {
        $this->flash->error(
            "Product does not exist"
        );

        return $this->dispatcher->forward(
            [
                "controller" => "products",
                "action"      => "index",
            ]
        );
    }

    $form = new ProductsForm();

    $data = $this->request->getPost();
}

```

```
if (!$form->isValid($data, $product)) {
    $messages = $form->getMessages();

    foreach ($messages as $message) {
        $this->flash->error($message);
    }

    return $this->dispatcher->forward(
        [
            "controller" => "products",
            "action"      => "new",
        ]
    );
}

if ($product->save() === false) {
    $messages = $product->getMessages();

    foreach ($messages as $message) {
        $this->flash->error($message);
    }

    return $this->dispatcher->forward(
        [
            "controller" => "products",
            "action"      => "new",
        ]
    );
}

$form->clear();

$this->flash->success(
    "Product was updated successfully"
);

return $this->dispatcher->forward(
    [
        "controller" => "products",
        "action"      => "index",
    ]
);
}
```

We have seen how Phalcon lets you create forms and bind data from a database in a structured way. In next chapter, we will see how to add custom HTML elements like a menu.

2.2.5 Tutorial 5: Customizing INVO

To finish the detailed explanation of INVO we are going to explain how to customize INVO adding UI elements and changing the title according to the controller executed.

Composants utilisateurs

Tous les éléments graphique et visuels de l'application ont été réalisés principalement avec [Bootstrap](#). Certains éléments, comme la barre de navigation, changent en fonction de l'état de l'application (connecté/déconnecté). Par exemple dans le coin en haut à droite, les liens "Log in / Sign up" (se connecter/s'inscrire) se changent en "Log out" (Se déconnecter) quand un utilisateur se connecte.

Cette partie de l'application est implémentée en utilisant le composant "Elements" (app/library/Elements.php).

```
<?php

use Phalcon\Mvc\User\Component;

class Elements extends Component
{
    public function getMenu()
    {
        // ...
    }

    public function getTabs()
    {
        // ...
    }
}
```

Cette classe étend de *Phalcon\Mvc\User\Component*, il n'est pas imposé d'étendre un composant avec cette classe, mais cela permet d'accéder plus rapidement/facilement aux services de l'application. Maintenant enregistrons cette classe au conteneur de service :

```
<?php

// Register a user component
$di->set(
    "elements",
    function () {
        return new Elements();
    }
);
```

Tout comme les contrôleurs, les plugins et les composants à l'intérieur des vues, ce composant à aussi accès aux services requis dans le conteneur en accédant juste à l'attribut :

```
<div class="navbar navbar-fixed-top">
    <div class="navbar-inner">
        <div class="container">
            <a class="btn btn-navbar" data-toggle="collapse" data-target=".nav-
→collapse">
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </a>

            <a class="brand" href="#">INVO</a>

            {{ elements.getMenu() }}
        </div>
    </div>
```

```
</div>

<div class="container">
    {{ content() }}

    <hr>

    <footer>
        <p>&copy; Company 2015</p>
    </footer>
</div>
```

La partie la plus importante est :

```
{{ elements.getMenu() }}
```

Changer le titre de manière dynamique

Quand vous naviguez sur le site, vous remarquerez que le titre change d'une page à l'autre. Cela est réalisé dans l'"initializer" de chaque contrôleur :

```
<?php

class ProductsController extends ControllerBase
{
    public function initialize()
    {
        // Set the document title
        $this->tag->setTitle(
            "Manage your product types"
        );

        parent::initialize();
    }

    // ...
}
```

Notez que la méthode `parent::initialize()` est aussi appelée, cela ajoute plus de donnée à la suite du titre :

```
<?php

use Phalcon\Mvc\Controller;

class ControllerBase extends Controller
{
    protected function initialize()
    {
        // Prepend the application name to the title
        $this->tag->prependTitle(
            "INVO | "
        );
    }

    // ...
}
```

Enfin, le titre est affiché dans la vue principale (app/views/index.volt) :

```
<!DOCTYPE html>
<html>
  <head>
    <?php echo $this->tag->getTitle(); ?>
  </head>

  <!-- ... -->
</html>
```

2.2.6 Tutorial 6: Vökuró

Vökuró is another sample application you can use to learn more about Phalcon. Vökuró is a small website that shows how to implement a security features and management of users and permissions. You can clone its code from [Github](#).

Project Structure

Once you clone the project in your document root you'll see the following structure:

```
vokuro/
  app/
    config/
    controllers/
    forms/
    library/
    models/
    views/
  cache/
  public/
    css/
    img/
  schemas/
```

This project follows a quite similar structure to INVO. Once you open the application in your browser <http://localhost/vokuro> you'll see something like this:

The application is divided into two parts, a frontend, where visitors can sign up the service and a backend where administrative users can manage registered users. Both frontend and backend are combined in a single module.

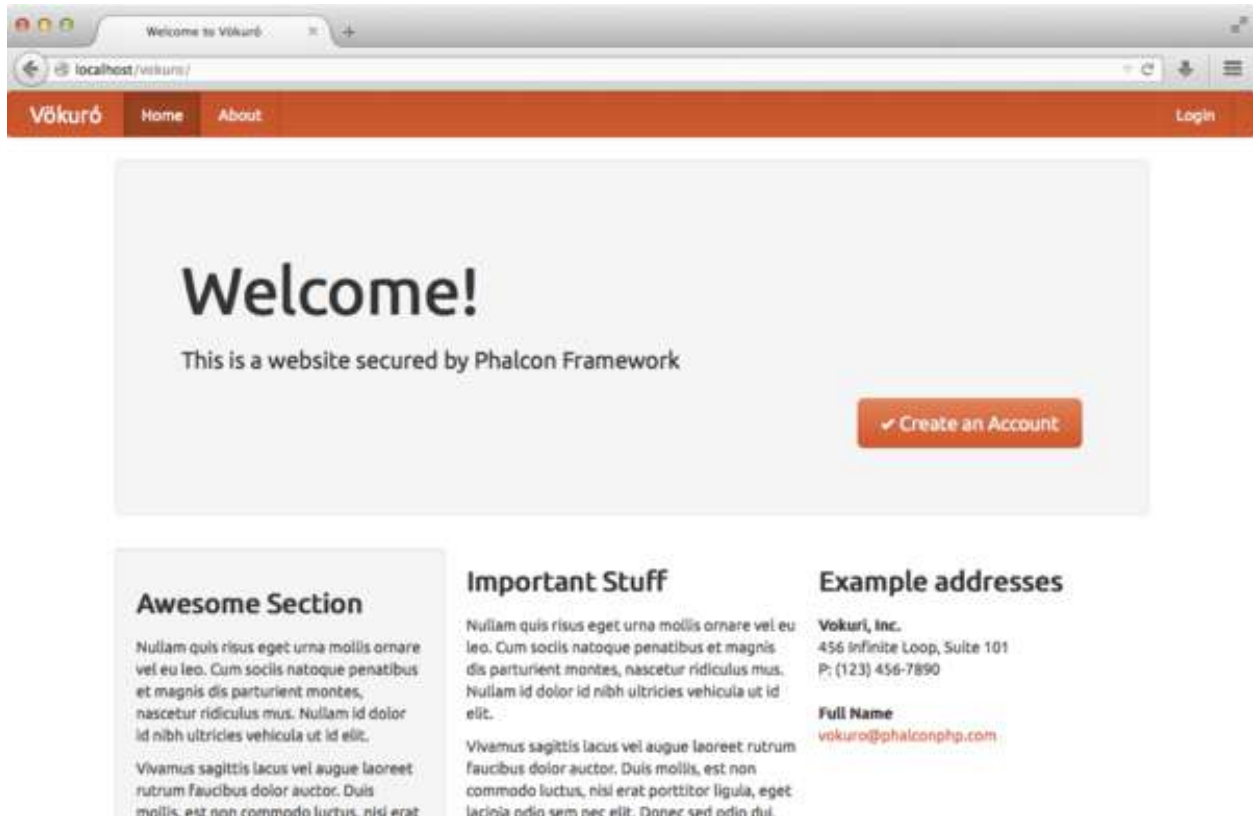
Load Classes and Dependencies

This project uses *Phalcon\Loader* to load controllers, models, forms, etc. within the project and *composer* to load the project's dependencies. So, the first thing you have to do before execute Vökuró is install its dependencies via *composer*. Assuming you have it correctly installed, type the following command in the console:

```
cd vokuro
composer install
```

Vökuró sends emails to confirm the sign up of registered users using Swift, the *composer.json* looks like:

```
{
  "require" : {
    "php" : ">=5.5.0",
```



```

"ext-phalcon" : ">=3.0.0",
"swiftmailer/swiftmailer" : "^5.4",
"amazonwebservices/aws-sdk-for-php" : "~1.0"
}
}

```

Now, there is a file called `app/config/loader.php` where all the auto-loading stuff is set up. At the end of this file you can see that the composer autoloader is included enabling the application to autoload any of the classes in the downloaded dependencies:

```

<?php

// ...

// Use composer autoloader to load vendor classes
require_once BASE_PATH . "/vendor/autoload.php";

```

Moreover, Vökuró, unlike the INVO, utilizes namespaces for controllers and models which is the recommended practice to structure a project. This way the autoloader looks slightly different than the one we saw before (`app/config/loader.php`):

```

<?php

use Phalcon\Loader;

$loader = new Loader();

$loader->registerNamespaces(

```

```
[
    "Vokuro\\Models"      => $config->application->modelsDir,
    "Vokuro\\Controllers" => $config->application->controllersDir,
    "Vokuro\\Forms"       => $config->application->formsDir,
    "Vokuro"              => $config->application->libraryDir,
]
);

$loader->register();

// ...
```

Instead of using `registerDirectories()`, we use `registerNamespaces()`. Every namespace points to a directory defined in the configuration file (`app/config/config.php`). For instance the namespace `Vokuro\Controllers` points to `app/controllers` so all the classes required by the application within this namespace requires it in its definition:

```
<?php

namespace Vokuro\Controllers;

class AboutController extends ControllerBase
{
    // ...
}
```

Sign Up

First, let's check how users are registered in Vökuró. When a user clicks the “Create an Account” button, the controller `SessionController` is invoked and the action “signup” is executed:

```
<?php

namespace Vokuro\Controllers;

use Vokuro\Forms\SignUpForm;

class RegisterController extends ControllerBase
{
    public function signupAction()
    {
        $form = new SignUpForm();

        // ...

        $this->view->form = $form;
    }
}
```

This action simply pass a form instance of `SignUpForm` to the view, which itself is rendered to allow the user enter the login details:

```
{{ form("class": "form-search") }}

<h2>
    Sign Up
</h2>
```

```
<p>{{ form.label("name") }}</p>
<p>
    {{ form.render("name") }}
    {{ form.messages("name") }}
</p>

<p>{{ form.label("email") }}</p>
<p>
    {{ form.render("email") }}
    {{ form.messages("email") }}
</p>

<p>{{ form.label("password") }}</p>
<p>
    {{ form.render("password") }}
    {{ form.messages("password") }}
</p>

<p>{{ form.label("confirmPassword") }}</p>
<p>
    {{ form.render("confirmPassword") }}
    {{ form.messages("confirmPassword") }}
</p>

<p>
    {{ form.render("terms") }} {{ form.label("terms") }}
    {{ form.messages("terms") }}
</p>

<p>{{ form.render("Sign Up") }}</p>

{{ form.render("csrf", ["value": security.getToken()]) }}
{{ form.messages("csrf") }}

<hr>

{{ endForm() }}
```

2.2.7 Tutorial 7: Créer une application REST API

Dans ce tutoriel, nous allons montrer comment créer une simple application qui fournit une API **RESTful** en utilisant les méthodes HTTP suivantes :

- GET pour récupérer et chercher les données
- POST pour ajouter des données
- PUT pour modifier les données
- DELETE pour supprimer les données

Définir l'API

L'API comprends les méthodes suivantes :

Method	URL	Action
GET	/api/robots	Récupérer tous les robots
GET	/api/robots/search/Astro	Cherche les robots ayant 'Astro' dans leur nom
GET	/api/robots/2	Récupérer les robots à partir de leur clé primaire
POST	/api/robots	Ajouter un nouveau robot
PUT	/api/robots/2	Modifier les robots à partir de leur clé primaire
DELETE	/api/robots/2	Supprimer les robots à partir de leur clé primaire

Créer l'application

Comme l'application est relativement simple, nous n'allons pas implémenter un environnement MVC complet. Dans notre cas nous allons utiliser une *micro application*.

La structure de fichier suivante sera largement suffisante :

```
my-rest-api/
  models/
    Robots.php
  index.php
  .htaccess
```

Tout d'abord nous avons besoin d'un .htaccess qui va contenir toutes les règles de réécriture d'URL pour notre fichier index.php :

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^((?s).*)$ index.php?url=/$1 [QSA,L]
</IfModule>
```

Ensuite dans notre fichier index.php, on ajoute ceci :

```
<?php

use Phalcon\Mvc\Micro;

$app = new Micro();

// Define the routes here

$app->handle();
```

Maintenant, nous allons créer les routes comme défini au dessus (le tableau) :

```
<?php

use Phalcon\Mvc\Micro;

$app = new Micro();

// Retrieves all robots
$app->get(
    "/api/robots",
    function () {

    }
);
```

```
// Searches for robots with $name in their name
$app->get(
    "/api/robots/search/{name}",
    function ($name) {

    }
);

// Retrieves robots based on primary key
$app->get(
    "/api/robots/{id:[0-9]+}",
    function ($id) {

    }
);

// Adds a new robot
$app->post(
    "/api/robots",
    function () {

    }
);

// Updates robots based on primary key
$app->put(
    "/api/robots/{id:[0-9]+}",
    function () {

    }
);

// Deletes robots based on primary key
$app->delete(
    "/api/robots/{id:[0-9]+}",
    function () {

    }
);

$app->handle();
```

Chaque route est définie avec une méthode qui a le même nom que la requête HTTP. Le premier paramètre est le modèle de la route suivi par une fonction anonyme. La route suivante `/api/robots/{id:[0-9]+}`, par exemple, prends un paramètre ID qui doit nécessairement avoir un format numérique.

Quand une requête URI correspond à une route défini, l'application exécute la fonction anonyme qui lui est liée.

Créer un Model

Notre API fournit des informations sur les 'robots', ces données doivent donc être enregistrées dans une base de données. Le model suivant nous permet d'accéder à la table comme si c'était un objet. Nous avons implémenté quelques règles en utilisant des validateurs. Ainsi nous serons tranquilles car les données respecteront toujours les conditions nécessaires pour notre application :

```

<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;
use Phalcon\Mvc\Model\Message;
use Phalcon\Mvc\Model\Validator\Uniqueness;
use Phalcon\Mvc\Model\Validator\InclusionIn;

class Robots extends Model
{
    public function validation()
    {
        // Type must be: droid, mechanical or virtual
        $this->validate(
            new InclusionIn(
                [
                    "field" => "type",
                    "domain" => [
                        "droid",
                        "mechanical",
                        "virtual",
                    ]
                ]
            )
        );

        // Robot name must be unique
        $this->validate(
            new Uniqueness(
                [
                    "field" => "name",
                    "message" => "The robot name must be unique",
                ]
            )
        );

        // Year cannot be less than zero
        if ($this->year < 0) {
            $this->appendMessage(
                new Message("The year cannot be less than zero")
            );
        }

        // Check if any messages have been produced
        if ($this->validationHasFailed() === true) {
            return false;
        }
    }
}

```

Maintenant nous devons mettre en place la connexion qui sera utilisée par le model and load it within our app :

```

<?php

use Phalcon\Loader;
use Phalcon\Mvc\Micro;
use Phalcon\Di\FactoryDefault;

```

```
use Phalcon\Db\Adapter\Pdo\Mysql as PdoMysql;

// Use Loader() to autoload our model
$loader = new Loader();

$loader->registerNamespaces(
    [
        "Store\\Toys" => __DIR__ . "/models/",
    ]
);

$loader->register();

$di = new FactoryDefault();

// Set up the database service
$di->set(
    "db",
    function () {
        return new PdoMysql(
            [
                "host"      => "localhost",
                "username" => "asimov",
                "password" => "zeroth",
                "dbname"   => "robotics",
            ]
        );
    }
);

// Create and bind the DI to the application
$app = new Micro($di);
```

Récupérer les données

Le premier gestionnaire que l'on a implémenté est celui qui retourne tous les robots à partir d'une méthode GET. Utilisons PHQL pour exécuter une simple requête qui retourne les résultats sous forme de JSON :

```
<?php

// Retrieves all robots
$app->get(
    "/api/robots",
    function () use ($app) {
        $phql = "SELECT * FROM Store\\Toys\\Robots ORDER BY name";

        $robots = $app->modelsManager->executeQuery($phql);

        $data = [];

        foreach ($robots as $robot) {
            $data[] = [
                "id"    => $robot->id,
                "name" => $robot->name,
            ];
        }
    }
);
```

```

        echo json_encode($data);
    }
};

```

PHQL, nous permet d’écrire des requêtes en utilisant un dialect SQL haut niveau et orienté objet qui va traduire la syntaxe SQL des requêtes en fonction du système de base de données que l’on utilise. Le mot clé “use” dans la fonction anonyme nous permet de passer des variables globales sous forme locale facilement.

La recherche par nom ressemblera à cela :

```

<?php

// Searches for robots with $name in their name
$app->get(
    "/api/robots/search/{name}",
    function ($name) use ($app) {
        $phql = "SELECT * FROM Store\\Toys\\Robots WHERE name LIKE :name: ORDER BY_
↪name";

        $robots = $app->modelsManager->executeQuery(
            $phql,
            [
                "name" => "%" . $name . "%"
            ]
        );

        $data = [];

        foreach ($robots as $robot) {
            $data[] = [
                "id" => $robot->id,
                "name" => $robot->name,
            ];
        }

        echo json_encode($data);
    }
);

```

Chercher avec l’identifiant “id” est relativement identique, dans notre cas, nous allons notifier l’utilisateur si le robot n’existe pas :

```

<?php

use Phalcon\Http\Response;

// Retrieves robots based on primary key
$app->get(
    "/api/robots/{id:[0-9]+}",
    function ($id) use ($app) {
        $phql = "SELECT * FROM Store\\Toys\\Robots WHERE id = :id:";

        $robot = $app->modelsManager->executeQuery(
            $phql,
            [
                "id" => $id,
            ]
        );
    }
);

```

```
)->getFirst();

// Create a response
$response = new Response();

if ($robot === false) {
    $response->setJsonContent(
        [
            "status" => "NOT-FOUND"
        ]
    );
} else {
    $response->setJsonContent(
        [
            "status" => "FOUND",
            "data" => [
                "id" => $robot->id,
                "name" => $robot->name
            ]
        ]
    );
}

return $response;
}
);
```

Ajouter des données

Prenons la données comme une chaîne JSON que l'on insère dans le corps de la requête. Nous allons utiliser PHQL pour l'insertion :

```
<?php

use Phalcon\Http\Response;

// Adds a new robot
$app->post(
    "/api/robots",
    function () use ($app) {
        $robot = $app->request->getJsonRawBody();

        $phql = "INSERT INTO Store\\Toys\\Robots (name, type, year) VALUES (:name:, ,
↪:type:, :year:)";

        $status = $app->modelsManager->executeQuery(
            $phql,
            [
                "name" => $robot->name,
                "type" => $robot->type,
                "year" => $robot->year,
            ]
        );
    }
);
```

```

// Create a response
$response = new Response();

// Check if the insertion was successful
if ($status->success() === true) {
    // Change the HTTP status
    $response->setStatusCode(201, "Created");

    $robot->id = $status->getModel()->id;

    $response->setJsonContent(
        [
            "status" => "OK",
            "data"    => $robot,
        ]
    );
} else {
    // Change the HTTP status
    $response->setStatusCode(409, "Conflict");

    // Send errors to the client
    $errors = [];

    foreach ($status->getMessages() as $message) {
        $errors[] = $message->getMessage();
    }

    $response->setJsonContent(
        [
            "status"    => "ERROR",
            "messages" => $errors,
        ]
    );
}

return $response;
}
);

```

Modifier les données

La modification de données est similaire à l'insertion. L'ID passé en paramètre indique quel robot doit être modifié :

```

<?php
use Phalcon\Http\Response;

// Updates robots based on primary key
$app->put(
    "/api/robots/{id:[0-9]+}",
    function ($id) use ($app) {
        $robot = $app->request->getJsonRawBody();

        $phql = "UPDATE Store\\Toys\\Robots SET name = :name:, type = :type:, year =
↳:year: WHERE id = :id:";

        $status = $app->modelsManager->executeQuery(

```

```
        $phql,
        [
            "id"    => $id,
            "name"  => $robot->name,
            "type"  => $robot->type,
            "year"  => $robot->year,
        ]
    );

    // Create a response
    $response = new Response();

    // Check if the insertion was successful
    if ($status->success() === true) {
        $response->setJsonContent(
            [
                "status" => "OK"
            ]
        );
    } else {
        // Change the HTTP status
        $response->setStatusCode(409, "Conflict");

        $errors = [];

        foreach ($status->getMessages() as $message) {
            $errors[] = $message->getMessage();
        }

        $response->setJsonContent(
            [
                "status"    => "ERROR",
                "messages" => $errors,
            ]
        );
    }

    return $response;
}

);
```

Supprimer des données

La suppression de données est relativement identique à la modification. L'identifiant est aussi passé en paramètre pour indiquer quel robot doit être supprimé :

```
<?php

use Phalcon\Http\Response;

// Deletes robots based on primary key
$app->delete(
    "/api/robots/{id:[0-9]+}",
    function ($id) use ($app) {
        $phql = "DELETE FROM Store\\Toys\\Robots WHERE id = :id:";
```



```

        $status = $app->modelsManager->executeQuery(
            $phql,
            [
                "id" => $id,
            ]
        );

        // Create a response
        $response = new Response();

        if ($status->success() === true) {
            $response->setJsonContent(
                [
                    "status" => "OK"
                ]
            );
        } else {
            // Change the HTTP status
            $response->setStatusCode(409, "Conflict");

            $errors = [];

            foreach ($status->getMessages() as $message) {
                $errors[] = $message->getMessage();
            }

            $response->setJsonContent(
                [
                    "status" => "ERROR",
                    "messages" => $errors,
                ]
            );
        }

        return $response;
    }
);

```

Tester notre application

En utilisant `curl` nous allons tester chaque route de notre application et vérifier que les opérations fonctionnent correctement.

Récupérer tous les robots :

```

curl -i -X GET http://localhost/my-rest-api/api/robots

HTTP/1.1 200 OK
Date: Tue, 21 Jul 2015 07:05:13 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 117
Content-Type: text/html; charset=UTF-8

[{"id": "1", "name": "Robotina"}, {"id": "2", "name": "Astro Boy"}, {"id": "3", "name": "Terminator"}]

```

Chercher un robot par son nom :

```
curl -i -X GET http://localhost/my-rest-api/api/robots/search/Astro

HTTP/1.1 200 OK
Date: Tue, 21 Jul 2015 07:09:23 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 31
Content-Type: text/html; charset=UTF-8

[{"id":"2","name":"Astro Boy"}]
```

Récupérer un robot par son ID :

```
curl -i -X GET http://localhost/my-rest-api/api/robots/3

HTTP/1.1 200 OK
Date: Tue, 21 Jul 2015 07:12:18 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 56
Content-Type: text/html; charset=UTF-8

{"status":"FOUND","data":{"id":"3","name":"Terminator"}}
```

Insérer un nouveau robot :

```
curl -i -X POST -d '{"name":"C-3PO","type":"droid","year":1977}'
http://localhost/my-rest-api/api/robots

HTTP/1.1 201 Created
Date: Tue, 21 Jul 2015 07:15:09 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 75
Content-Type: text/html; charset=UTF-8

{"status":"OK","data":{"name":"C-3PO","type":"droid","year":1977,"id":"4"}}
```

Essayer d'insérer un nouveau robot avec le nom d'un robot existant :

```
curl -i -X POST -d '{"name":"C-3PO","type":"droid","year":1977}'
http://localhost/my-rest-api/api/robots

HTTP/1.1 409 Conflict
Date: Tue, 21 Jul 2015 07:18:28 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 63
Content-Type: text/html; charset=UTF-8

{"status":"ERROR","messages":["The robot name must be unique"]}
```

Modifier un robot avec un type inconnu :

```
curl -i -X PUT -d '{"name":"ASIMO","type":"humanoid","year":2000}'
http://localhost/my-rest-api/api/robots/4

HTTP/1.1 409 Conflict
Date: Tue, 21 Jul 2015 08:48:01 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 104
Content-Type: text/html; charset=UTF-8
```

```
{ "status": "ERROR", "messages": ["Value of field 'type' must be part of
    list: droid, mechanical, virtual"] }
```

Enfin, la suppression de robots :

```
curl -i -X DELETE http://localhost/my-rest-api/api/robots/4

HTTP/1.1 200 OK
Date: Tue, 21 Jul 2015 08:49:29 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 15
Content-Type: text/html; charset=UTF-8

{ "status": "OK" }
```

Conclusion

Comme nous l'avons vu, développer une API RESTful avec Phalcon est simple. Plus loin dans la documentation, nous expliqueront en détail comment utiliser une micro application et nous aborderont aussi le langage *PHQL* plus en détail.

2.2.8 Liste d'exemple de projets

Les exemples suivants sont des applications complètes que vous pouvez utiliser pour apprendre Phalcon et servir de base pour vos propres sites/applications:

2.3 Components

2.3.1 Injection de dépendance/Localisation de Service

Before reading this section, it is wise to read *the section which explains why Phalcon uses service location and dependency injection*.

Phalcon\Di est un composant qui met en œuvre l'Injection de Dépendance et la Localisation de Service et il est lui-même un conteneur pour cela.

Comme Phalcon est fortement découplé, *Phalcon\Di* est essentiel pour intégrer les différents composants dans le framework. Le développeur peut également exploiter ce composant pour injecter des dépendances et gérer les instances globales des différentes classes utilisées dans l'application.

A la base, ce composant implémente le patron *Inversion de Contrôle*. En appliquant cela, les objets ne reçoivent pas leur dépendances en utilisant des accesseurs ou des constructeurs, mais en interrogeant un service injecteur de dépendance. Ceci réduit la complexité tant qu'il n'y aura qu'une seule façon d'obtenir les dépendances nécessaires au composant.

De plus, ce patron augmente la testabilité du code, le rendant ainsi moins vulnérable aux erreurs.

Inscription de services dans le conteneur

Le framework comme le développeur peuvent inscrire des service. Lorsqu'un composant A nécessite un composant B (ou une instance de cette classe) pour fonctionner, il peut demander le composant B au conteneur plutôt que créer une

nouvelle instance du composant B.

Cette façon de faire procure plusieurs avantages:

- Nous pouvons facilement remplacer un composant par un autre réalisé par nos soins ou un tiers.
- Nous avons un contrôle complet sur l'initialisation de l'objet, nous permettant de préparer les objets comme nous le souhaitons avant de les livrer aux composants.
- Nous pouvons récupérer des instances globales de composant, d'une manière structurée et unifiée.

Plusieurs styles de définitions permettent d'inscrire les services:

Inscription simple

Comme vu précédemment, il existe plusieurs façons d'inscrire un service. Voici ceux que nous appelons "simple":

Chaîne de caractères (string)

Ce mode s'attend à un nom de classe valide, retournant un objet de la classe spécifiée, qui si elle n'est pas chargée, le sera en utilisant un chargeur automatique de classes. Ce mode de définition ne permet pas de spécifier des arguments pour constructeur de la classe ni des paramètres:

```
<?php
// Return new Phalcon\Http\Request();
$di->set (
    "request",
    "Phalcon\\Http\\Request"
);
```

Instance de classe

Ce mode s'attend à un objet. Comme l'objet n'a pas besoin d'être résolu puisqu'il est déjà un objet, certains diront que ce n'est pas vraiment une injection de dépendance. Toutefois, cela peut être utile si vous souhaitez forcer la dépendance retournée à être toujours le même objet ou la même valeur:

```
<?php
use Phalcon\Http\Request;

// Return new Phalcon\Http\Request();
$di->set (
    "request",
    new Request ()
);
```

Fermetures (Closures)/Fonctions anonymes:

Cette méthode offre une grande liberté pour construire les dépendances comme désirées, cependant il est difficile de changer extérieurement sans avoir à changer complètement la définition de la dépendance:

```
<?php

use Phalcon\Db\Adapter\Pdo\Mysql as PdoMysql;

$di->set(
    "db",
    function () {
        return new PdoMysql(
            [
                "host"      => "localhost",
                "username" => "root",
                "password" => "secret",
                "dbname"    => "blog",
            ]
        );
    }
);
```

Certaines limites peuvent être contournées en passant des variables supplémentaires à l'environnement de la fermeture:

```
<?php

use Phalcon\Config;
use Phalcon\Db\Adapter\Pdo\Mysql as PdoMysql;

$config = new Config(
    [
        "host"      => "127.0.0.1",
        "username" => "user",
        "password" => "pass",
        "dbname"    => "my_database",
    ]
);

// Utilisation de la variable $config dans la portée courante.
$di->set(
    "db",
    function () use ($config) {
        return new PdoMysql(
            [
                "host"      => $config->host,
                "username" => $config->username,
                "password" => $config->password,
                "dbname"    => $config->name,
            ]
        );
    }
);
```

Vous pouvez également accéder à d'autres services DI en utilisant la méthode `get ()` :

```
<?php

use Phalcon\Config;
use Phalcon\Db\Adapter\Pdo\Mysql as PdoMysql;

$di->set(
    "config",
```

```
function () {  
    return new Config(  
        [  
            "host"      => "127.0.0.1",  
            "username" => "utilisateur",  
            "password" => "mot_de_passe",  
            "dbname"   => "ma_base_de_donnees",  
        ]  
    );  
}  
);  
  
// Avec le service 'config' du DI  
$di->set(  
    "db",  
    function () {  
        $config = $this->get("config");  
  
        return new PdoMysql(  
            [  
                "host"      => $config->host,  
                "username" => $config->username,  
                "password" => $config->password,  
                "dbname"   => $config->name,  
            ]  
        );  
    }  
);
```

Inscription Complexe

S'il est nécessaire de changer la définition d'un service sans devoir instancier/résoudre le service, nous devons alors définir les services en utilisant la syntaxe tableau. La définition d'un service sous forme de tableau peut être un peu plus verbeuse:

```
<?php  
  
use Phalcon\Logger\Adapter\File as LoggerFile;  
  
// Inscription d'un service "logger" avec un nom de classe et ses paramètres  
$di->set(  
    "logger",  
    [  
        "className" => "Phalcon\\Logger\\Adapter\\File",  
        "arguments" => [  
            [  
                "type"  => "parameter",  
                "value" => "../apps/logs/error.log",  
            ]  
        ]  
    ]  
);  
  
// En utilisant une fonction anonyme  
$di->set(  
    "logger",
```

```
function () {
    return new LoggerFile("../apps/logs/error.log");
}
);
```

Les deux inscriptions précédentes produisent le même résultat. Cependant, la définition sous forme de tableau permet une altération des paramètres du service si nécessaire:

```
<?php

// Changement du nom de service
$di->getService("logger")->setClassName("MyCustomLogger");

// Changement du premier paramètre sans instancier le logger
$di->getService("logger")->setParameter(
    0,
    [
        "type" => "parameter",
        "value" => "../apps/logs/error.log",
    ]
);
```

De plus, en utilisant la syntaxe tableau, vous pouvez exploiter trois type d'injection de dépendance:

Injection de constructeur

Ce type d'injection transmet les dépendances au constructeur de la classe. Admettons que nous ayons le composant suivant:

```
<?php

namespace SomeApp;

use Phalcon\Http\Response;

class SomeComponent
{
    /**
     * @var Response
     */
    protected $_response;

    protected $_someFlag;

    public function __construct(Response $response, $someFlag)
    {
        $this->_response = $response;
        $this->_someFlag = $someFlag;
    }
}
```

Le service peut être inscrit de cette façon:

```
<?php

$di->set (
    "response",
    [
        "className" => "Phalcon\\Http\\Response"
    ]
);

$di->set (
    "someComponent",
    [
        "className" => "SomeApp\\SomeComponent",
        "arguments" => [
            [
                "type" => "service",
                "name" => "response",
            ],
            [
                "type" => "parameter",
                "value" => true,
            ],
        ]
    ]
);
```

Le service “response” (*Phalcon\\Http\\Response*) est résolu pour être transmis en premier argument au constructeur, alors que le second est une valeur booléenne (true) transmise telle quelle.

Injection d’accesseur

Les classes peuvent posséder des accesseurs pour injecter des dépendances optionnelles. Nos précédentes classes peuvent être modifiées pour accepter des dépendances avec des accesseurs:

```
<?php

namespace SomeApp;

use Phalcon\\Http\\Response;

class SomeComponent
{
    /**
     * @var Response
     */
    protected $_response;

    protected $_someFlag;

    public function setResponse(Response $response)
    {
        $this->_response = $response;
    }
}
```



```

public function setFlag($someFlag)
{
    $this->_someFlag = $someFlag;
}

```

Un service avec une injection par accesseur peut être inscrite comme suit:

```

<?php

$di->set(
    "response",
    [
        "className" => "Phalcon\\Http\\Response",
    ]
);

$di->set(
    "someComponent",
    [
        "className" => "SomeApp\\SomeComponent",
        "calls" => [
            [
                "method" => "setResponse",
                "arguments" => [
                    [
                        "type" => "service",
                        "name" => "response",
                    ]
                ]
            ],
            [
                "method" => "setFlag",
                "arguments" => [
                    [
                        "type" => "parameter",
                        "value" => true,
                    ]
                ]
            ]
        ]
    ]
);

```

Injection de propriétés

Une stratégie moins courante est d'injecter directement des dépendances ou des paramètres aux attributs publics de la classe:

```

<?php

namespace SomeApp;

use Phalcon\Http\Response;

class SomeComponent

```

```
{  
    /**  
     * @var Response  
     */  
    public $response;  
  
    public $someFlag;  
}
```

Un service avec un injection de propriétés peut être inscrite comme suit:

```
<?php  
  
$di->set(  
    "response",  
    [  
        "className" => "Phalcon\\Http\\Response",  
    ]  
);  
  
$di->set(  
    "someComponent",  
    [  
        "className" => "SomeApp\\SomeComponent",  
        "properties" => [  
            [  
                "name" => "response",  
                "value" => [  
                    "type" => "service",  
                    "name" => "response",  
                ],  
            ],  
            [  
                "name" => "someFlag",  
                "value" => [  
                    "type" => "parameter",  
                    "value" => true,  
                ],  
            ],  
        ],  
    ]  
);
```

Les différents types de paramètre supportés sont les suivants:

Type	Description	Exemple
paramètre	Représente une valeur littérale transmise en paramètre	["type" => "parameter", "value" => 1234]
service	Représente un autre service dans le conteneur de services	["type" => "service", "name" => "request"]
instance	Représente un objet qui doit être construit dynamiquement	["type" => "instance", "className" => "DateTime", "arguments" => ["now"]]

La résolution d'un service dont la définition est complexe peut être légèrement plus lente que pour les définitions simples vues précédemment. Cependant, ceci fournit une approche plus robuste pour définir et injecter des services.

Le mélange de différents types de définitions est permis. Chacun décide de la méthode d'inscription des service la plus appropriée en fonction des besoins de l'application.

Forme tableau

L'écriture sous forme de tableau est possible pour inscrire des services:

```
<?php

use Phalcon\Di;
use Phalcon\Http\Request;

// Création du conteneur d'Injection de Dépendance
$di = new Di();

// D'après son nom
$di["request"] = "Phalcon\\Http\\Request";

// Chargement tardif avec une fonction anonyme
$di["request"] = function () {
    return new Request();
};

// En inscrivant directement une instance
$di["request"] = new Request();

// Avec un tableau de définition
$di["request"] = [
    "className" => "Phalcon\\Http\\Request",
];
```

Dans les exemples précédents, lorsque le framework doit accéder aux données demandées, il interroge le service identifié en tant que ‘request’ dans le conteneur. Le conteneur retourne une instance du service demandé. Le développeur peut éventuellement remplacer les composants selon ses besoins.

Chacune des méthodes (vues dans les exemples précédents) utilisée pour définir/inscrire un service a ses avantages et ses inconvénients. C’est au développeur de choisir laquelle utiliser en fonction des exigences.

Définir un service par une chaîne de caractères est simple mais manque de souplesse. Définir un service par un tableau offre plus de flexibilité mais rend le code plus compliqué. La fonction lambda est un bon équilibre entre les deux mais risque de nécessiter plus de maintenance que nécessaire.

Phalcon\Di offre un chargement tardif pour chaque service qu’il stocke. A moins que le développeur choisisse d’instancier directement et de le stocker dans le conteneur, chaque objet qui lui est confié (via tableau, chaîne de caractères, etc.) sera chargé tardivement c.à.d instancié lors de la demande.

Résolution de services

L’obtention d’un service à partir d’un conteneur peut se faire simplement en utilisant la méthode “get”. Une nouvelle instance du service sera retournée:

```
<?php $request = $di->get("request");
```

Ou en invoquant la méthode magique:

```
<?php

$request = $di->getRequest();
```

Ou en utilisant l’écriture tableau:

```
<?php

$request = $di["request"];
```

Les arguments sont transmis au constructeur en ajoutant un tableau en paramètre de la méthode “get”:

```
<?php

// new MyComponent("some-parameter", "other")
$component = $di->get(
    "MyComponent",
    [
        "some-parameter",
        "other",
    ]
);
```

Événements

Phalcon\Di est capable d’envoyer des événements à un *EventsManager* s’il existe. Les événements sont déclenchés en utilisant le type “di”. Les événements qui retournent la valeur booléenne faux peuvent interrompre l’opération en cours. Les événements suivants sont supportés:

Nom d'événement	Déclenchement	Stoppe l'opération ?	Destinataire
beforeServiceResolve	Déclenché avant la résolution de service. Les écouteurs reçoivent le nom du service ainsi que les paramètres qui lui sont transmis	Non	Écouteurs
afterServiceResolve	Déclenché après la résolution de service. Les écouteurs reçoivent le nom du service, l’instance, ainsi que les paramètres qui lui sont transmis	Non	Écouteurs

Services partagés

Les services peuvent être inscrits en tant que service “partagé”. Ceci signifie qu’ils se comporteront toujours comme des *singletons*. Une fois que le service est résolu une première fois la même instance est systématiquement retournée lorsqu’un consommateur récupère le service depuis le conteneur:

```
<?php

use Phalcon\Session\Adapter\Files as SessionFiles;

// Inscription du service de session comme "toujours partagé"
$di->setShared(
    "session",
    function () {
        $session = new SessionFiles();

        $session->start();

        return $session;
    }
);
```

```
// Localisation du service pour la première fois
$session = $di->get("session");

// Retourne l'objet instancié initialement
$session = $di->getSession();
```

Une autre façon d’inscrire des services partagés est de transmettre “true” au troisième paramètre de “set”:

```
<?php

// Inscription du service de session comme "toujours partagé"
$di->set(
    "session",
    function () {
        // ...
    },
    true
);
```

Si un service n’est pas inscrit comme partagé et vous voulez être sûr d’accéder à une instance partagée à chaque fois que le service est obtenu auprès de DI, vous pouvez utiliser la méthode ‘getShared’:

```
<?php

$request = $di->getShared("request");
```

Manipuler les services individuellement

Une fois qu’un service est inscrit dans le conteneur de services, vous pouvez le récupérer pour le manipuler individuellement:

```
<?php

use Phalcon\Http\Request;

// Inscription du service "request"
$di->set("request", "Phalcon\\Http\\Request");

// Récupère le service
$requestService = $di->getService("request");

// Modifie sa définition
$requestService->setDefinition(
    function () {
        return new Request();
    }
);

// Le transforme en "partagé"
$requestService->setShared(true);

// Résolution du service (retourne une instance de Phalcon\Http\Request)
$request = $requestService->resolve();
```

Instanciation de classes via le Conteneur de Services

Lorsque vous demandez un service au conteneur de services, s'il n'en trouve pas un avec le même nom, il tente de charger une classe avec le même nom. Grâce à ce comportement nous pouvons remplacer n'importe quelle autre simplement en inscrivant un service avec son nom:

```
<?php

// Inscription d'un contrôleur en tant que service
$di->set (
    "IndexController",
    function () {
        $component = new Component ();

        return $component;
    },
    true
);

// Inscription d'un contrôleur en tant que service
$di->set (
    "MyOtherComponent",
    function () {
        // Actuellement retourne un autre composant
        $component = new AnotherComponent ();

        return $component;
    }
);

// Création d'un instance via le conteneur de service.
$myComponent = $di->get ("MyOtherComponent");
```

Vous pouvez profiter de ceci en instanciant toujours vos classes depuis le conteneur de services (même s'ils ne sont pas inscrits en tant que service). Le DI prendra par défaut un chargeur automatique valide pour charger la classe. En faisant comme ceci, vous pourrez aisément remplacer n'importe quelle classe en implémentant une définition pour elle.

Injection automatique pour le DI lui-même

Si une classe ou un composant ai besoin que le DI localise lui-même les services, le DI peut automatiquement s'injecter les instances qu'il crée. Pour ceci, vous devez implémenter l'interface *Phalcon\Di\InjectionAwareInterface* dans vos classes:

```
<?php

use Phalcon\DiInterface;
use Phalcon\Di\InjectionAwareInterface;

class MyClass implements InjectionAwareInterface
{
    /**
     * @var DiInterface
     */
    protected $_di;
```

```

    public function setDi(DiInterface $di)
    {
        $this->_di = $di;
    }

    public function getDi()
    {
        return $this->_di;
    }
}

```

Une fois que le service est résolu, la variable `$di` sera transmise automatiquement à `setDi()`:

```

<?php

// Inscription du service
$di->set("myClass", "MyClass");

// Résolution du service (NOTE: $myClass->setDi($di) est automatiquement appelée)
$myClass = $di->get("myClass");

```

Organisation des services en fichiers

Vous pouvez mieux organiser votre application en déplaçant l'inscription des services dans des fichiers distincts au lieu de tout mettre dans l'amorce de l'application:

```

<?php

$di->set(
    "router",
    function () {
        return include "../app/config/routes.php";
    }
);

```

Ainsi le fichier ("`../app/config/routes.php`") renvoi l'objet résolu:

```

<?php

$router = new MyRouter();

$router->post("/login");

return $router;

```

Accès au DI de manière statique

Si nécessaire, vous pouvez accéder au dernier DI créé dans une fonction statique de la façon suivante:

```

<?php

use Phalcon\Di;

class SomeComponent
{

```

```
public static function someMethod()
{
    // Récupère le service de session
    $session = Di::getDefault()->getSession();
}
```

Construction du DI par défaut

Bien que le caractère découplé de Phalcon offre une grande liberté et flexibilité, peut-être que nous voulons simplement l'utiliser comme un framework full-stack. Pour réaliser ceci, le framework fournit une variante de *Phalcon\Di* appelée *Phalcon\Di\FactoryDefault*. Cette classe inscrit automatiquement les services appropriés qui sont encapsulés dans le framework afin qu'il agisse comme un full-stack.

```
<?php

use Phalcon\Di\FactoryDefault;

$di = new FactoryDefault();
```

Convention de nommage des services

Bien que vous puissiez inscrire les services avec le nom que vous voulez, Phalcon a plusieurs conventions de nommage qui permettent d'obtenir le bon service (built-in) au bon moment.

Nom de service	Description	Par défaut	Partagé
dispatcher	Service de ventilation des contrôleurs	<i>Phalcon\Mvc\Dispatcher</i>	Oui
router	Service de routage	<i>Phalcon\Mvc\Router</i>	Oui
url	Service de génération d'URL	<i>Phalcon\Mvc\Url</i>	Oui
request	HTTP Request Environment Service	<i>Phalcon\Http\Request</i>	Oui
response	HTTP Response Environment Service	<i>Phalcon\Http\Response</i>	Oui
cookies	HTTP Cookies Management Service	<i>Phalcon\Http\Response\Cookies</i>	Oui
filter	Service de filtrage des entrées	<i>Phalcon\Filter</i>	Oui
flash	Service des messages flash	<i>Phalcon\Flash\Direct</i>	Oui
flashSession	Service de session des messages flash	<i>Phalcon\Flash\Session</i>	Oui
session	Service de session	<i>Phalcon\Session\Adapter\Files</i>	Oui
eventsManager	Service de gestion des événements	<i>Phalcon\Events\Manager</i>	Oui
db	Service élémentaire de connexion aux bases de données	<i>Phalcon\Db</i>	Oui
security	Auxiliaires de sécurité	<i>Phalcon\Security</i>	Oui
crypt	Cryptage/Décryptage	<i>Phalcon\Crypt</i>	Oui
tag	Aide de génération HTML	<i>Phalcon\Tag</i>	Oui
escaper	Echappement contextuel	<i>Phalcon\Escaper</i>	Oui
annotations	Analyseur d'annotations	<i>Phalcon\Annotations\Adapter\Memory</i>	Oui
modelsManager	Service de gestion des modèles	<i>Phalcon\Mvc\Model\Manager</i>	Oui
modelsMeta-data	Service de métadonnées des modèles	<i>Phalcon\Mvc\Model\MetaData\Memory</i>	Oui
transactionManager	Service de gestion des transactions	<i>Phalcon\Mvc\Model\Transaction\Manager</i>	Oui
modelsCache	Cache pour les modèles coté serveur	Aucun	Non
viewsCache	Cache des fragments de vue coté serveur	Aucun	Non

Création de votre propre DI

Pour remplacer le DI fournit par Phalcon, vous devez soit implémenter l'interface *Phalcon\DiInterface*, soit étendre un existant.

2.3.2 L'architecture MVC

Phalcon offre les classes orientés objets nécessaires pour implémenter l'architecture avec model, vue et contrôleur (plus connue comme **MVC**). Ce patron de conception est très largement utilisé par les autres framework web et applications bureau.

Les avantages du MVC:

- Séparation de la partie métier de la partie interface utilisateur ainsi que de la couche d'accès aux données.
- Repérer plus facilement les dépendances de code afin de faciliter la maintenance.

Si vous décidez d'utiliser le MVC, chaque requête de votre application sera gérée par l'architecture MVC. Les classes de Phalcon sont écrites en C, ce qui offre une haute performance à ce principe, pour une application PHP.

Les modèles

Un modèle représente les informations (données) de l'application et les règles pour manipuler ces données. Les modèles sont principalement utilisés pour gérer l'interaction avec une base de données. Dans la plupart des cas,

chaque table de votre base de données correspondra à un model de votre application. L'essentiel de la logique de votre application sera concentrée sur les models. *En savoir plus*

Les vues

Les vues représentent l'interface utilisateur. Elles sont souvent en HTML, avec du PHP intégré pour exécuter certaines tâches lié principalement à la représentation des données. Les vues s'occupent de retranscrire les données de manière visible sur un navigateur ou tout autre support visuel. *En savoir plus*

Les contrôleurs

Les contrôleurs gèrent le “flux” entre les modèles et les vues. Ils sont responsables de la gestion des requêtes venant du navigateur, d'interroger le modèle pour les données et transmettre ces données à la vue. *En savoir plus*

2.3.3 Utilisation de Contrôleurs

Les contrôleurs fournissent un certain nombre de méthodes qui sont appelées actions. Les actions sont des méthodes du contrôleur qui traitent les requêtes. Par défaut toutes les méthodes publiques d'un contrôleur sont reliées à des actions et sont accessibles par URL. Les actions sont responsables de l'interprétation de la requête et de la création de la réponse. Habituellement les réponses sont sous la forme de vues mais il existe d'autres façons de créer des réponses adaptées.

Par exemple, en accédant à une URL comme ceci: <http://localhost/blog/posts/show/2015/the-post-title> Phalcon par défaut décomposera chaque partie comme ceci:

Répertoire Phalcon	blog
Contrôleur	posts
Action	show
Paramètre	2015
Paramètre	the-post-title

Dans ce cas, le contrôleur “PostsController” gèrera cette requête. Il n'y a pas d'emplacement particulier pour placer les contrôleurs dans une application, ils seraient chargés grâce à un *chargeur automatique*, et donc vous restez libre d'organiser vos contrôleurs selon vos besoins.

Les contrôleurs doivent posséder le suffixe “Controller”, et les actions le suffixe “Action”. Un exemple de contrôleur se trouve ci-après:

```
<?php
use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {

    }

    public function showAction($year, $postTitle)
    {

    }
}
```

Les paramètres supplémentaires de l'URI sont définis comme des paramètres de l'action, ainsi ils restent facilement accessibles en utilisant des variables locales. Un contrôleur peut éventuellement étendre *Phalcon\Mvc\Controller*. En faisant ceci, le contrôleur accède facilement aux services de l'application.

Les paramètres sans valeur par défaut sont correctement gérés. La définition de paramètres optionnels se fait comme d'habitude en PHP:

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {

    }

    public function showAction($year = 2015, $postTitle = "some default title")
    {

    }
}
```

Les paramètres sont assignés dans le même ordre qu'ils sont transmis dans la route. Vous pouvez récupérer arbitrairement des paramètres de la façon suivante:

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {

    }

    public function showAction()
    {
        $year      = $this->dispatcher->getParam("year");
        $postTitle = $this->dispatcher->getParam("postTitle");
    }
}
```

Boucle de répartition (dispatch)

La boucle de répartition sera exécutée dans le "Dispatcher" tant qu'il reste des actions à exécuter. Dans l'exemple précédent, seule une action était exécutée. Voyons maintenant comment la méthode `forward()` peut fournir un flot plus complexe d'opération à la boucle de répartition en faisant suivre le fil d'exécution à un autre contrôleur ou une autre action.

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
```

```
{  
    public function indexAction()  
    {  
    }  
  
    public function showAction($year, $postTitle)  
    {  
        $this->flash->error(  
            "Vous n'avez pas la permission d'accéder à cette zone"  
        );  
  
        // Redirection du flux à une autre action  
        $this->dispatcher->forward(  
            [  
                "controller" => "users",  
                "action"      => "signin",  
            ],  
        );  
    }  
}
```

Si les utilisateurs n’ont pas la permission d’accéder à une certaine action, ils seront redirigés vers le contrôleur “Users”, action “signin”.

```
<?php  
  
use Phalcon\Mvc\Controller;  
  
class UsersController extends Controller  
{  
    public function indexAction()  
    {  
    }  
  
    public function signinAction()  
    {  
    }  
}
```

Votre application n’est pas limitée en nombre de “forward”, tant qu’il n’y a pas de références circulaires, sinon votre application sera arrêtée. S’il ne reste plus d’action à répartir dans la boucle, le répartiteur invoquera automatiquement la couche vue du MVC qui est gérée par *Phalcon\Mvc\View*.

Initialisation des contrôleurs

Phalcon\Mvc\Controller propose une méthode d’initialisation qui est exécutée en premier avant n’importe quelle action du contrôleur. L’utilisation de la méthode “__construct” est à proscrire.

```
<?php  
  
use Phalcon\Mvc\Controller;  
  
class PostsController extends Controller
```

```

{
    public $settings;

    public function initialize()
    {
        $this->settings = [
            "mySetting" => "value",
        ];
    }

    public function saveAction()
    {
        if ($this->settings["mySetting"] === "value") {
            // ...
        }
    }
}

```

La méthode “initialize” n’est appelée que si l’événement “beforeExecuteRoute” est exécuté avec succès. Ceci évite l’exécution de l’initialiseur sans autorisation.

Si vous souhaitez procéder à une initialisation juste après la construction du contrôleur, vous pouvez définir la méthode “onConstruct”:

```

<?php
use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function onConstruct()
    {
        // ...
    }
}

```

Soyez attentifs au fait que la méthode `onConstruct()` sera exécutée même si aucune action n’existe dans le contrôleur, ou bien que l’utilisateur n’y ait pas accès (selon le contrôle d’accès fournit par le développeur).

Injection de services

Si un contrôleur étend *Phalcon\Mvc\Controller* il a alors facilement accès au conteneur de service de l’application. Si vous avez par exemple inscrit un service comme celui-ci:

```

<?php
use Phalcon\Di;

$di = new Di();

$di->set(
    "storage",
    function () {
        return new Storage(
            "/un/repertoire"
        );
    }
);

```

```
    },  
    true  
);
```

Nous pouvons alors accéder à ce service de plusieurs façons:

```
<?php  
  
use Phalcon\Mvc\Controller;  
  
class FilesController extends Controller  
{  
    public function saveAction()  
    {  
        // Injection de service en accédant seulement à la propriété du même nom  
        $this->storage->save('/un/fichier');  
  
        // Accès au service depuis le DI  
        $this->di->get('storage')->save('/un/fichier');  
  
        // Une autre façon avec l'accesseur magique  
        $this->di->getStorage()->save('/un/fichier');  
  
        // Une autre façon avec l'accesseur magique  
        $this->getDi()->getStorage()->save('/un/fichier');  
  
        // Avec l'écriture tableau  
        $this->di['storage']->save('/un/fichier');  
    }  
}
```

Si vous utilisez Phalcon comme un framework full-stack, vous pouvez consulter les services fournis *par défaut* par le framework.

Requête et réponse

En supposant que le framework fournisse un ensemble de services pré-inscrit, nous allons expliquer comment interagir avec l'environnement HTTP. Le service “request” contient un instance de *Phalcon\Http\Request* et le service “response” est une instance de *Phalcon\Http\Response* qui représente ce qui est renvoyé au client.

```
<?php  
  
use Phalcon\Mvc\Controller;  
  
class PostsController extends Controller  
{  
    public function indexAction()  
    {  
    }  
  
    public function saveAction()  
    {  
        // Vérifie que la requête utilise la méthode POST  
        if ($this->request->isPost()) {  
            // Access POST data  
            $customerName = $this->request->getPost("name");  
        }  
    }  
}
```

```

        $customerBorn = $this->request->getPost("born");
    }
}

```

Normalement, l'objet réponse n'est pas utilisé directement mais il est construit avant l'exécution de l'action. Parfois, comme dans l'événement "afterDispatch" il peut être utile d'accéder à la réponse directement:

```

<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {

    }

    public function notFoundAction()
    {
        // Envoi d'une entête de réponse HTTP 404
        $this->response->setStatusCode(404, "Not Found");
    }
}

```

Vous apprendrez plus sur l'environnement HTTP dans les articles dédiés à *request* et *response*.

Données de Session

Les sessions nous aident à maintenir la persistance des données entre les requêtes. Vous pouvez accéder à *Phalcon\Session\Bag* à partir de n'importe quel contrôleur pour encapsuler les données devant être persistantes.

```

<?php

use Phalcon\Mvc\Controller;

class UserController extends Controller
{
    public function indexAction()
    {
        $this->persistent->name = "Michael";
    }

    public function welcomeAction()
    {
        echo "Welcome, ", $this->persistent->name;
    }
}

```

Utilisation des Services en tant que Contrôleurs

Les services peuvent agir comme des contrôleurs. Les classes contrôleur sont toujours interrogées depuis le conteneur de services. En conséquence, n'importe quelle autre classe inscrite avec son nom peut aisément remplacer un contrôleur:

```
<?php

// Inscription d'un contrôleur en tant que service
$di->set (
    "IndexController",
    function () {
        $component = new Component ();

        return $component;
    }
);

// Inscription d'un contrôleur avec espace de nom en tant que service
$component = new Component ();
$di->set (
    "Backend\\Controllers\\IndexController",
    function () {
        $component = new Component ();

        return $component;
    }
);
```

Les Événements dans les Contrôleurs

Les contrôleurs agissent automatiquement comme des écouteurs pour les événements du *répartiteur*. La réalisation de méthodes avec ces noms d'événements vous permet de créer des points d'interception avant que les actions ne soient exécutées:

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function beforeExecuteRoute($dispatcher)
    {
        // Ceci est exécuté avant chaque action trouvée
        if ($dispatcher->getActionName() === "save") {
            $this->flash->error(
                "You don't have permission to save posts"
            );

            $this->flash->error("Vous n'avez pas l'autorisation d'enregistrer des_
↵annonces");
            $this->dispatcher->forward(
                [
                    "controller" => "home",
                    "action"      => "index",
                ]
            );

            return false;
        }
    }
}
```



```

public function afterExecuteRoute($dispatcher)
{
    // Exécutée après chaque action trouvée
}

```

2.3.4 Travailler avec les Modèles

Un modèle représente l'information (donnée) d'une application et les règles pour manipuler cette donnée. Les modèles sont principalement utilisés pour gérer les règles d'interaction avec la table correspondante dans la base données. La plupart du temps, à chaque table dans la base correspondra un modèle dans votre application. L'essentiel de la logique métier de votre application sera concentré dans les modèles.

Phalcon\Mvc\Model est la base de chaque modèle dans une application Phalcon. Il fournit une indépendance vis à vis de la base de données, une fonctionnalité _CRUD élémentaire, des capacités de recherche avancées et la possibilité de relier les modèles entre eux au travers d'autres service. *Phalcon\Mvc\Model* évite la nécessité d'utiliser des instructions SQL parce qu'il traduit dynamiquement les méthodes vers les opérations du moteur de bases de données respectif.

Les modèles sont prévus pour travailler avec les bases de données sur une couche élevée d'abstraction. Si vous devez exploiter des bases de données à un bas niveau consultez la documentation du composant *Phalcon\Db*.

Création de modèles

Un modèle est une classe qui étend *Phalcon\Mvc\Model*. Son nom de classe doit suivre la notation camel:

```

<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class RobotParts extends Model
{
}

```

Si vous utilisez PHP 5.4/5.5, il est recommandé que vous déclariez chaque colonne qui fait partie du modèle afin de préserver la mémoire et de réduire les allocations en mémoire.

Par défaut, le modèle "Store\Toys\RobotParts" fait référence à la table "robot_parts". Si vous souhaitez spécifiez un autre nom pour la table de correspondance, vous pouvez utiliser la méthode `setSource()`:

```

<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class RobotParts extends Model
{
    public function initialize()
    {
        $this->setSource("toys_robot_parts");
    }
}

```

```
}  
}
```

Le modèle RobotParts est désormais relié à la table “toys_robots_parts”. La méthode `initialize()` facilite la mise en place d’un comportement personnalisé comme par exemple une table différente.

La méthode `initialize()` n’est invoquée qu’une seule fois lors de la requête, il est destiné à effectuer des initialisations qui s’appliquent à toutes les instances du modèle créées au sein de l’application. Si vous voulez réaliser des tâches d’initialisation à chaque instantiation vous le pouvez avec la méthode `onConstruct()`:

```
<?php  
  
namespace Store\Toys;  
  
use Phalcon\Mvc\Model;  
  
class RobotParts extends Model  
{  
    public function onConstruct()  
    {  
        // ...  
    }  
}
```

Propriétés publiques contre Accesseurs

Les modèles peuvent être implémentés avec des propriétés à portée publique, ce qui signifie que chaque propriété peut être lue ou écrite sans aucune restriction à partir de n’importe quel code qui instancie le modèle:

```
<?php  
  
namespace Store\Toys;  
  
use Phalcon\Mvc\Model;  
  
class Robots extends Model  
{  
    public $id;  
  
    public $name;  
  
    public $price;  
}
```

Avec des accesseurs, vous contrôlez quelles sont les propriétés qui sont visibles publiquement et vous pouvez effectuer diverses transformations sur les données (qui ne seraient pas possible autrement) ainsi qu’ajouter des règles de validation sur les données portées par l’objet:

```
<?php  
  
namespace Store\Toys;  
  
use InvalidArgumentException;  
use Phalcon\Mvc\Model;  
  
class Robots extends Model
```

```

{
    protected $id;

    protected $name;

    protected $price;

    public function getId()
    {
        return $this->id;
    }

    public function setName($name)
    {
        // Le nom est-il trop court ?
        if (strlen($name) < 10) {
            throw new InvalidArgumentException(
                "Le nom est trop court"
            );
        }

        $this->name = $name;
    }

    public function getName()
    {
        return $this->name;
    }

    public function setPrice($price)
    {
        // Les prix négatifs sont interdits
        if ($price < 0) {
            throw new InvalidArgumentException(
                "Le prix ne peut être négatif"
            );
        }

        $this->price = $price;
    }

    public function getPrice()
    {
        // Conversion de la valeur en type double avant utilisation
        return (double) $this->price;
    }
}

```

Les propriétés publiques sont moins complexes à développer. Cependant, les accesseurs augmentent grandement la testabilité, l'extensibilité et la maintenabilité des applications. C'est au développeur de décider quelle est la stratégie est la plus appropriée pour l'application en cours de création. L'ORM est compatible avec les deux approches de définition de propriétés.

Les tirets bas (`_`) dans les noms de propriétés peuvent être problématiques avec les accesseurs

Si vous utilisez des tirets bas dans les noms de propriété, vous devez toujours utiliser la forme camelcase pour la déclaration de vos accesseurs pour une utilisation des méthodes magiques (par ex. `$model->getPropertyName` au lieu de `$model->getProperty_name`, `$model->findByPropertyName` au lieu de `$model->findByProperty_name`, etc.). Comme

le système s’attend à une forme camelcase, et que les tirets bas sont généralement supprimés, il est recommandé de nommer vos propriétés de la manière indiquée dans la documentation. Vous pouvez utiliser un mapping de colonnes (comme décrit avant) pour assurer une bonne correspondance entre vos propriétés et les homologues dans la base de données.

Comprendre le lien entre les Enregistrements et les Objets

Chaque instance d’un modèle représente une ligne dans la table. Vous accédez facilement aux données de l’enregistrement en lisant les propriétés de l’objet. Par exemple, pour une table “robots” avec ces enregistrements:

```
mysql> select * from robots;
+----+-----+-----+-----+
| id | name      | type      | year |
+----+-----+-----+-----+
|  1 | Robotina  | mechanical | 1972 |
|  2 | Astro Boy | mechanical | 1952 |
|  3 | Terminator | cyborg    | 2029 |
+----+-----+-----+-----+
3 rows in set (0.00 sec)
```

Vous pourriez trouver un enregistrement particulier d’après sa clé primaire et imprimer son nom:

```
<?php
use Store\Toys\Robots;

// Trouve l'enrgt avec id = 3
$robot = Robots::findFirst(3);

// Imprime "Terminator"
echo $robot->name;
```

Une fois que l’enregistrement est en mémoire, vous pouvez effectuer des modifications sur ces données et enregistrer les changements:

```
<?php
use Store\Toys\Robots;

$robot = Robots::findFirst(3);

$robot->name = "RoboCop";

$robot->save();
```

Comme vous pouvez le constater, il n’est pas nécessaire d’utiliser directement des instructions SQL. *Phalcon\Mvc\Model* fournit une haute abstraction de la base de données pour les applications web.

Trouver des enregistrements

Phalcon\Mvc\Model offre également différentes méthodes pour chercher des enregistrements. Les exemples qui suivent vous montrent comment extraire un ou plusieurs enregistrements à partir d’un modèle:

```
<?php
use Store\Toys\Robots;
```

```
// Combien y-a-t'il de robots ?
$robots = Robots::find();
echo "There are ", count($robots), "\n";

// Combien y-a-t'il de robots 'mechanical' ?
$robots = Robots::find("type = 'mechanical'");
echo "There are ", count($robots), "\n";

// Récupère et imprime les robots 'virtual' par ordre de nom
$robots = Robots::find(
    [
        "type = 'virtual'",
        "order" => "name",
    ]
);
foreach ($robots as $robot) {
    echo $robot->name, "\n";
}

// Récupère les 100 premier robots 'virtual' par ordre de nom
$robots = Robots::find(
    [
        "type = 'virtual'",
        "order" => "name",
        "limit" => 100,
    ]
);
foreach ($robots as $robot) {
    echo $robot->name, "\n";
}
```

Si vous voulez trouver un enregistrement d'après une donnée externe (telle qu'une entrée utilisateur) ou une variable, vous devez utiliser la *liaison de paramètres*.

Vous pouvez également utiliser la méthode `findFirst()` pour récupérer le premier enregistrement qui correspond au critère fournit:

```
<?php
use Store\Toys\Robots;

// Quel est le premier robot dans la table robots ?
$robot = Robots::findFirst();
echo "The robot name is ", $robot->name, "\n";

// Quel est le premier robot 'mechanical' dans la table robots ?
$robot = Robots::findFirst("type = 'mechanical'");
echo "The first mechanical robot name is ", $robot->name, "\n";

// Récupère le premier robot 'virtual' par ordre de nom
$robot = Robots::findFirst(
    [
        "type = 'virtual'",
        "order" => "name",
    ]
);
echo "The first virtual robot name is ", $robot->name, "\n";
```

Les deux méthodes `find()` et `findFirst()` acceptent un tableau associatif spécifiant les critères de recherche:

```
<?php

use Store\Toys\Robots;

$robot = Robots::findFirst(
    [
        "type = 'virtual'",
        "order" => "name DESC",
        "limit" => 30,
    ]
);

$robots = Robots::find(
    [
        "conditions" => "type = ?1",
        "bind"        => [
            1 => "virtual",
        ]
    ]
);
```

Les différentes options de requête sont:

Paramètre	Description	Exemple
conditions	Conditions pour l'opération de recherche. Il est utilisé pour extraire seulement les enregistrements qui répondent au critère spécifié. Par défaut <i>Phalcon\Mvc\Model</i> suppose que les conditions sont en premier paramètre.	"conditions" => "name LIKE 'steve%'"
columns	Spécifie les colonnes à renvoyer au lieu de toutes colonnes du modèles. Avec cette option, l'objet est incomplet lorsqu'il est retourné	"columns" => "id, name"
bind	Bind est utilisé conjointement avec des options en remplaçant des espaces réservés et échappant les valeurs améliorant ainsi la sécurité	"bind" => array("status" => "A", "type" => "some-time")
bind-Types	Lors de la liaison de paramètres, vous pouvez utiliser ce paramètre pour introduire une conversion de type du paramètre lié, augmentant encore la sécurité	"bindTypes" => array(Column::BIND_PARAM_STR, Column::BIND_PARAM_INT)
order	Est utilisé pour trier le résultat. Un ou plusieurs champs séparés par une virgule.	"order" => "name DESC, status"
limit	Limite le résultat à une certaine plage	"limit" => 10
offset	Décale le resultat d'un certain nombre de lignes.	"offset" => 5
group	Collecte les données au travers de plusieurs enregistrement et regroupe les résultats selon une ou plusieurs colonnes	"group" => "name, status"
for_update	Avec cette option, doc: <i>Phalcon\Mvc\Model</i> <.. <i>Phalcon\Mvc\Model</i> > lit les dernières données disponibles en activant un verrou exclusif sur chaque enregistrement	"for_update" => true
shared_lock	Avec cette option, doc: <i>Phalcon\Mvc\Model</i> <.. <i>Phalcon\Mvc\Model</i> > lit les dernières données disponibles en activant un verrou partagé sur chaque enregistrement	"shared_lock" => true
cache	Met en cache le résultat, réduisant les accès au système relationnel	"cache" => array("lifetime" => 3600, "key" => "my-find-key")
hydration	Définit la stratégie d'hydratation pour alimenter chaque enregistrement du résultat	"hydration" => Resultset::HYDRATE_OBJECTS

Si vous préférez, il existe une façon plus orientée objet pour créer des requêtes plutôt qu'utiliser un tableau de paramètres:

```
<?php
use Store\Toys\Robots;

$robots = Robots::query()
    ->where("type = :type:")
    ->andWhere("year < 2000")
    ->bind(["type" => "mechanical"])
    ->order("name")
    ->execute();
```

La méthode statique `query()` retourne un objet *Phalcon\Mvc\ModelCriteria* qui plus favorable à l'autocomplétion des IDE.

Toutes les requêtes sont gérées en interne comme des requêtes *PHQL*. PHQL est un langage de haut niveau semblable au SQL et orienté objet. Ce langage dispose d'autre caractéristiques pour réaliser des requêtes comme des jointures

avec d'autres modèles, des regroupement, des agrégats, etc.

Enfin, il existe la méthode `findFirstBy<property-name>()`. Cette méthode étend la méthode `findFirst()` mentionnée plus tôt. Elle permet de réaliser rapidement une restitution depuis la table en exploitant le nom de la propriété elle-même et en transmettant en paramètre les données à rechercher sur cette colonne. Suivons un exemple en reprenant notre modèle Robots mentionné précédemment:

```
<?php
namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public $id;

    public $name;

    public $price;
}
```

Nous disposons de trois propriétés pour travailler avec: `$id`, `$name` et `$price`. Bon, mettons que vous voulez récupérer le premier enregistrement de la table avec le nom "Terminator". Ceci peut être écrit ainsi:

```
<?php
use Store\Toys\Robots;

$name = "Terminator";

$robot = Robots::findFirstByName($name);

if ($robot) {
    echo "Le premier robot avec le nom " . $name . " coûte " . $robot->price . ".";
} else {
    echo "Il n'existe pas dans la table de robot avec le nom " . $name . ".";
}
```

Notez que nous avons utilisé "Name" dans l'appel de la méthode et transmis la variable `$name` qui contient le nom que nous recherchons dans notre table. Notez également que lorsque nous trouvons une correspondance avec notre requête, toutes les autres propriétés nous sont également disponibles.

Jeux de résultat de modèles

Alors que `findFirst()` retourne directement une instance de la classe appelée (s'il existe des données à renvoyer), la méthode `find()` retourne un *Phalcon\Mvc\Model\Resultset\Simple*. C'est un objet qui encapsule toutes les fonctionnalités d'un jeu d'enregistrement comme le parcours, la recherche d'enregistrements spécifiques, le décompte, etc.

Ces objets sont plus puissants que les tableaux standards. Une des plus intéressantes caractéristiques de *Phalcon\Mvc\Model\Resultset* est qu'à n'importe quel moment il n'y a qu'un seul enregistrement en mémoire. Ceci facilite grandement la gestion de la mémoire surtout lorsqu'on travaille avec de grands volumes de données.

```
<?php
use Store\Toys\Robots;
```



```

// Récupère tous les robots
$robots = Robots::find();

// Parcours avec un foreach
foreach ($robots as $robot) {
    echo $robot->name, "\n";
}

// Parcours avec un while
$robots->rewind();

while ($robots->valid()) {
    $robot = $robots->current();

    echo $robot->name, "\n";

    $robots->next();
}

// Décompte du jeu de résultat
echo count($robots);

// Une autre façon de décompter le jeu de résultat
echo $robots->count();

// Déplace le curseur interne au troisième robot
$robots->seek(2);

$robot = $robots->current();

// Accède au robot par sa position dans le jeu de résultat
$robot = $robots[5];

// Vérifie qu'il existe un enregistrement à une certaine position
if (isset($robots[3])) {
    $robot = $robots[3];
}

// Prend le premier enregistrement dans le résultat
$robot = $robots->getFirst();

// Prend le dernier enregistrement
$robot = $robots->getLast();

```

Les jeux de résultat de Phalcon émulent les curseurs défilables. Vous pouvez prendre n'importe quel ligne juste d'après sa position, ou déplacer le pointeur interne à une position spécifique. Notez que certains SGBD ne supportent pas les curseurs défilables ce qui oblige à ré-exécuter la requête pour faire repartir le curseur depuis le début et d'obtenir l'enregistrement à la position demandée. De même, si un jeu de résultat doit être parcouru plusieurs fois, la requête sera exécutée d'autant de fois.

Comme le stockage en mémoire de volumineux résultats peut être gourmand en ressources, les jeux de résultat sont extraits de la base données par morceaux de 32 lignes, réduisant la nécessité de re-exécuter la requête dans la plupart des cas.

Notez que les jeux de résultats peuvent être sérialisés et stockés dans un cache serveur. *Phalcon\Cache* peut aider dans cette tâche. Cependant, la sérialisation de données oblige *Phalcon\Mvc\Model* à récupérer toutes les données de la base dans un tableau consommant ainsi plus de mémoire que nécessaire.

```
<?php

// Demande tous les enregistrements depuis le modèle
$parts = Parts::find();

// Stocke le jeu de résultat dans un fichier
file_put_contents(
    "cache.txt",
    serialize($parts)
);

// Récupère les données depuis un fichier
$parts = unserialize(
    file_get_contents("cache.txt")
);

// Parcourt les données
foreach ($parts as $part) {
    echo $part->id;
}
```

Filtrer les jeux d'enregistrement

La méthode la plus efficace pour filtrer les données est de définir des critères de recherche, les bases de données exploitant les index pour retourner les données plus rapidement. Phalcon vous permet de filtrer les données avec PHP en utilisant n'importe quelle ressource qui n'est pas disponible dans la base de données:

```
<?php

$customers = Customers::find();

$customers = $customers->filter(
    function ($customer) {
        // Retourne que les clients avec un e-mail valide
        if (filter_var($customer->email, FILTER_VALIDATE_EMAIL)) {
            return $customer;
        }
    }
);
```

Liaison de Paramètres

La liaison de paramètres est également supportée dans *Phalcon\Mvc\Model*. Vous êtes encouragés à utiliser cette méthode pour éliminer la possibilité que votre code soit le sujet d'attaques par injection SQL.

```
<?php

use Store\Toys\Robots;

// Interrogation de robots en liant les paramètres avec des marqueurs texte
// Paramètres dont les clés sont les même que les marqueurs
$robots = Robots::find(
    [
        "name = :name: AND type = :type:",
    ]
);
```

```

        "bind" => [
            "name" => "Robotina",
            "type" => "maid",
        ],
    ],
);

// Interrogation de robots en liant les paramètres avec les marqueurs numériques
$robots = Robots::find(
    [
        "name = ?1 AND type = ?2",
        "bind" => [
            1 => "Robotina",
            2 => "maid",
        ],
    ],
);

// Interrogation de robots avec à la fois des marqueurs numériques et textuels
// Paramètres dont les clés sont les même que les marqueurs
$robots = Robots::find(
    [
        "name = :name: AND type = ?1",
        "bind" => [
            "name" => "Robotina",
            1      => "maid",
        ],
    ],
);

```

En plaçant des marqueurs numériques, vous devez les écrire sous forme d'entier comme 1 ou 2. Dans ce cas "1" ou "2" sont considérés comme du texte et non des nombres, donc l'espace marqué ne peut pas être remplacé avec succès.

Les chaînes de caractères sont automatiquement échappées à l'aide de [PDO](#). Cette fonction prend en compte le jeu de caractères de la connexion, donc il est recommandé de définir le bon jeu de caractères dans les paramètres de la connexion ou bien dans la configuration de la base de données. Un mauvais jeu de caractères risque de produire des effets indésirables lors du stockage ou de la récupération des données.

De plus, vous pouvez définir le paramètre "bindTypes" qui permet de définir comment les paramètres sont liés en accord avec leurs types de données.

```

<?php

use Phalcon\Db\Column;
use Store\Toys\Robots;

// Paramètre lié
$parameters = [
    "name" => "Robotina",
    "year" => 2008,
];

// Conversion de type
$types = [
    "name" => Column::BIND_PARAM_STR,
    "year" => Column::BIND_PARAM_INT,
];

```

```
// Interrogation de robots en liant les paramètres à des marqueurs textuels
$robots = Robots::find(
    [
        "name = :name: AND year = :year:",
        "bind" => $parameters,
        "bindTypes" => $types,
    ]
);
```

Comme le type par défaut est `Phalcon\Db\Column::BIND_PARAM_STR`, il n'est pas nécessaire de préciser le paramètre "bindTypes" si toutes les colonnes sont de ce type.

Si vous attachez des tableaux aux paramètres liés, conservez à l'esprit que les index sont basés zéro:

```
<?php
use Store\Toys\Robots;

$array = ["a","b","c"]; // $array: [[0] => "a", [1] => "b", [2] => "c"]

unset($array[1]); // $array: [[0] => "a", [2] => "c"]

// Maintenant nous devons réindexer le tableau
$array = array_values($array); // $array: [[0] => "a", [1] => "c"]

$robots = Robots::find(
    [
        'letter IN ({letter:array})',
        'bind' => [
            'letter' => $array
        ]
    ]
);
```

La liaison de paramètres est disponible pour chaque méthode de requêtage tel que `find()` et `findFirst()` mais aussi les méthodes de calcul comme `count()`, `sum()`, `average()`, etc.

Si vous utilisez les "finders", les paramètres sont automatiquement liés:

```
<?php
use Store\Toys\Robots;

// Requête liant explicitement un paramètre
$robots = Robots::find(
    [
        "name = ?0",
        "bind" => [
            "Ultron",
        ],
    ]
);

// Requête liant implicitement un paramètre
$robots = Robots::findByName("Ultron");
```

Initialisation et Préparation d'Enregistrement récupéré

Il peut arriver qu'après avoir obtenu un enregistrement depuis la base de données, il soit nécessaire d'initialiser les données avant qu'elles ne soient utilisées dans le reste de l'application. Vous implémentez pour cela la méthode `afterFetch()` dans le modèle, cet événement sera exécuté juste après la création de l'instance et l'assignation des données:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public $id;

    public $name;

    public $status;

    public function beforeSave()
    {
        // Conversion du tableau en chaîne de caractères
        $this->status = join(",", $this->status);
    }

    public function afterFetch()
    {
        // Conversion de la chaîne de caractères en tableau
        $this->status = explode(",", $this->status);
    }

    public function afterSave()
    {
        // Conversion de la chaîne de caractères en tableau
        $this->status = explode(",", $this->status);
    }
}
```

Si vous utilisez les accesseurs et/ou les propriétés publiques, vous pouvez initialiser le champ une fois qu'il est accédé:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public $id;

    public $name;

    public $status;

    public function getStatus()
    {

```

```
        return explode(",", $this->status);
    }
}
```

Génération de calculs

Les calculs (ou les agrégations) sont des aides pour les fonctions couramment utilisées des SGBD comme COUNT, SUM, MAX, MIN ou AVG. *Phalcon\Mvc\Model* permet d'utiliser ces fonctions directement depuis les méthodes exposées.

Exemples de Count:

```
<?php

// Combien y-a-t'il d'employés ?
$rowcount = Employees::count();

// Combien de zones différentes sont assignées aux employés ?
$rowcount = Employees::count(
    [
        "distinct" => "area",
    ]
);

// Combien y-a-t'il d'employés dans le secteur "Testing" ?
$rowcount = Employees::count(
    "area = 'Testing'"
);

// Dénombre les employés en groupant le résultat par secteur
$group = Employees::count(
    [
        "group" => "area",
    ]
);
foreach ($group as $row) {
    echo "There are ", $row->rowcount, " in ", $row->area;
}

// Dénombre les employés en les groupant par secteur et ordonnant le résultat sur le
↳compte
$group = Employees::count(
    [
        "group" => "area",
        "order" => "rowcount",
    ]
);

// Évite les injections SQL avec des paramètres liés
$group = Employees::count(
    [
        "type > ?0",
        "bind" => [
            $type
        ],
    ]
);
```

Exemples de Sum:

```

<?php

// A combien s'élève le salaire de tous les employés ?
$total = Employees::sum(
    [
        "column" => "salary",
    ]
);

// A combien s'élève le salaire de tous les employés du secteur des ventes ?
$total = Employees::sum(
    [
        "column"      => "salary",
        "conditions" => "area = 'Sales'",
    ]
);

// Génère un regroupement des salaires par secteur
$group = Employees::sum(
    [
        "column" => "salary",
        "group"  => "area",
    ]
);
foreach ($group as $row) {
    echo "The sum of salaries of the ", $row->area, " is ", $row->sumatory;
}

// Génère un regroupement des salaires par secteur en ordonnant
// les salaires du plus grand au plus petit
$group = Employees::sum(
    [
        "column" => "salary",
        "group"  => "area",
        "order"  => "sumatory DESC",
    ]
);

// Évite les injections SQL avec des paramètres liés
$group = Employees::sum(
    [
        "conditions" => "area > ?0",
        "bind"       => [
            $area
        ],
    ]
);

```

Exemples d'Average:

```

<?php

// Quel est le salaire moyen de tous les employés ?
$average = Employees::average(
    [

```

```
        "column" => "salary",
    ]
);

// Quel est le salaire moyen de tous les employés du secteur des ventes ?
$average = Employees::average(
    [
        "column"      => "salary",
        "conditions" => "area = 'Sales'",
    ]
);

// Évite les injections SQL avec des paramètres liés
$average = Employees::average(
    [
        "column"      => "age",
        "conditions" => "area > ?0",
        "bind"        => [
            $area
        ],
    ]
);
```

Exemples Max/Min:

```
<?php

// Quel est l'âge le plus élevé de tous les employés ?
$age = Employees::maximum(
    [
        "column" => "age",
    ]
);

// Quel est l'âge le plus élevé de tous les employés du secteur des ventes ?
$age = Employees::maximum(
    [
        "column"      => "age",
        "conditions" => "area = 'Sales'",
    ]
);

// Quel est le salaire le plus bas de tous les employés ?
$salary = Employees::minimum(
    [
        "column" => "salary",
    ]
);
```

Création et Mise à jour d'Enregistrements

La méthode `Phalcon\Mvc\Model::save()` vous permet de créer ou de mettre à jour les enregistrement selon s'ils existent déjà dans la table associée au modèle. La méthode "save" est appelée en interne par les méthodes "create" et "update" de *Phalcon\Mvc\Model*. Pour que cela fonctionne comme prévu, il est nécessaire d'avoir correctement défini une clé primaire dans l'entité pour déterminer si un enregistrement should be updated or created.

De plus, la méthode exécute les validateurs associés, les clés étrangères virtuelle ainsi que les événements qui sont

définis dans le modèle:

```
<?php

use Store\Toys\Robots;

$robot = new Robots();

$robot->type = "mechanical";
$robot->name = "Astro Boy";
$robot->year = 1952;

if ($robot->save() === false) {
    echo "Umh, We can't store robots right now: \n";

    $messages = $robot->getMessages();

    foreach ($messages as $message) {
        echo $message, "\n";
    }
} else {
    echo "Great, a new robot was saved successfully!";
}
```

Un tableau peut être transmis à “save” pour éviter d’assigner chaque colonne manuellement. *Phalcon\Mvc\Model* va vérifier s’il existe des setters pour les colonnes indiquées dans le tableau en leur donnant priorité plutôt que d’affecter directement les valeurs des attributs:

```
<?php

use Store\Toys\Robots;

$robot = new Robots();

$robot->save(
    [
        "type" => "mechanical",
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);
```

Les valeurs qui sont assignées soit directement, soit à l’aide d’un tableau d’attributs, sont échappées et assainies selon le type de données relatif à l’attribut. Donc, n’ayez crainte des injections SQL lors de la transmission d’un tableau peu sûr:

```
<?php

use Store\Toys\Robots;

$robot = new Robots();

$robot->save($_POST);
```

Sans précaution, une affectation de masse pourrait permettre de définir la valeur à n’importe quelle colonne de la base de données. N’utilisez uniquement cette fonction que si vous voulez permettre à un utilisateur d’insérer ou de mettre à jour toutes les colonnes du modèle, même si ces champs ne sont pas soumis par le formulaire.

Vous pouvez ajouter un paramètre supplémentaire à “save” pour indiquer la liste blanche des champs qui seront pris en compte lors de l’assignation de masse:

```
<?php

use Store\Toys\Robots;

$robot = new Robots();

$robot->save(
    $_POST,
    [
        "name",
        "type",
    ]
);
```

Créer/Mettre à jour avec Confiance

Lorsqu’une application contient beaucoup d’accès concurrents, nous pourrions nous attendre à créer un enregistrement alors qu’il est mis à jour. Cela peut arriver en utilisant `Phalcon\Mvc\Model::save()` lors de la persistance des enregistrement en base. Pour être absolument certain que l’enregistrement soit créé ou mis à jour, nous pouvons remplacer l’appel de `save()` par `create()` ou `update()`:

```
<?php

use Store\Toys\Robots;

$robot = new Robots();

$robot->type = "mechanical";
$robot->name = "Astro Boy";
$robot->year = 1952;

// Cet enregistrement sera seulement créé
if ($robot->create() === false) {
    echo "Umh, We can't store robots right now: \n";

    $messages = $robot->getMessages();

    foreach ($messages as $message) {
        echo $message, "\n";
    }
} else {
    echo "Great, a new robot was created successfully!";
}
```

Les méthodes “create” et “update” acceptent également un tableau de valeurs en paramètre.

Suppression d’enregistrements

La méthode `Phalcon\Mvc\Model::delete()` permet de supprimer un enregistrement. Vous pouvez l’utiliser comme suit:

```
<?php

use Store\Toys\Robots;

$robot = Robots::findFirst(11);

if ($robot !== false) {
    if ($robot->delete() === false) {
        echo "Sorry, we can't delete the robot right now: \n";

        $messages = $robot->getMessages();

        foreach ($messages as $message) {
            echo $message, "\n";
        }
    } else {
        echo "The robot was deleted successfully!";
    }
}
```

Vous pouvez également supprimer plusieurs enregistrements en parcourant un jeu d'enregistrement avec foreach:

```
<?php

use Store\Toys\Robots;

$robots = Robots::find(
    "type = 'mechanical'"
);

foreach ($robots as $robot) {
    if ($robot->delete() === false) {
        echo "Sorry, we can't delete the robot right now: \n";

        $messages = $robot->getMessages();

        foreach ($messages as $message) {
            echo $message, "\n";
        }
    } else {
        echo "The robot was deleted successfully!";
    }
}
```

Les événements qui suivent servent à définir des règles métier qui seront exécutées lors d'une opération de suppression:

Opération	Nom	Opération stoppée ?	Explication
Deleting	beforeDelete	Oui	Lancé avant l'opération de suppression
Deleting	afterDelete	Non	Lancé après l'opération de suppression

Avec les événements ci-dessus vous pouvez également définir des règles métier dans les modèles:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;
```

```
class Robots extends Model
{
    public function beforeDelete()
    {
        if ($this->status === "A") {
            echo "The robot is active, it can't be deleted";

            return false;
        }

        return true;
    }
}
```

2.3.5 Relation de modèle

Relations entre modèles

Il existe quatre types de relations: un-à-un, un-à-plusieurs, plusieurs-à-un, plusieurs-à-plusieurs. La relation peut être unidirectionnelle ou bidirectionnelle, et chacune peut être simple (entre deux modèles) ou plus complexe (une combinaison de modèles). Le gestionnaire de modèles s'occupe des contraintes de clés étrangères pour ces relations, la définition de celles-ci contribue à l'intégrité référentielle aussi aisément que l'accès rapide aux enregistrements liés à un modèle. Grâce à la mise en œuvre de relations, il devient facile d'accéder aux données des modèles associés à chaque enregistrement d'une manière uniforme.

Relations Unidirectionnelles

Les relations unidirectionnelles sont celles qui sont dirigés d'un modèle vers un autre mais pas réciproquement.

Relations Bidirectionnelles

Les relations bidirectionnelles construisent des relations entre deux modèles, et chaque modèle établit une relation réciproque à l'autre.

Définition de relations

Dans Phalcon, les relations doivent être définies dans la méthode `initialize()` d'un modèle. Les méthodes `belongsTo()`, `hasOne()`, `hasMany()` et `hasManyToMany()` définissent la relation entre un ou plusieurs champs du modèle courant vers des champs d'un autre modèle. Chacune de ces méthodes requiert 3 paramètres: champs locaux, modèle référencé, champs référencés.

Méthode	Description
<code>hasMany</code>	Définit une relation 1-n
<code>hasOne</code>	Définit une relation 1-1
<code>belongsTo</code>	Définit une relation n-1
<code>hasManyToMany</code>	Définit une relation n-n

Le schéma suivant montre 3 tables dont les relations vont nous servir d'exemples sur les relations:

```

CREATE TABLE `robots` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(70) NOT NULL,
  `type` varchar(32) NOT NULL,
  `year` int(11) NOT NULL,
  PRIMARY KEY (`id`)
);

CREATE TABLE `robots_parts` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `robots_id` int(10) NOT NULL,
  `parts_id` int(10) NOT NULL,
  `created_at` DATE NOT NULL,
  PRIMARY KEY (`id`),
  KEY `robots_id` (`robots_id`),
  KEY `parts_id` (`parts_id`)
);

CREATE TABLE `parts` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(70) NOT NULL,
  PRIMARY KEY (`id`)
);

```

- Le modèle “Robots” a plusieurs “RobotsParts”.
- Le modèle “Parts” a plusieurs “RobotsParts”.
- Le modèle “RobotsParts” appartient aux modèles “Robots” et “Parts” dans une relation plusieurs-à-un.
- Le modèle “Robots” a une relation plusieurs-à-plusieurs vers “Parts” au travers de “RobotsParts”.

Regardez le diagramme EER pour mieux comprendre les relations:

Les modèles et leurs relations pourraient être implémentées comme suit:

```

<?php

namespace Store\Toys;

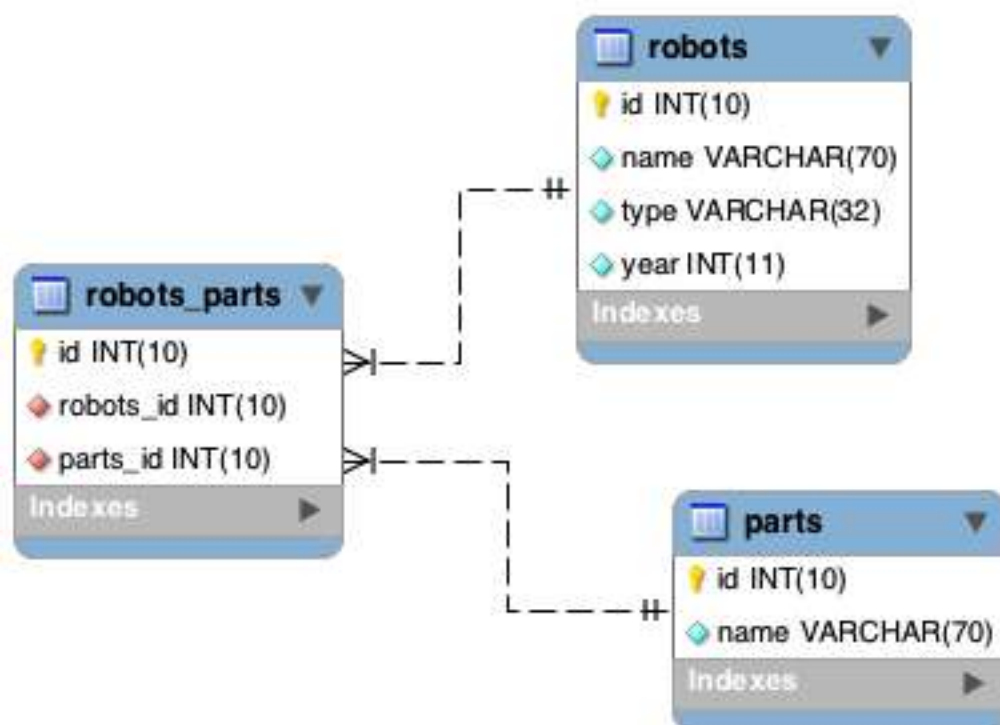
use Phalcon\Mvc\Model;

class Robots extends Model
{
    public $id;

    public $name;

    public function initialize()
    {
        $this->hasMany(
            "id",
            "RobotsParts",
            "robots_id"
        );
    }
}

```



```

<?php

use Phalcon\Mvc\Model;

class Parts extends Model
{
    public $id;

    public $name;

    public function initialize()
    {
        $this->hasMany(
            "id",
            "RobotsParts",
            "parts_id"
        );
    }
}

```

```

<?php

use Phalcon\Mvc\Model;

class RobotsParts extends Model
{
    public $id;

    public $robots_id;

    public $parts_id;

    public function initialize()
    {
        $this->belongsTo(
            "robots_id",
            "Store\\Toys\\Robots",
            "id"
        );

        $this->belongsTo(
            "parts_id",
            "Parts",
            "id"
        );
    }
}

```

Le premier paramètre indique le champ dans le modèle local impliqué dans la relation; le deuxième indique le nom du modèle référencé et le troisième le nom du champ dans le modèle référencé. Vous pouvez également utiliser des tableaux pour définir plusieurs champs dans la relation.

Les relations de type plusieurs à plusieurs nécessitent 3 modèles et la définition des attributs impliqués dans la relation:

```

<?php

namespace Store\Toys;

```

```
use Phalcon\Mvc\Model;

class Robots extends Model
{
    public $id;

    public $name;

    public function initialize()
    {
        $this->hasManyToMany(
            "id",
            "RobotsParts",
            "robots_id", "parts_id",
            "Parts",
            "id"
        );
    }
}
```

Profiter de l'avantage des relations

En définissant explicitement les relations entre modèles, il est aisé de trouver les enregistrements relatifs à un enregistrement particulier.

```
<?php

use Store\Toys\Robots;

$robot = Robots::findFirst(2);

foreach ($robot->robotsParts as $robotPart) {
    echo $robotPart->parts->name, "\n";
}
```

Phalcon utilise les méthodes magiques `__set`/`__get`/`__call` pour stocker ou récupérer les données relatives.

En accédant à un attribut du même nom que la relation, nous récupérons tous les enregistrements relatifs.

```
<?php

use Store\Toys\Robots;

$robot = Robots::findFirst();

/ Tous les enregistrements relatifs dans RobotsParts
$robotsParts = $robot->robotsParts; /
```

De même, vous pouvez utiliser un accesseur magique:

```
<?php

use Store\Toys\Robots;

$robot = Robots::findFirst();
```



```
// Tous les enregistrements relatifs dans RobotsParts
$robotsParts = $robot->getRobotsParts();

// Transmission de paramètres
$robotsParts = $robot->getRobotsParts(
    [
        "limit" => 5,
    ]
);
```

Si une méthode appelée porte le préfixe “get” alors *Phalcon\Mvc\Model* retournera un résultat `findFirst()/find()`. L'exemple suivant compare la récupération de résultats relatif avec et sans les méthodes magiques:

```
<?php

use Store\Toys\Robots;

$robot = Robots::findFirst(2);

// Le modèle Robots a une relation 1-n
// (hasMany) avec RobotsParts
$robotsParts = $robot->robotsParts;

// Seulement les "parts" qui répondent à la condition
$robotsParts = $robot->getRobotsParts(
    [
        "created_at" => ":date:",
        "bind" => [
            "date" => "2015-03-15"
        ]
    ]
);

$robotPart = RobotsParts::findFirst(1);

// le modèle RobotsParts a une relation n-1
// (belongsTo) avec Robots
$robot = $robotPart->robots;
```

Obtenir des enregistrements relatifs manuellement:

```
<?php

use Store\Toys\Robots;

$robot = Robots::findFirst(2);

// Le modèle Robots a une relation 1-n
// (hasMany) avec RobotsParts
$robotsParts = RobotsParts::find(
    [
        "robots_id" => ":id:",
        "bind" => [
            "id" => $robot->id,
        ]
    ]
);
```

```
// Seulement les "parts" qui répondent à la condition
$robotsParts = RobotsParts::find(
    [
        "robots_id = :id: AND created_at = :date:",
        "bind" => [
            "id" => $robot->id,
            "date" => "2015-03-15",
        ]
    ]
);

$robotPart = RobotsParts::findFirst(1);

// le modèle RobotsParts a une relation n-1
// (belongsTo) avec Robots
$robot = Robots::findFirst(
    [
        "id = :id:",
        "bind" => [
            "id" => $robotPart->robots_id,
        ]
    ]
);
```

Les méthodes “get” sont utilisées pour rechercher avec `find()` ou `findFirst()` les enregistrements associés selon le type de la relation:

Type	Description	Méthode implicite
Belongs-To	Retourne une instance du modèle de l’enregistrement directement associé	<code>findFirst</code>
Has-One	Retourne une instance du modèle de l’enregistrement directement associé	<code>findFirst</code>
Has-Many	Retourne une collection d’instances du modèle référencé	<code>find</code>
Has-Many-to-Many	Retourne une collection d’instances du modèle référencé. Réalise implicitement des “inner joins” avec les modèles concernés	(requête complexe)

Vous pouvez également utiliser le préfixe “count” pour retourner un entier qui indique le nombre d’enregistrements relatifs:

```
<?php
use Store\Toys\Robots;

$robot = Robots::findFirst(2);

echo "The robot has ", $robot->countRobotsParts(), " parts\n";
```

Alias dans les relations

Pour mieux comprendre comment les alias marchent, consultez l’exemple suivant:

La table “robots_similar” contient une fonction pour indiquer comment chaque robot est similaire à d’autres:

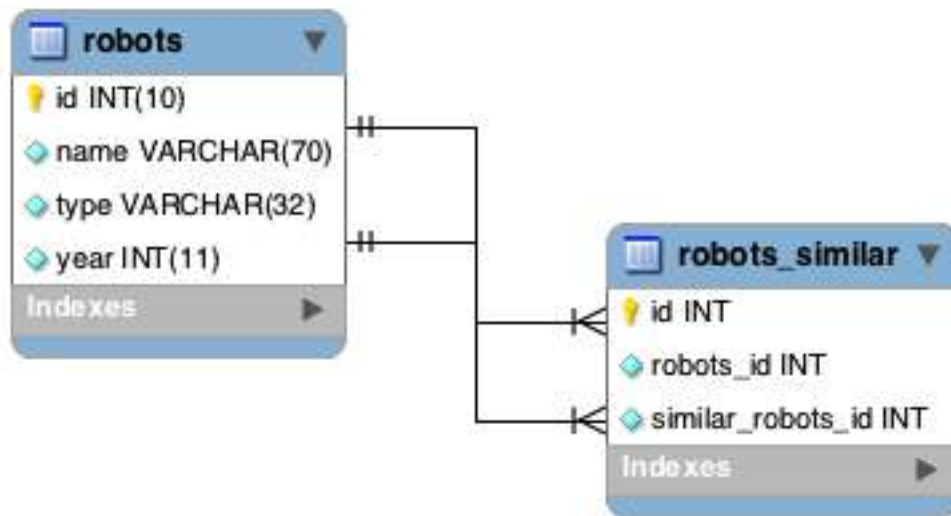
```
mysql> desc robots_similar;
+-----+-----+-----+-----+-----+-----+
| Field                | Type                | Null | Key | Default | Extra                |
```

```

+-----+-----+-----+-----+-----+-----+
| id      | int(10) unsigned | NO | PRI | NULL | auto_increment |
| robots_id | int(10) unsigned | NO | MUL | NULL |                 |
| similar_robots_id | int(10) unsigned | NO |     | NULL |                 |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

```

Les deux champs “robots_id” et “similar_robots_id” possèdent une relation vers le modèle Robots:



Un modèle qui définit une association de cette table et de ses relation est le suivant:

```

<?php

class RobotsSimilar extends Phalcon\Mvc\Model
{
    public function initialize()
    {
        $this->belongsTo(
            "robots_id",
            "Store\\Toys\\Robots",
            "id"
        );

        $this->belongsTo(
            "similar_robots_id",
            "Store\\Toys\\Robots",
            "id"
        );
    }
}

```

Tant que les deux relations pointent le même modèle (Robots), obtenir les enregistrements associés par les relations n’est pas très clair:

```

<?php

$robotsSimilar = RobotsSimilar::findFirst();

```

```
// Retourne l'enregistrement sous-jacent à la colonne robots_id
// Mais c'est aussi un belongsTo qui ne retourne qu'un seul enregistrement
// mais le nom "getRobots" semble indiquer qu'il en retourne plus d'un
$robot = $robotsSimilar->getRobots();

// Mais alors, comment récupérer l'enregistrement sous-jacent à la colonne similar_
↳ robots_id
// Si les deux relations possèdent le même nom ?
```

Les alias nous permettent de renommer chacune des relations, pour résoudre ce type de problèmes:

```
<?php

use Phalcon\Mvc\Model;

class RobotsSimilar extends Model
{
    public function initialize()
    {
        $this->belongsTo(
            "robots_id",
            "Store\\Toys\\Robots",
            "id",
            [
                "alias" => "Robot",
            ]
        );

        $this->belongsTo(
            "similar_robots_id",
            "Store\\Toys\\Robots",
            "id",
            [
                "alias" => "SimilarRobot",
            ]
        );
    }
}
```

Avec ces définitions d'alias nous pouvons récupérer aisément les enregistrements relatifs:

```
<?php

$robotsSimilar = RobotsSimilar::findFirst();

// Retourne l'enregistrement sous-jacent à la colonne (robots_id)
$robot = $robotsSimilar->getRobot();
$robot = $robotsSimilar->robot;

// Retourne l'enregistrement sous-jacent à la colonne (similar_robots_id)
$similarRobot = $robotsSimilar->getSimilarRobot();
$similarRobot = $robotsSimilar->similarRobot;
```

Accesseurs magiques contre méthodes explicites

La plupart des IDEs et des éditeurs ayant une autocomplétion ne peuvent pas déterminer le bon type avec les accesseurs magiques. Donc, au lieu d'utiliser les accesseurs magiques vous pouvez éventuellement définir explicitement ces méthodes avec leur docblock respectif aidant ainsi les IDE de produire une meilleur autocomplétion:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public $id;

    public $name;

    public function initialize()
    {
        $this->hasMany(
            "id",
            "RobotsParts",
            "robots_id"
        );
    }

    /**
     * Return the related "robots parts"
     *
     * @return \RobotsParts[]
     */
    public function getRobotsParts($parameters = null)
    {
        return $this->getRelated('RobotsParts', $parameters);
    }
}
```

Clés étrangères virtuelles

Par défaut, les relations n'agissent pas comme les clés étrangères des bases de données, ce qui fait que si vous tentez d'insérer ou de mettre à jour une valeur sans avoir une valeur valide dans le modèle référencé, Phalcon ne produira pas de message de validation. Vous pouvez modifier de comportement en ajoutant un quatrième paramètre lors de la définition de la relation.

Le modèle RobotsParts peut être modifié pour montrer cette capacité:

```
<?php

use Phalcon\Mvc\Model;

class RobotsParts extends Model
{
    public $id;

    public $robots_id;
```

```
public $parts_id;

public function initialize()
{
    $this->belongsTo(
        "robots_id",
        "Store\\Toys\\Robots",
        "id",
        [
            "foreignKey" => true
        ]
    );

    $this->belongsTo(
        "parts_id",
        "Parts",
        "id",
        [
            "foreignKey" => [
                "message" => "The part_id does not exist on the Parts model"
            ]
        ]
    );
}
```

Si vous altérez une relation `belongsTo()` pour qu'elle agisse comme une clé étrangère, elle vérifiera que les valeurs insérées ou mises à jour sur ces champs sont valides dans le modèle référencé. De même, si une relation `hasMany()/hasOne()` est altérée, elle vérifiera que les enregistrements ne peuvent pas être supprimés si l'enregistrement en question est utilisé dans le modèle référencé.

```
<?php

use Phalcon\Mvc\Model;

class Parts extends Model
{
    public function initialize()
    {
        $this->hasMany(
            "id",
            "RobotsParts",
            "parts_id",
            [
                "foreignKey" => [
                    "message" => "The part cannot be deleted because other robots are_
↪using it",
                ]
            ]
        );
    }
}
```

Une clé étrangère virtuelle peut être modifiée pour autoriser des valeurs nulles comme suit:

```
<?php

use Phalcon\Mvc\Model;
```

```

class RobotsParts extends Model
{
    public $id;

    public $robots_id;

    public $parts_id;

    public function initialize()
    {
        $this->belongsTo(
            "parts_id",
            "Parts",
            "id",
            [
                "foreignKey" => [
                    "allowNulls" => true,
                    "message"    => "The part_id does not exist on the Parts model",
                ]
            ]
        );
    }
}

```

Action en cascade ou Restrictions

Les relations qui agissent en tant que relation étrangère virtuelle restreignent par défaut la création, la suppression et la mise à jours d'enregistrements afin de maintenir l'intégrité des données:

```

<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;
use Phalcon\Mvc\Model\Relation;

class Robots extends Model
{
    public $id;

    public $name;

    public function initialize()
    {
        $this->hasMany(
            "id",
            "Parts",
            "robots_id",
            [
                "foreignKey" => [
                    "action" => Relation::ACTION_CASCADE,
                ]
            ]
        );
    }
}

```

```
}
```

Le code ci-dessus fait en sorte que les enregistrements référencés (parts) soient supprimés si l'enregistrement maître (robot) est supprimé.

Stockage des enregistrements relatifs

Les propriétés magiques peuvent être utilisées pour stocker les enregistrements et les propriétés associées;

```
<?php

// Creation d'un artiste
$artist = new Artists();

$artist->name      = "Shinichi Osawa";
$artist->country   = "Japan";

// Creation d'un album
$album = new Albums();

$album->name       = "The One";
$album->artist     = $artist; // Assign the artist
$album->year       = 2008;

// Sauvegarde les 2 enregistrements
$album->save();
```

Sauvegarder un enregistrement et ses enregistrements associés dans une relation has-many:

```
<?php

// Récupère un artiste existant
$artist = Artists::findFirst(
    "name = 'Shinichi Osawa'"
);

// Création d'un album
$album = new Albums();

$album->name      = "The One";
$album->artist    = $artist;

$songs = [];

// Création du premier morceau
$songs[0]        = new Songs();
$songs[0]->name   = "Star Guitar";
$songs[0]->duration = "5:54";

// Création du deuxième morceau
$songs[1]        = new Songs();
$songs[1]->name   = "Last Days";
$songs[1]->duration = "4:29";

// Assignment du tableau de morceaux
$album->songs = $songs;
```



```
// Enregistre l'album et ses morceaux
$album->save();
```

L'enregistrement simultané de l'album et de l'artiste implique l'utilisation implicite d'une transaction, ainsi s'il y a un problème lors de la sauvegarde des enregistrements associés, le parent ne sera pas sauvegardé non plus. Les messages sont renvoyés à l'utilisateur pour l'informer d'éventuelles erreurs.

Note: L'ajout d'entités relatives en surchargeant les méthodes suivantes n'est pas possible:

- `Phalcon\Mvc\Model::beforeSave()`
- `Phalcon\Mvc\Model::beforeCreate()`
- `Phalcon\Mvc\Model::beforeUpdate()`

Vous devez surcharger la méthode `Phalcon\Mvc\Model::save()` dans un modèle pour que cela fonctionne.

Opérations sur les jeux de résultat

Si un jeu de résultat est composé d'objets complets, le jeu de résultat est dans la capacité de réaliser des opérations sur les enregistrements d'une façon simple:

Mise à jour des enregistrements relatifs

Au lieu de faire ceci:

```
<?php

$parts = $robots->getParts();

foreach ($parts as $part) {
    $part->stock = 100;
    $part->updated_at = time();

    if ($part->update() === false) {
        $messages = $part->getMessages();

        foreach ($messages as $message) {
            echo $message;
        }

        break;
    }
}
```

vous pouvez faire cela:

```
<?php

$robots->getParts()->update(
    [
        "stock" => 100,
        "updated_at" => time(),
    ]
);
```

'update' accepte aussi des fonctions anonymes pour filter les enregistrements à mettre à jour:

```
<?php

$data = [
    "stock"      => 100,
    "updated_at" => time(),
];

// Mise à jour de tous les parts excepté ceux qui ont le type basic
$robots->getParts()->update(
    $data,
    function ($part) {
        if ($part->type === Part::TYPE_BASIC) {
            return false;
        }

        return true;
    }
);
```

Suppression des enregistrements relatifs

Au lieu de faire ceci:

```
<?php

$parts = $robots->getParts();

foreach ($parts as $part) {
    if ($part->delete() === false) {
        $messages = $part->getMessages();

        foreach ($messages as $message) {
            echo $message;
        }

        break;
    }
}
```

vous pouvez faire cela:

```
<?php

$robots->getParts()->delete();
```

`delete()` accepte aussi une fonction anonyme pour filtrer les enregistrements à supprimer:

```
<?php

// Supprime uniquement ceux dont le stock est positif ou nul
$robots->getParts()->delete(
    function ($part) {
        if ($part->stock < 0) {
            return false;
        }
    }
);
```

```

        return true;
    }
);

```

2.3.6 Événements et Modèles

Événements et Gestionnaire d'événements

Les modèles vous permettent d'écrire des événements qui seront générés lors de la réalisation d'une insertion/mise à jour(m.à.j.)/suppression. Il permettent de définir les règles métiers. Ce qui suit sont les événements supportés par *Phalcon\Mvc\Model* et leur ordre d'exécution:

Opération	Nom	Opération stoppée ? Explication	
insertion / m.à.j.	beforeValidation	Oui Est exécuté avant la validation des champs sur du texte nul ou vide ou bien des clés étrangères	
insertion	beforeValidationOnCreate	Oui	Est exécuté avant la validation des champs sur du texte nul ou vide ou bien des clés étrangères lors d'une opération d'insertion
m.à.j.	beforeValidationOnUpdate	Oui	Est exécuté avant la validation des champs sur du texte nul ou vide ou bien des clés étrangères lors d'une opération de mise à jour
insertion / m.à.j.	onValidationFails	Oui (systématiquement)	Est exécuté lors de l'échec d'une validation d'intégrité
insertion	afterValidationOnCreate	Oui	Est exécuté après la validation des champs sur du texte nul ou vide ou bien des clés étrangères lors d'une opération d'insertion
m.à.j.	afterValidationOnUpdate	Oui	Est exécuté après la validation des champs sur du texte nul ou vide ou bien des clés étrangères lors d'une opération de mise à jour
insertion / m.à.j.	afterValidation	Oui	Est exécuté après la validation des champs sur du texte nul ou vide ou bien des clés étrangères
insertion / m.à.j.	beforeSave	Oui	Lancé avant l'opération requise sur le SGBD
m.à.j.	beforeUpdate	Oui	Lancé avant l'opération de mise à jour requise sur le SGBD
insertion	beforeCreate	Oui	Lancé avant l'opération d'insertion requise sur le SGBD
m.à.j.	afterUpdate	Non	Lancé après l'opération de mise à jour requise sur le SGBD
insertion	afterCreate	Non	Lancé après l'opération d'insertion requise sur le SGBD
insertion / m.à.j.	afterSave	Non	Lancé après l'opération requise sur le SGBD

Mise en œuvre d'événements dans la classe du Modèle

La façon la plus facile pour faire en sorte qu'un modèle réagisse aux événement est de réaliser dans la classe une méthode du même nom que l'événement:

```

<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function beforeValidationOnCreate()
    {

```

```
        echo "Ceci est exécuté avant la création d'un Robot !";
    }
}
```

Les événements peuvent être utiles pour assigner des valeurs avant la réalisation d'une opération comme par exemple:

```
<?php
use Phalcon\Mvc\Model;

class Products extends Model
{
    public function beforeCreate()
    {
        // Établir la date de création
        $this->created_at = date("Y-m-d H:i:s");
    }

    public function beforeUpdate()
    {
        // Établir la date de modification
        $this->modified_in = date("Y-m-d H:i:s");
    }
}
```

Utilisation d'un Gestionnaire d'Événements personnalisé

De plus, ce composant est intégré dans *Phalcon\Events\Manager*, ce qui signifie que nous pouvons créer des écouteurs qui s'exécutent lors du déclenchement d'un événement.

```
<?php
namespace Store\Toys;

use Phalcon\Mvc\Model;
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;

class Robots extends Model
{
    public function initialize()
    {
        $eventsManager = new EventsManager();

        // Attache une fonction anonyme pour écouter les événements de "model"
        $eventsManager->attach(
            "model:beforeSave",
            function (Event $event, $robot) {
                if ($robot->name === "Scooby Doo") {
                    echo "Scooby Doo isn't a robot!";

                    return false;
                }

                return true;
            }
        );
    }
}
```

```

    );

    // Attache le gestionnaire d'événement à l'événement
    $this->setEventManager($eventsManager);
}
}

```

Dans l'exemple précédent, le Gestionnaire d'Événements agit comme un pont entre l'objet et l'écouteur (la fonction anonyme). Les événements fuseront vers les écouteurs lors de la sauvegarde de 'robots':

```

<?php

use Store\Toys\Robots;

$robot = new Robots();

$robot->name = "Scooby Doo";
$robot->year = 1969;

$robot->save();

```

Si vous voulez que tous les objets créés dans votre application utilisent le même EventsManager, vous devez alors l'assigner au Gestionnaire de Modèles:

```

<?php

use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;

// Inscription du service "modelsManager"
$di->setShared(
    "modelsManager",
    function () {
        $eventsManager = new EventsManager();

        // Attache une fonction anonyme en tant qu'écouteur pour les événements de "model"
        $eventsManager->attach(
            "model:beforeSave",
            function (Event $event, $model) {
                // Capture les événements produits par le modèle
                ↪ "Robots"

                if (get_class($model) === "Store\\Toys\\Robots") {
                    if ($model->name === "Scooby Doo") {
                        echo "Scooby Doo isn't a robot!";

                        return false;
                    }
                }

                return true;
            }
        );

        // Établissement d'un EventsManager par défaut
        $modelsManager = new ModelsManager();

        $modelsManager->setEventManager($eventsManager);
    }
);

```

```
        return $modelsManager;
    }
);
```

Si un écouteur retourne “faux” alors ceci interrompt l’opération en cours d’exécution.

Journalisation des instructions SQL de bas niveau

Lorsqu’on utilise un composant de haut niveau d’abstraction tel que *Phalcon\Mvc\Model* pour accéder aux données, il devient difficile de savoir quelles sont les instructions qui sont finalement envoyées au SGBD. *Phalcon\Mvc\Model* est supporté en interne par *Phalcon\Db*. *Phalcon\Logger* interagit avec *Phalcon\Db*, fournissant des capacités de journalisation sur la couche d’abstraction de la base de données, ce qui nous permet de journaliser les instructions quand elles surviennent.

```
<?php

use Phalcon\Logger;
use Phalcon\Events\Manager;
use Phalcon\Logger\Adapter\File as FileLogger;
use Phalcon\Db\Adapter\Pdo\Mysql as Connection;

$di->set(
    "db",
    function () {
        $eventsManager = new EventsManager();

        $logger = new FileLogger("app/logs/debug.log");

        // Ecoute tous les événements de la base de données
        $eventsManager->attach(
            "db:beforeQuery",
            function ($event, $connection) use ($logger) {
                $logger->log(
                    $connection->getSQLStatement(),
                    Logger::INFO
                );
            }
        );

        $connection = new Connection(
            [
                "host"      => "localhost",
                "username" => "root",
                "password" => "secret",
                "dbname"    => "invo",
            ]
        );

        // Assigne l'eventsManager à l'instance de l'adaptateur de bd
        $connection->setEventsManager($eventsManager);

        return $connection;
    }
);
```

Comme les modèles utilisent la connexion par défaut à la base de données, toutes les instructions SQL envoyées au SGBD seront journalisées dans un fichier:

```
<?php

use Store\Toys\Robots;

$robot = new Robots();

$robot->name      = "Robby the Robot";
$robot->created_at = "1956-07-21";

if ($robot->save() === false) {
    echo "Cannot save robot";
}
```

D'après le code ci-dessus, le fichier *app/logs/db.log* doit contenir quelque chose du genre:

```
[Mon, 30 Apr 12 13:47:18 -0500][DEBUG][Resource Id #77] INSERT INTO robots
(name, created_at) VALUES ('Robby the Robot', '1956-07-21')
```

Profilage des instructions SQL

Grâce à *Phalcon\Db*, le composant sous-jacent de *Phalcon\Mvc\Model*, il est possible de profiler les instructions SQL générées par l'ORM afin d'analyser la performance d'opérations en base de données. Avec ceci vous pouvez diagnostiquer des problèmes de performance et découvrir les goulots d'étranglement.

```
<?php

use Phalcon\Db\Profiler as ProfilerDb;
use Phalcon\Events\Manager as EventsManager;
use Phalcon\Db\Adapter\Pdo\Mysql as MysqlPdo;

$di->set(
    "profiler",
    function () {
        return new ProfilerDb();
    },
    true
);

$di->set(
    "db",
    function () use ($di) {
        $eventsManager = new EventsManager();

        // Récupère une instance partagée de DbProfiler
        $profiler = $di->getProfiler();

        // Ecoute tous les événements de base de données
        $eventsManager->attach(
            "db",
            function ($event, $connection) use ($profiler) {
                if ($event->getType() === "beforeQuery") {
                    $profiler->startProfile(
                        $connection->getSQLStatement()
                    );
                }
            }
        );
    }
);
```

```
        if ($event->getType() === "afterQuery") {
            $profiler->stopProfile();
        }
    }
);

$connection = new MysqlPdo(
    [
        "host"      => "localhost",
        "username"  => "root",
        "password"  => "secret",
        "dbname"    => "invo",
    ]
);

// Assigne l'eventsManager à l'instance de l'adaptateur de bd
$connection->setEventsManager($eventsManager);

return $connection;
}
);
```

Profilons quelques requêtes:

```
<?php

use Store\Toys\Robots;

// Envoi de quelques instructions SQL à la base
Robots::find();

Robots::find(
    [
        "order" => "name",
    ]
);

Robots::find(
    [
        "limit" => 30,
    ]
);

// Récupère les profils générés par le profileur
$profiles = $di->get("profiler")->getProfiles();

foreach ($profiles as $profile) {
    echo "SQL Statement: ", $profile->getSQLStatement(), "\n";
    echo "Start Time: ", $profile->getInitialTime(), "\n";
    echo "Final Time: ", $profile->getFinalTime(), "\n";
    echo "Total Elapsed Time: ", $profile->getTotalElapsedSeconds(), "\n";
}
```

Chaque profil généré contient le temps en millisecondes que chaque instruction prend pour compléter ainsi l'instruction SQL générée.

2.3.7 Model Behaviors

Behaviors are shared conducts that several models may adopt in order to re-use code, the ORM provides an API to implement behaviors in your models. Also, you can use the events and callbacks as seen before as an alternative to implement Behaviors with more freedom.

A behavior must be added in the model initializer, a model can have zero or more behaviors:

```
<?php

use Phalcon\Mvc\Model;
use Phalcon\Mvc\Model\Behavior\Timestampable;

class Users extends Model
{
    public $id;

    public $name;

    public $created_at;

    public function initialize()
    {
        $this->addBehavior(
            new Timestampable(
                [
                    "beforeCreate" => [
                        "field" => "created_at",
                        "format" => "Y-m-d",
                    ]
                ]
            )
        );
    }
}
```

The following built-in behaviors are provided by the framework:

Name	Description
Timestampable	Allows to automatically update a model's attribute saving the datetime when a record is created or updated
SoftDelete	Instead of permanently delete a record it marks the record as deleted changing the value of a flag column

Timestampable

This behavior receives an array of options, the first level key must be an event name indicating when the column must be assigned:

```
<?php

use Phalcon\Mvc\Model\Behavior\Timestampable;

public function initialize()
{
    $this->addBehavior(
        new Timestampable(
```

```
        [
            "beforeCreate" => [
                "field" => "created_at",
                "format" => "Y-m-d",
            ]
        ]
    )
};
}
```

Each event can have its own options, ‘field’ is the name of the column that must be updated, if ‘format’ is a string it will be used as format of the PHP’s function `date`, format can also be an anonymous function providing you the free to generate any kind timestamp:

```
<?php

use DateTime;
use DateTimeZone;
use Phalcon\Mvc\Model\Behavior\Timestampable;

public function initialize()
{
    $this->addBehavior(
        new Timestampable(
            [
                "beforeCreate" => [
                    "field" => "created_at",
                    "format" => function () {
                        $datetime = new DateTime(
                            new DateTimeZone("Europe/Stockholm")
                        );

                        return $datetime->format("Y-m-d H:i:sP");
                    }
                ]
            ]
        )
    );
}
```

If the option ‘format’ is omitted a timestamp using the PHP’s function `time`, will be used.

SoftDelete

This behavior can be used in the following way:

```
<?php

use Phalcon\Mvc\Model;
use Phalcon\Mvc\Model\Behavior\SoftDelete;

class Users extends Model
{
    const DELETED = "D";

    const NOT_DELETED = "N";
}
```

```

public $id;

public $name;

public $status;

public function initialize()
{
    $this->addBehavior(
        new SoftDelete(
            [
                "field" => "status",
                "value" => Users::DELETED,
            ]
        )
    );
}
}

```

This behavior accepts two options: ‘field’ and ‘value’, ‘field’ determines what field must be updated and ‘value’ the value to be deleted. Let’s pretend the table ‘users’ has the following data:

```
mysql> select * from users;
+----+-----+-----+
| id | name   | status |
+----+-----+-----+
|  1 | Lana   | N      |
|  2 | Brandon | N      |
+----+-----+-----+
2 rows in set (0.00 sec)
```

If we delete any of the two records the status will be updated instead of delete the record:

```
<?php
Users::findFirst(2)->delete();
```

The operation will result in the following data in the table:

```
mysql> select * from users;
+----+-----+-----+
| id | name   | status |
+----+-----+-----+
|  1 | Lana   | N      |
|  2 | Brandon | D      |
+----+-----+-----+
2 rows in set (0.01 sec)
```

Note that you need to specify the deleted condition in your queries to effectively ignore them as deleted records, this behavior doesn’t support that.

Creating your own behaviors

The ORM provides an API to create your own behaviors. A behavior must be a class implementing the *Phalcon\Mvc\Model\BehaviorInterface*. Also, *Phalcon\Mvc\Model\Behavior* provides most of the methods needed to ease the implementation of behaviors.

The following behavior is an example, it implements the Blameable behavior which helps identify the user that is performed operations over a model:

```
<?php

use Phalcon\Mvc\Model\Behavior;
use Phalcon\Mvc\Model\BehaviorInterface;

class Blameable extends Behavior implements BehaviorInterface
{
    public function notify($eventType, $model)
    {
        switch ($eventType) {

            case "afterCreate":
            case "afterDelete":
            case "afterUpdate":

                $userName = // ... get the current user from session

                // Store in a log the username, event type and primary key
                file_put_contents(
                    "logs/blamable-log.txt",
                    $userName . " " . $eventType . " " . $model->id
                );

                break;

            default:
                /* ignore the rest of events */
        }
    }
}
```

The former is a very simple behavior, but it illustrates how to create a behavior, now let's add this behavior to a model:

```
<?php

use Phalcon\Mvc\Model;

class Profiles extends Model
{
    public function initialize()
    {
        $this->addBehavior(
            new Blameable()
        );
    }
}
```

A behavior is also capable of intercepting missing methods on your models:

```
<?php

use Phalcon\Tag;
use Phalcon\Mvc\Model\Behavior;
use Phalcon\Mvc\Model\BehaviorInterface;

class Sluggable extends Behavior implements BehaviorInterface
{
    public function missingMethod($model, $method, $arguments = [])
    {
        // If the method is 'getSlug' convert the title
        if ($method === "getSlug") {
            return Tag::friendlyTitle($model->title);
        }
    }
}
```

Call that method on a model that implements Sluggable returns a SEO friendly title:

```
<?php

$title = $post->getSlug();
```

Using Traits as behaviors

Starting from PHP 5.4 you can use [Traits](#) to re-use code in your classes, this is another way to implement custom behaviors. The following trait implements a simple version of the Timestampable behavior:

```
<?php

trait MyTimestampable
{
    public function beforeCreate()
    {
        $this->created_at = date("r");
    }

    public function beforeUpdate()
    {
        $this->updated_at = date("r");
    }
}
```

Then you can use it in your model as follows:

```
<?php

use Phalcon\Mvc\Model;

class Products extends Model
{
    use MyTimestampable;
}
```

2.3.8 Models Metadata

To speed up development *Phalcon\Mvc\Model* helps you to query fields and constraints from tables related to models. To achieve this, *Phalcon\Mvc\Model\MetaData* is available to manage and cache table metadata.

Sometimes it is necessary to get those attributes when working with models. You can get a metadata instance as follows:

```
<?php

$robot = new Robots();

// Get Phalcon\Mvc\Model\MetaData instance
$metadata = $robot->getModelsMetaData();

// Get robots fields names
$attributes = $metadata->getAttributes($robot);
print_r($attributes);

// Get robots fields data types
$dataTypes = $metadata->getDataTypes($robot);
print_r($dataTypes);
```

Caching Metadata

Once the application is in a production stage, it is not necessary to query the metadata of the table from the database system each time you use the table. This could be done caching the metadata using any of the following adapters:

Adapter	Description	API
Memory	This adapter is the default. The metadata is cached only during the request. When the request is completed, the metadata are released as part of the normal memory of the request. This adapter is perfect when the application is in development so as to refresh the metadata in each request containing the new and/or modified fields.	<i>Phalcon\Mvc\Model\MetaData\Memory</i>
Session	This adapter stores metadata in the <code>\$_SESSION</code> superglobal. This adapter is recommended only when the application is actually using a small number of models. The metadata are refreshed every time a new session starts. This also requires the use of <code>session_start()</code> to start the session before using any models.	<i>Phalcon\Mvc\Model\MetaData\Session</i>
Apc	This adapter uses the Alternative PHP Cache (APC) to store the table metadata. You can specify the lifetime of the metadata with options. This is the most recommended way to store metadata when the application is in production stage.	<i>Phalcon\Mvc\Model\MetaData\Apc</i>
XCache	This adapter uses XCache to store the table metadata. You can specify the lifetime of the metadata with options. This is the most recommended way to store metadata when the application is in production stage.	<i>Phalcon\Mvc\Model\MetaData\Xcache</i>
Files	This adapter uses plain files to store metadata. By using this adapter the disk-reading is increased but the database access is reduced.	<i>Phalcon\Mvc\Model\MetaData\Files</i>

As other ORM's dependencies, the metadata manager is requested from the services container:

```
<?php

use Phalcon\Mvc\Model\MetaData\Apc as ApcMetaData;

$di["modelsMetadata"] = function () {
    // Create a metadata manager with APC
    $metadata = new ApcMetaData(
```

```

        [
            "lifetime" => 86400,
            "prefix"   => "my-prefix",
        ]
    );

    return $metadata;
};

```

Metadata Strategies

As mentioned above the default strategy to obtain the model's metadata is database introspection. In this strategy, the information schema is used to know the fields in a table, its primary key, nullable fields, data types, etc.

You can change the default metadata introspection in the following way:

```

<?php

use Phalcon\Mvc\Model\MetaData\Apc as ApcMetadata;

$di["modelsMetadata"] = function () {
    // Instantiate a metadata adapter
    $metadata = new ApcMetadata(
        [
            "lifetime" => 86400,
            "prefix"   => "my-prefix",
        ]
    );

    // Set a custom metadata introspection strategy
    $metadata->setStrategy(
        new MyIntrospectionStrategy()
    );

    return $metadata;
};

```

Database Introspection Strategy

This strategy doesn't require any customization and is implicitly used by all the metadata adapters.

Annotations Strategy

This strategy makes use of *annotations* to describe the columns in a model:

```

<?php

use Phalcon\Mvc\Model;

class Robots extends Model
{
    /**
     * @Primary

```

```
* @Identity
* @Column(type="integer", nullable=false)
*/
public $id;

/**
 * @Column(type="string", length=70, nullable=false)
 */
public $name;

/**
 * @Column(type="string", length=32, nullable=false)
 */
public $type;

/**
 * @Column(type="integer", nullable=false)
 */
public $year;
}
```

Annotations must be placed in properties that are mapped to columns in the mapped source. Properties without the @Column annotation are handled as simple class attributes.

The following annotations are supported:

Name	Description
Primary	Mark the field as part of the table's primary key
Identity	The field is an auto_increment/serial column
Column	This marks an attribute as a mapped column

The annotation @Column supports the following parameters:

Name	Description
type	The column's type (string, integer, decimal, boolean)
length	The column's length if any
nullable	Set whether the column accepts null values or not

The annotations strategy could be set up this way:

```
<?php

use Phalcon\Mvc\Model\MetaData\Apc as ApcMetaData;
use Phalcon\Mvc\Model\MetaData\Strategy\Annotations as StrategyAnnotations;

$di["modelsMetadata"] = function () {
    // Instantiate a metadata adapter
    $metadata = new ApcMetaData(
        [
            "lifetime" => 86400,
            "prefix"   => "my-prefix",
        ]
    );

    // Set a custom metadata database introspection
    $metadata->setStrategy(
        new StrategyAnnotations()
    );
};
```



```

    return $metadata;
};

```

Manual Metadata

Phalcon can obtain the metadata for each model automatically without the developer must set them manually using any of the introspection strategies presented above.

The developer also has the option of define the metadata manually. This strategy overrides any strategy set in the metadata manager. New columns added/modified/removed to/from the mapped table must be added/modified/removed also for everything to work properly.

The following example shows how to define the metadata manually:

```

<?php

use Phalcon\Mvc\Model;
use Phalcon\Db\Column;
use Phalcon\Mvc\Model\MetaData;

class Robots extends Model
{
    public function metaData()
    {
        return array(
            // Every column in the mapped table
            MetaData::MODELS_ATTRIBUTES => [
                "id",
                "name",
                "type",
                "year",
            ],

            // Every column part of the primary key
            MetaData::MODELS_PRIMARY_KEY => [
                "id",
            ],

            // Every column that isn't part of the primary key
            MetaData::MODELS_NON_PRIMARY_KEY => [
                "name",
                "type",
                "year",
            ],

            // Every column that doesn't allows null values
            MetaData::MODELS_NOT_NULL => [
                "id",
                "name",
                "type",
            ],

            // Every column and their data types
            MetaData::MODELS_DATA_TYPES => [
                "id" => Column::TYPE_INTEGER,
                "name" => Column::TYPE_VARCHAR,
                "type" => Column::TYPE_VARCHAR,
            ],
        );
    }
}

```

```
        "year" => Column::TYPE_INTEGER,
    ],

    // The columns that have numeric data types
    MetaData::MODELS_DATA_TYPES_NUMERIC => [
        "id"    => true,
        "year" => true,
    ],

    // The identity column, use boolean false if the model doesn't have
    // an identity column
    MetaData::MODELS_IDENTITY_COLUMN => "id",

    // How every column must be bound/casted
    MetaData::MODELS_DATA_TYPES_BIND => [
        "id"    => Column::BIND_PARAM_INT,
        "name" => Column::BIND_PARAM_STR,
        "type" => Column::BIND_PARAM_STR,
        "year" => Column::BIND_PARAM_INT,
    ],

    // Fields that must be ignored from INSERT SQL statements
    MetaData::MODELS_AUTOMATIC_DEFAULT_INSERT => [
        "year" => true,
    ],

    // Fields that must be ignored from UPDATE SQL statements
    MetaData::MODELS_AUTOMATIC_DEFAULT_UPDATE => [
        "year" => true,
    ],

    // Default values for columns
    MetaData::MODELS_DEFAULT_VALUES => [
        "year" => "2015",
    ],

    // Fields that allow empty strings
    MetaData::MODELS_EMPTY_STRING_VALUES => [
        "name" => true,
    ],
    );
}
```

2.3.9 Model Transactions

When a process performs multiple database operations, it might be important that each step is completed successfully so that data integrity can be maintained. Transactions offer the ability to ensure that all database operations have been executed successfully before the data is committed to the database.

Transactions in Phalcon allow you to commit all operations if they were executed successfully or rollback all operations if something went wrong.

Manual Transactions

If an application only uses one connection and the transactions aren't very complex, a transaction can be created by just moving the current connection into transaction mode and then commit or rollback the operation whether it is successful or not:

```
<?php

use Phalcon\Mvc\Controller;

class RobotsController extends Controller
{
    public function saveAction()
    {
        // Start a transaction
        $this->db->begin();

        $robot = new Robots();

        $robot->name      = "WALL-E";
        $robot->created_at = date("Y-m-d");

        // The model failed to save, so rollback the transaction
        if ($robot->save() === false) {
            $this->db->rollback();
            return;
        }

        $robotPart = new RobotParts();

        $robotPart->robots_id = $robot->id;
        $robotPart->type      = "head";

        // The model failed to save, so rollback the transaction
        if ($robotPart->save() === false) {
            $this->db->rollback();

            return;
        }

        // Commit the transaction
        $this->db->commit();
    }
}
```

Implicit Transactions

Existing relationships can be used to store records and their related instances, this kind of operation implicitly creates a transaction to ensure that data is correctly stored:

```
<?php

$robotPart = new RobotParts();

$robotPart->type = "head";
```

```
$robot = new Robots();

$robot->name      = "WALL·E";
$robot->created_at = date("Y-m-d");
$robot->robotPart = $robotPart;

// Creates an implicit transaction to store both records
$robot->save();
```

Isolated Transactions

Isolated transactions are executed in a new connection ensuring that all the generated SQL, virtual foreign key checks and business rules are isolated from the main connection. This kind of transaction requires a transaction manager that globally manages each transaction created ensuring that they are correctly rolled back/committed before ending the request:

```
<?php

use Phalcon\Mvc\Model\Transaction\Failed as TxFailed;
use Phalcon\Mvc\Model\Transaction\Manager as TxManager;

try {
    // Create a transaction manager
    $manager = new TxManager();

    // Request a transaction
    $transaction = $manager->get();

    $robot = new Robots();

    $robot->setTransaction($transaction);

    $robot->name      = "WALL·E";
    $robot->created_at = date("Y-m-d");

    if ($robot->save() === false) {
        $transaction->rollback(
            "Cannot save robot"
        );
    }

    $robotPart = new RobotParts();

    $robotPart->setTransaction($transaction);

    $robotPart->robots_id = $robot->id;
    $robotPart->type       = "head";

    if ($robotPart->save() === false) {
        $transaction->rollback(
            "Cannot save robot part"
        );
    }

    // Everything's gone fine, let's commit the transaction
```

```

        $transaction->commit();
    } catch (TxFailed $e) {
        echo "Failed, reason: ", $e->getMessage();
    }
}

```

Transactions can be used to delete many records in a consistent way:

```

<?php

use Phalcon\Mvc\Model\Transaction\Failed as TxFailed;
use Phalcon\Mvc\Model\Transaction\Manager as TxManager;

try {
    // Create a transaction manager
    $manager = new TxManager();

    // Request a transaction
    $transaction = $manager->get();

    // Get the robots to be deleted
    $robots = Robots::find(
        "type = 'mechanical'"
    );

    foreach ($robots as $robot) {
        $robot->setTransaction($transaction);

        // Something's gone wrong, we should rollback the transaction
        if ($robot->delete() === false) {
            $messages = $robot->getMessages();

            foreach ($messages as $message) {
                $transaction->rollback(
                    $message->getMessage()
                );
            }
        }
    }

    // Everything's gone fine, let's commit the transaction
    $transaction->commit();

    echo "Robots were deleted successfully!";
} catch (TxFailed $e) {
    echo "Failed, reason: ", $e->getMessage();
}

```

Transactions are reused no matter where the transaction object is retrieved. A new transaction is generated only when a `commit()` or `rollback()` is performed. You can use the service container to create the global transaction manager for the entire application:

```

<?php

use Phalcon\Mvc\Model\Transaction\Manager as TransactionManager

$di->setShared(
    "transactions",
    function () {

```

```
        return new TransactionManager();
    }
);
```

Then access it from a controller or view:

```
<?php
use Phalcon\Mvc\Controller;

class ProductsController extends Controller
{
    public function saveAction()
    {
        // Obtain the TransactionsManager from the services container
        $manager = $this->di->getTransactions();

        // Or
        $manager = $this->transactions;

        // Request a transaction
        $transaction = $manager->get();

        // ...
    }
}
```

While a transaction is active, the transaction manager will always return the same transaction across the application.

2.3.10 Validation de modèles

Validation de l'intégrité des données

Phalcon\Mvc\Model fournit plusieurs événements pour valider les données et rédiger les règles métier. L'événement spécial "validation" nous permet d'appeler des validateurs prédéfinis sur l'enregistrement. Phalcon expose quelques validateurs déjà prêts à l'emploi à ce niveau de validation.

L'exemple suivant montre comment l'utiliser:

```
<?php
namespace Store\Toys;

use Phalcon\Mvc\Model;
use Phalcon\Validation;
use Phalcon\Validation\Validator\Uniqueness;
use Phalcon\Validation\Validator\InclusionIn;

class Robots extends Model
{
    public function validation()
    {
        $validator = new Validation();

        $validator->add(
            "type",
```

```

        new InclusionIn(
            [
                "domain" => [
                    "Mechanical",
                    "Virtual",
                ]
            ]
        )
    );

    $validator->add(
        "name",
        new Uniqueness(
            [
                "message" => "The robot name must be unique",
            ]
        )
    );

    return $this->validate($validator);
}
}

```

L'exemple précédent effectue une validation en utilisant le validateur prédéfini "InclusionIn". Il vérifie que la valeur du champ "type" soit dans la liste de "domain". Si la valeur n'est pas incluse dans la méthode alors le validateur échoue et retourne "faux". Les validateurs prédéfinis qui suivent sont disponibles:

Pour plus d'information sur les validateurs, consultez [Validation documentation](#).

L'idée derrière la création de validateurs est de les rendre réutilisables entre plusieurs modèles. Un validateur peut être aussi simple que:

```

<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;
use Phalcon\Mvc\Model\Message;

class Robots extends Model
{
    public function validation()
    {
        if ($this->type === "Old") {
            $message = new Message(
                "Sorry, old robots are not allowed anymore",
                "type",
                "MyType"
            );

            $this->appendMessage($message);

            return false;
        }

        return true;
    }
}

```

Messages de validation

Phalcon\Mvc\Model dispose d'un sous-système de messagerie qui fournit un moyen flexible de sortir ou stocker des messages de validation générés pendant les processus d'insertion/mise à jour.

Chaque message consiste en une instance de la classe *Phalcon\Mvc\Model\Message*. L'ensemble de messages générés peut être récupéré avec la méthode `getMessages()`. Chaque message contient une information étendue comme le nom du champ à l'origine du message ou bien le type du message:

```
<?php

if ($robot->save() === false) {
    $messages = $robot->getMessages();

    foreach ($messages as $message) {
        echo "Message: ", $message->getMessage();
        echo "Field: ", $message->getField();
        echo "Type: ", $message->getType();
    }
}
```

`getMessages()` peut générer les types de messages de validation suivants:

Type	Description
PresenceOf	Généré lorsqu'un champ avec un attribut non-nul en base tente d'insérer/mettre à jour une valeur nulle
ConstraintViolation	Généré lorsqu'un champ à clé étrangère tente d'insérer/mettre à jour une valeur qui n'existe pas dans le modèle référencé
InvalidValue	Généré lorsqu'un validateur échoue à cause d'une valeur invalide
InvalidCreateAttempt	Produit lors de la tentative de création d'un enregistrement qui existe déjà
InvalidUpdateAttempt	Produit lors de la tentative de mise à jour d'un enregistrement qui n'existe pas

La méthode `getMessages()` peut être surchargée dans un modèle pour remplacer/traduire le message par défaut qui est généré automatiquement par l'ORM:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function getMessages()
    {
        $messages = [];

        foreach (parent::getMessages() as $message) {
            switch ($message->getType()) {
                case "InvalidCreateAttempt":
                    $messages[] = "The record cannot be created because it already_
↵exists";
                    break;

                case "InvalidUpdateAttempt":
                    $messages[] = "The record cannot be updated because it doesn't_
↵exist";
            }
        }
    }
}
```



```

        break;

        case "PresenceOf":
            $messages[] = "The field " . $message->getField() . " is mandatory
↪";

            break;
    }

    }

    return $messages;
}
}

```

Événement d'échec de validation

D'autres types d'événement sont disponibles lorsque le processus de validation détecte une incohérence:

Opération	Nom	Explication
Insertion ou M.à.j	notSaved	Déclenché lorsqu'une opération INSERT ou UPDATE échoue pour une raison quelconque
Insertion, suppression ou M.à.j	onValidation-Fails	Déclenché lorsqu'une opération de manipulation sur les données échoue

2.3.11 Travailler avec des modèles (Avancé)

Modes d'hydratation de données

Comme mentionné plus haut, les jeux de résultat sont des collections complètes d'objets, ce qui signifie que chaque résultat renvoyé est un objet qui représente une ligne dans la base de données. Ces objets peuvent être modifiés et re-sauvegardés pour la persistance:

```

<?php

use Store\Toys\Robots;

$robots = Robots::find();

// Manipulation d'un jeu complet de résultats d'objets
foreach ($robots as $robot) {
    $robot->year = 2000;

    $robot->save();
}

```

Parfois les enregistrement récupérés ne doivent être présentés à l'utilisateur qu'en lecture seule. Dans ces cas il peut être utile de changer la manière dont les enregistrement sont présentés afin de faciliter leur manipulation. La stratégie utilisée pour présenter les objets retournés dans un jeu de résultat est appelée "mode d'hydratation":

```

<?php

use Phalcon\Mvc\Model\Resultset;
use Store\Toys\Robots;

$robots = Robots::find();

```

```
// Retourne tous les robots dans un tableau
$robots->setHydrateMode(
    Resultset::HYDRATE_ARRAYS
);

foreach ($robots as $robot) {
    echo $robot["year"], PHP_EOL;
}

// Retourne tous les robots dans une stdClass
$robots->setHydrateMode(
    Resultset::HYDRATE_OBJECTS
);

foreach ($robots as $robot) {
    echo $robot->year, PHP_EOL;
}

// Retourne tous les robots dans une instance de Robots
$robots->setHydrateMode(
    Resultset::HYDRATE_RECORDS
);

foreach ($robots as $robot) {
    echo $robot->year, PHP_EOL;
}
```

Le mode d’hydratation peut également être transmis en paramètre de “find”:

```
<?php

use Phalcon\Mvc\Model\Resultset;
use Store\Toys\Robots;

$robots = Robots::find(
    [
        "hydration" => Resultset::HYDRATE_ARRAYS,
    ]
);

foreach ($robots as $robot) {
    echo $robot["year"], PHP_EOL;
}
```

Les colonnes identité auto-générées

Certains modèles peuvent avoir une colonne identité. Ces colonnes servent habituellement de clé primaire dans la table rattachée. *Phalcon\Mvc\Model* peut reconnaître la colonne identité et l’omet dans l’instruction SQL INSERT générée, laissant le SGBD générer ainsi automatiquement la valeur pour lui. Systématiquement après chaque création d’enregistrement, le champ identité est rempli avec la valeur générée par le SGBD:

```
<?php

$robot->save();
```

```
echo "The generated id is: ", $robot->id;
```

Phalcon\Mvc\Model est capable de reconnaître la colonne identité. Selon le SGBD, ces colonnes peut être des colonnes “serial” comme dans PostgreSQL ou “auto_increment” dans le cas de MySQL.

PostgreSQL utilise les séquences pour générer des valeurs numérique. Par défaut, Phalcon tente d’obtenir les valeurs depuis la séquence “<table>_<field>_seq”, comme par exemple “robots_id_seq”. Si cette séquence a un nom différent, alors la méthode “getSequenceName” doit être réalisée:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function getSequenceName()
    {
        return "robots_sequence_name";
    }
}
```

Omission de colonnes

Pour indiquer à *Phalcon\Mvc\Model* qu’il doit omettre systématiquement des champs lors de la création ou la mise à jour d’enregistrement afin de déléguer au SGDB la mission d’assigner les valeurs soit par défaut soit par l’intermédiaire d’un déclencheur:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function initialize()
    {
        // Omission de colonnes sur l'INSERT et l'UPDATE
        $this->skipAttributes([
            "year",
            "price",
        ]);

        // Omit uniquement à la création
        $this->skipAttributesOnCreate([
            "created_at",
        ]);

        // Omit uniquement à la mise à jour
        $this->skipAttributesOnUpdate([
            "year",
            "price",
        ]);
    }
}
```

```
        [
            "modified_in",
        ]
    );
}
```

Ceci ignorera ces champs sur chaque opération d'INSERT ou d'UPDATE pour l'ensemble de l'application. Si vous voulez ignorer des attributs selon l'opération INSERT ou UPDATE, vous devez spécifier un deuxième paramètre (booléen) - vrai pour le remplacement. Forcer une nouvelle valeur par défaut peut être réalisée de la façon suivante:

```
<?php

use Store\Toys\Robots;

use Phalcon\Db\RawValue;

$robot = new Robots();

$robot->name      = "Bender";
$robot->year      = 1999;
$robot->created_at = new RawValue("default");

$robot->create();
```

Une fonction de rappel peut être utilisée pour réaliser une assignation conditionnelle des valeurs par défaut:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;
use Phalcon\Db\RawValue;

class Robots extends Model
{
    public function beforeCreate()
    {
        if ($this->price > 10000) {
            $this->type = new RawValue("default");
        }
    }
}
```

N'utilisez jamais *Phalcon\Db\RawValue* pour assigner des valeurs externes (comme les entrées utilisateur) ou des données variables. Les valeurs de ces champs sont ignorées lors de la liaison de paramètres à la requête. Ceci peut être sujet à des attaques par injection SQL.

Mise à jour dynamique

Par défaut, les instructions SQL UPDATE sont créées avec toutes les colonnes définies dans le modèle (full all-field SQL update). Vous pouvez modifier des modèles spécifiques pour réaliser des mises à jour dynamiques. Dans ce cas, seuls les champs qui ont changé seront utilisés dans l'instruction SQL finale.

Dans certains cas, cela peut améliorer les performances en réduisant le trafic entre l'application et le serveur de base de données. Ceci est particulièrement utile lorsque la table contient des champs blob ou textuels:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function initialize()
    {
        $this->useDynamicUpdate(true);
    }
}
```

Correspondance indépendante de colonnes

L'ORM supporte une correspondance indépendante de colonnes, ce qui permet au développeur d'utiliser des noms de colonnes dans le modèles différents de ceux de la table. Phalcon reconnaîtra les nouveaux noms de colonnes et les renommra pour qu'ils correspondent aux colonnes respectives dans la base. Ceci est une caractéristique intéressante lorsqu'on a besoin de renommer des champs sans avoir à se soucier de toutes les requêtes du code. Un simple changement dans la correspondance de colonnes et le modèle s'occupera du reste. Par exemple:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public $code;

    public $theName;

    public $theType;

    public $theYear;

    public function columnMap()
    {
        // Les clés sont les vrais noms dans la table et
        // Les valeurs sont leur noms dans l'application
        return [
            "id"      => "code",
            "the_name" => "theName",
            "the_type" => "theType",
            "the_year" => "theYear",
        ];
    }
}
```

Ainsi vous pouvez utiliser simplement les nouveaux noms dans votre code:

```
<?php

use Store\Toys\Robots;
```

```
// Rechercher un robot par son nom
$robot = Robots::findFirst(
    "theName = 'Voltron'"
);

echo $robot->theName, "\n";

// Récupérer les robots triés par type
$robot = Robots::find(
    [
        "order" => "theType DESC",
    ]
);

foreach ($robots as $robot) {
    echo "Code: ", $robot->code, "\n";
}

// Création d'un robot
$robot = new Robots();

$robot->code      = "10101";
$robot->theName    = "Bender";
$robot->theType    = "Industrial";
$robot->theYear    = 2999;

$robot->save();
```

Prenez en considération ce qui suit lors du renommage de colonnes:

- Les références aux attributs dans les relations et validateurs doivent utiliser les nouveaux noms
- Se référer au nom réel résultera en une exception de la part de l'ORM

La correspondance indépendante de colonnes vous permet:

- D'écrire des application en utilisant vos propre conventions
- D'éliminer les suffixe ou préfixe dans votre code
- De renommer les colonnes sans avoir à modifier le code de votre application

Instantanés d'enregistrements

Des modèles spéciaux peuvent être définis pour maintenir un instantané d'enregistrements lors de l'interrogation. Vous pouvez utiliser cette caractéristique pour mettre en œuvre un audit ou bien juste pour savoir quels sont les champs qui ont changés depuis leur dernière interrogation:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function initialize()
    {
```

```

        $this->keepSnapshots(true);
    }
}

```

En activant cette caractéristique, l'application consomme un peu plus de mémoire pour conserver les valeurs d'origine obtenues depuis la persistance. Dans les modèles qui ont activés cette caractéristique vous pouvez vérifier quels sont les champs qui ont changé:

```

<?php

use Store\Toys\Robots;

// Récupère un enregistrement depuis la base
$robot = Robots::findFirst();

// Modifie une colonne
$robot->name = "Other name";

var_dump($robot->getChangedFields()); // ["name"]

var_dump($robot->hasChanged("name")); // true

var_dump($robot->hasChanged("type")); // false

```

Pointer un schéma différent

Si un modèle est rattaché à une table qui se trouve dans un autre schéma ou base que celui par défaut, vous pouvez utiliser la méthode `setSchema()` pour définir cela:

```

<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function initialize()
    {
        $this->setSchema("toys");
    }
}

```

Définition de plusieurs bases de données

Dans Phalcon, tous les modèles peuvent dépendre de la même connexion à la base de données ou en avoir un particulier. Actuellement, lorsque *Phalcon\Mvc\Model* a besoin de se connecter à la base, il interroge le service "db" dans le container de services de l'application. Vous pouvez surcharger le paramétrage de ce service dans la méthode `initialize()`:

```

<?php

use Phalcon\Db\Adapter\Pdo\Mysql as MysqlPdo;
use Phalcon\Db\Adapter\Pdo\PostgreSQL as PostgreSQLPdo;

```

```
// Ce service retourne une base de données MySQL
$di->set(
    "dbMysql",
    function () {
        return new MysqlPdo(
            [
                "host"      => "localhost",
                "username"  => "root",
                "password"  => "secret",
                "dbname"    => "invo",
            ]
        );
    }
);

// Ce service retourne une base de données PostgreSQL
$di->set(
    "dbPostgres",
    function () {
        return new PostgreSQLPdo(
            [
                "host"      => "localhost",
                "username"  => "postgres",
                "password"  => "",
                "dbname"    => "invo",
            ]
        );
    }
);
```

Ainsi, dans la méthode `initialize()`, nous définissons le service de connexion pour le modèle:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function initialize()
    {
        $this->setConnectionService("dbPostgres");
    }
}
```

Mais Phalcon offre encore plus de flexibilité, nous pouvons définir une connexion pour la lecture et une pour l'écriture. Ceci est particulièrement utile pour équilibrer la charge entre les bases de données dans une architecture maître-esclave:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
```



```

public function initialize()
{
    $this->setReadConnectionService("dbSlave");

    $this->setWriteConnectionService("dbMaster");
}
}

```

L'ORM fournit aussi la capacité d'**Horizontal Sharding**, en vous permettant de mettre en place une sélection de “shard” en fonction des conditions actuelles de la requête:

```

<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    /**
     * Sélection dynamique d'un shard
     *
     * @param array $intermediate
     * @param array $bindParams
     * @param array $bindTypes
     */
    public function selectReadConnection($intermediate, $bindParams, $bindTypes)
    {
        // Vérifie la présence d'une clause 'where'
        if (isset($intermediate["where"])) {
            $conditions = $intermediate["where"];

            // Choix des shard potentiels en fonction des conditions
            if ($conditions["left"]["name"] === "id") {
                $id = $conditions["right"]["value"];

                if ($id > 0 && $id < 10000) {
                    return $this->getDI()->get("dbShard1");
                }

                if ($id > 10000) {
                    return $this->getDI()->get("dbShard2");
                }
            }
        }

        // Utilisation du shard par défaut
        return $this->getDI()->get("dbShard0");
    }
}

```

La méthode `selectReadConnection()` est appelée pour sélectionner la bonne connexion. Cette méthode intercepte chaque nouvelle requête exécutée:

```

<?php

use Store\Toys\Robots;

```

```
$robot = Robots::findFirst('id = 101');
```

Injection de services dans les modèles

Si vous devez accéder aux services de l'application à partir d'un modèle, l'exemple qui suit vous montre comment faire:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function notSaved()
    {
        // Obtention du service flash à partir du conteneur DI
        $flash = $this->getDI()->getFlash();

        $messages = $this->getMessages();

        // Affiche les messages de validation
        foreach ($messages as $message) {
            $flash->error($message);
        }
    }
}
```

L'événement “notSaved” est déclenché à chaque échec des actions “create” ou “update”. Ainsi nous envoyons les messages de validation dans le service “flash” obtenu depuis le conteneur DI. En faisant comme cela, nous n'avons pas besoin d'imprimer le message après chaque sauvegarde.

Activation/Désactivation de fonctionnalités

Dans l'ORM nous avons mis en place un mécanisme qui vous permette d'activer ou de désactiver à la volée des fonctionnalités particulière ou des options. Vous pouvez désactiver de que vous n'utilisez pas dans l'ORM. Ces options peuvent également être désactivées temporairement si nécessaire:

```
<?php

use Phalcon\Mvc\Model;

Model::setup(
[
    "events"          => false,
    "columnRenaming" => false,
]
);
```

Les options sont:

Option	Description	Dé-faut
events	Enables/Disables les rappels, crochets et notifications d'événement de tous les modèles	true
columnRenaming	Active/Désactive le renommage de colonnes	true
notNullValidations	L'ORM valide automatiquement les colonnes non nulles présentes dans la table rattachée	true
virtualForeignKeys	Active/Désactive les clés étrangères virtuelles	true
phqlLiterals	Active/Désactive les littéraux dans le parser PHQL	true
lateStateBinding	Active/Désactive l'état tardif de la méthode <code>Mvc\Model::cloneResultMap()</code>	false

Composant autonome

L'utilisation de *Phalcon\Mvc\Model* en mode autonome est montrée ci-dessous:

```
<?php

use Phalcon\Di;
use Phalcon\Mvc\Model;
use Phalcon\Mvc\Model\Manager as ModelsManager;
use Phalcon\Db\Adapter\Pdo\Sqlite as Connection;
use Phalcon\Mvc\Model\Metadata\Memory as MetaData;

$di = new Di();

// Etablissement d'une connexion
$di->set(
    "db",
    new Connection(
        [
            "dbname" => "sample.db",
        ]
    )
);

// Définition d'un gestionnaire de modèles
$di->set(
    "modelsManager",
    new ModelsManager()
);

// Utilisation d'un adaptateur de métadonnées
$di->set(
    "modelsMetadata",
    new MetaData()
);

// Création d'un modèle
class Robots extends Model
{
}

// Utilisation du modèle
```

```
echo Robots::count();
```

2.3.12 Phalcon Query Language (PHQL)

Phalcon Query Language, PhalconQL or simply PHQL is a high-level, object-oriented SQL dialect that allows to write queries using a standardized SQL-like language. PHQL is implemented as a parser (written in C) that translates syntax in that of the target RDBMS.

To achieve the highest performance possible, Phalcon provides a parser that uses the same technology as [SQLite](#). This technology provides a small in-memory parser with a very low memory footprint that is also thread-safe.

The parser first checks the syntax of the pass PHQL statement, then builds an intermediate representation of the statement and finally it converts it to the respective SQL dialect of the target RDBMS.

In PHQL, we've implemented a set of features to make your access to databases more secure:

- Bound parameters are part of the PHQL language helping you to secure your code
- PHQL only allows one SQL statement to be executed per call preventing injections
- PHQL ignores all SQL comments which are often used in SQL injections
- PHQL only allows data manipulation statements, avoiding altering or dropping tables/databases by mistake or externally without authorization
- PHQL implements a high-level abstraction allowing you to handle tables as models and fields as class attributes

Usage Example

To better explain how PHQL works consider the following example. We have two models “Cars” and “Brands”:

```
<?php
use Phalcon\Mvc\Model;

class Cars extends Model
{
    public $id;

    public $name;

    public $brand_id;

    public $price;

    public $year;

    public $style;

    /**
     * This model is mapped to the table sample_cars
     */
    public function getSource()
    {
        return "sample_cars";
    }

    /**
```

```

    * A car only has a Brand, but a Brand have many Cars
    */
    public function initialize()
    {
        $this->belongsTo("brand_id", "Brands", "id");
    }
}

```

And every Car has a Brand, so a Brand has many Cars:

```

<?php

use Phalcon\Mvc\Model;

class Brands extends Model
{
    public $id;

    public $name;

    /**
     * The model Brands is mapped to the "sample_brands" table
     */
    public function getSource()
    {
        return "sample_brands";
    }

    /**
     * A Brand can have many Cars
     */
    public function initialize()
    {
        $this->hasMany("id", "Cars", "brand_id");
    }
}

```

Creating PHQL Queries

PHQL queries can be created just by instantiating the class *Phalcon\Mvc\Model\Query*:

```

<?php

use Phalcon\Mvc\Model\Query;

// Instantiate the Query
$query = new Query(
    "SELECT * FROM Cars",
    $this->getDI()
);

// Execute the query returning a result if any
$cars = $query->execute();

```

From a controller or a view, it's easy to create/execute them using an injected *models manager*:

```
<?php

// Executing a simple query
$query = $this->modelsManager->createQuery("SELECT * FROM Cars");
$cars = $query->execute();

// With bound parameters
$query = $this->modelsManager->createQuery("SELECT * FROM Cars WHERE name = :name:");
$cars = $query->execute(
    [
        "name" => "Audi",
    ]
);
```

Or simply execute it:

```
<?php

// Executing a simple query
$cars = $this->modelsManager->executeQuery(
    "SELECT * FROM Cars"
);

// Executing with bound parameters
$cars = $this->modelsManager->executeQuery(
    "SELECT * FROM Cars WHERE name = :name:",
    [
        "name" => "Audi",
    ]
);
```

Selecting Records

As the familiar SQL, PHQL allows querying of records using the SELECT statement we know, except that instead of specifying tables, we use the models classes:

```
<?php

$query = $manager->createQuery(
    "SELECT * FROM Cars ORDER BY Cars.name"
);

$query = $manager->createQuery(
    "SELECT Cars.name FROM Cars ORDER BY Cars.name"
);
```

Classes in namespaces are also allowed:

```
<?php

$phql = "SELECT * FROM Formula\Cars ORDER BY Formula\Cars.name";
$query = $manager->createQuery($phql);

$phql = "SELECT Formula\Cars.name FROM Formula\Cars ORDER BY Formula\Cars.name";
$query = $manager->createQuery($phql);
```

```
$phql = "SELECT c.name FROM Formula\Cars c ORDER BY c.name";
$query = $manager->createQuery($phql);
```

Most of the SQL standard is supported by PHQL, even nonstandard directives such as LIMIT:

```
<?php

$phql = "SELECT c.name FROM Cars AS c WHERE c.brand_id = 21 ORDER BY c.name LIMIT 100
↪";

$query = $manager->createQuery($phql);
```

Result Types

Depending on the type of columns we query, the result type will vary. If you retrieve a single whole object, then the object returned is a *Phalcon\Mvc\Model\Resultset\Simple*. This kind of resultset is a set of complete model objects:

```
<?php

$phql = "SELECT c.* FROM Cars AS c ORDER BY c.name";

$cars = $manager->executeQuery($phql);

foreach ($cars as $car) {
    echo "Name: ", $car->name, "\n";
}
```

This is exactly the same as:

```
<?php

$cars = Cars::find(
    [
        "order" => "name"
    ]
);

foreach ($cars as $car) {
    echo "Name: ", $car->name, "\n";
}
```

Complete objects can be modified and re-saved in the database because they represent a complete record of the associated table. There are other types of queries that do not return complete objects, for example:

```
<?php

$phql = "SELECT c.id, c.name FROM Cars AS c ORDER BY c.name";

$cars = $manager->executeQuery($phql);

foreach ($cars as $car) {
    echo "Name: ", $car->name, "\n";
}
```

We are only requesting some fields in the table, therefore those cannot be considered an entire object, so the returned object is still a resultset of type *Phalcon\Mvc\Model\Resultset\Simple*. However, each element is a standard object that

only contain the two columns that were requested.

These values that don't represent complete objects are what we call scalars. PHQL allows you to query all types of scalars: fields, functions, literals, expressions, etc..:

```
<?php

$phql = "SELECT CONCAT(c.id, ' ', c.name) AS id_name FROM Cars AS c ORDER BY c.name";

$cars = $manager->executeQuery($phql);

foreach ($cars as $car) {
    echo $car->id_name, "\n";
}
```

As we can query complete objects or scalars, we can also query both at once:

```
<?php

$phql = "SELECT c.price*0.16 AS taxes, c.* FROM Cars AS c ORDER BY c.name";

$result = $manager->executeQuery($phql);
```

The result in this case is an object *Phalcon\Mvc\Model\Resultset\Complex*. This allows access to both complete objects and scalars at once:

```
<?php

foreach ($result as $row) {
    echo "Name: ", $row->cars->name, "\n";
    echo "Price: ", $row->cars->price, "\n";
    echo "Taxes: ", $row->taxes, "\n";
}
```

Scalars are mapped as properties of each “row”, while complete objects are mapped as properties with the name of its related model.

Joins

It's easy to request records from multiple models using PHQL. Most kinds of Joins are supported. As we defined relationships in the models, PHQL adds these conditions automatically:

```
<?php

$phql = "SELECT Cars.name AS car_name, Brands.name AS brand_name FROM Cars JOIN Brands
↪";

$rows = $manager->executeQuery($phql);

foreach ($rows as $row) {
    echo $row->car_name, "\n";
    echo $row->brand_name, "\n";
}
```

By default, an INNER JOIN is assumed. You can specify the type of JOIN in the query:


```
<?php

$phql = "SELECT Cars.*, Brands.* FROM Cars INNER JOIN Brands";
$rows = $manager->executeQuery($phql);

$phql = "SELECT Cars.*, Brands.* FROM Cars LEFT JOIN Brands";
$rows = $manager->executeQuery($phql);

$phql = "SELECT Cars.*, Brands.* FROM Cars LEFT OUTER JOIN Brands";
$rows = $manager->executeQuery($phql);

$phql = "SELECT Cars.*, Brands.* FROM Cars CROSS JOIN Brands";
$rows = $manager->executeQuery($phql);
```

It is also possible to manually set the conditions of the JOIN:

```
<?php

$phql = "SELECT Cars.*, Brands.* FROM Cars INNER JOIN Brands ON Brands.id = Cars.
↳brands_id";

$rows = $manager->executeQuery($phql);
```

Also, the joins can be created using multiple tables in the FROM clause:

```
<?php

$phql = "SELECT Cars.*, Brands.* FROM Cars, Brands WHERE Brands.id = Cars.brands_id";
$rows = $manager->executeQuery($phql);

foreach ($rows as $row) {
    echo "Car: ", $row->cars->name, "\n";
    echo "Brand: ", $row->brands->name, "\n";
}
```

If an alias is used to rename the models in the query, those will be used to name the attributes in the every row of the result:

```
<?php

$phql = "SELECT c.*, b.* FROM Cars c, Brands b WHERE b.id = c.brands_id";
$rows = $manager->executeQuery($phql);

foreach ($rows as $row) {
    echo "Car: ", $row->c->name, "\n";
    echo "Brand: ", $row->b->name, "\n";
}
```

When the joined model has a many-to-many relation to the ‘from’ model, the intermediate model is implicitly added to the generated query:

```
<?php

$phql = "SELECT Artists.name, Songs.name FROM Artists " .
        "JOIN Songs WHERE Artists.genre = 'Trip-Hop'";
```

```
$result = $this->modelsManager->executeQuery($phql);
```

This code executes the following SQL in MySQL:

```
SELECT `artists`.`name`, `songs`.`name` FROM `artists`  
INNER JOIN `albums` ON `albums`.`artists_id` = `artists`.`id`  
INNER JOIN `songs` ON `albums`.`songs_id` = `songs`.`id`  
WHERE `artists`.`genre` = 'Trip-Hop'
```

Aggregations

The following examples show how to use aggregations in PHQL:

```
<?php  
  
// How much are the prices of all the cars?  
$phql = "SELECT SUM(price) AS summatory FROM Cars";  
$row = $manager->executeQuery($phql)->getFirst();  
echo $row['summatory'];  
  
// How many cars are by each brand?  
$phql = "SELECT Cars.brand_id, COUNT(*) FROM Cars GROUP BY Cars.brand_id";  
$rows = $manager->executeQuery($phql);  
foreach ($rows as $row) {  
    echo $row->brand_id, ' ', $row["1"], "\n";  
}  
  
// How many cars are by each brand?  
$phql = "SELECT Brands.name, COUNT(*) FROM Cars JOIN Brands GROUP BY 1";  
$rows = $manager->executeQuery($phql);  
foreach ($rows as $row) {  
    echo $row->name, ' ', $row["1"], "\n";  
}  
  
$phql = "SELECT MAX(price) AS maximum, MIN(price) AS minimum FROM Cars";  
$rows = $manager->executeQuery($phql);  
foreach ($rows as $row) {  
    echo $row["maximum"], ' ', $row["minimum"], "\n";  
}  
  
// Count distinct used brands  
$phql = "SELECT COUNT(DISTINCT brand_id) AS brandId FROM Cars";  
$rows = $manager->executeQuery($phql);  
foreach ($rows as $row) {  
    echo $row->brandId, "\n";  
}
```

Conditions

Conditions allow us to filter the set of records we want to query. The WHERE clause allows to do that:

```
<?php
```

```
// Simple conditions
$phql = "SELECT * FROM Cars WHERE Cars.name = 'Lamborghini Espada'";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.price > 10000";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE TRIM(Cars.name) = 'Audi R8'";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.name LIKE 'Ferrari%'";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.name NOT LIKE 'Ferrari%'";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.price IS NULL";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.id IN (120, 121, 122)";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.id NOT IN (430, 431)";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.id BETWEEN 1 AND 100";
$cars = $manager->executeQuery($phql);
```

Also, as part of PHQL, prepared parameters automatically escape the input data, introducing more security:

```
<?php

$phql = "SELECT * FROM Cars WHERE Cars.name = :name:";
$cars = $manager->executeQuery(
    $phql,
    [
        "name" => "Lamborghini Espada"
    ]
);

$phql = "SELECT * FROM Cars WHERE Cars.name = ?0";
$cars = $manager->executeQuery(
    $phql,
    [
        0 => "Lamborghini Espada"
    ]
);
```

Inserting Data

With PHQL it's possible to insert data using the familiar INSERT statement:

```
<?php

// Inserting without columns
$phql = "INSERT INTO Cars VALUES (NULL, 'Lamborghini Espada', "
```

```
. "7, 10000.00, 1969, 'Grand Tourer')";
$manager->executeQuery($phql);

// Specifying columns to insert
$phql = "INSERT INTO Cars (name, brand_id, year, style) "
    . "VALUES ('Lamborghini Espada', 7, 1969, 'Grand Tourer')";
$manager->executeQuery($phql);

// Inserting using placeholders
$phql = "INSERT INTO Cars (name, brand_id, year, style) "
    . "VALUES (:name:, :brand_id:, :year:, :style)";
$manager->executeQuery(
    $phql,
    [
        "name"      => "Lamborghini Espada",
        "brand_id"  => 7,
        "year"      => 1969,
        "style"     => "Grand Tourer",
    ]
);
```

Phalcon doesn't only transform the PHQL statements into SQL. All events and business rules defined in the model are executed as if we created individual objects manually. Let's add a business rule on the model cars. A car cannot cost less than \$ 10,000:

```
<?php

use Phalcon\Mvc\Model;
use Phalcon\Mvc\Model\Message;

class Cars extends Model
{
    public function beforeCreate()
    {
        if ($this->price < 10000) {
            $this->appendMessage(
                new Message("A car cannot cost less than $ 10,000")
            );

            return false;
        }
    }
}
```

If we made the following INSERT in the models Cars, the operation will not be successful because the price does not meet the business rule that we implemented. By checking the status of the insertion we can print any validation messages generated internally:

```
<?php

$phql = "INSERT INTO Cars VALUES (NULL, 'Nissan Versa', 7, 9999.00, 2015, 'Sedan')";

$result = $manager->executeQuery($phql);

if ($result->success() === false) {
    foreach ($result->getMessages() as $message) {
        echo $message->getMessage();
    }
}
```

```
}
```

Updating Data

Updating rows is very similar than inserting rows. As you may know, the instruction to update records is UPDATE. When a record is updated the events related to the update operation will be executed for each row.

```
<?php

// Updating a single column
$sql = "UPDATE Cars SET price = 15000.00 WHERE id = 101";
$manager->executeQuery($sql);

// Updating multiples columns
$sql = "UPDATE Cars SET price = 15000.00, type = 'Sedan' WHERE id = 101";
$manager->executeQuery($sql);

// Updating multiples rows
$sql = "UPDATE Cars SET price = 7000.00, type = 'Sedan' WHERE brands_id > 5";
$manager->executeQuery($sql);

// Using placeholders
$sql = "UPDATE Cars SET price = ?0, type = ?1 WHERE brands_id > ?2";
$manager->executeQuery(
    $sql,
    [
        0 => 7000.00,
        1 => 'Sedan',
        2 => 5,
    ]
);
```

An UPDATE statement performs the update in two phases:

- First, if the UPDATE has a WHERE clause it retrieves all the objects that match these criteria,
- Second, based on the queried objects it updates/changes the requested attributes storing them to the relational database

This way of operation allows that events, virtual foreign keys and validations take part of the updating process. In summary, the following code:

```
<?php

$sql = "UPDATE Cars SET price = 15000.00 WHERE id > 101";

$result = $manager->executeQuery($sql);

if ($result->success() === false) {
    $messages = $result->getMessages();

    foreach ($messages as $message) {
        echo $message->getMessage();
    }
}
```

is somewhat equivalent to:

```
<?php

$messages = null;

$process = function () use (&$messages) {
    $cars = Cars::find("id > 101");

    foreach ($cars as $car) {
        $car->price = 15000;

        if ($car->save() === false) {
            $messages = $car->getMessages();

            return false;
        }
    }

    return true;
};

$success = $process();
```

Deleting Data

When a record is deleted the events related to the delete operation will be executed for each row:

```
<?php

// Deleting a single row
$sql = "DELETE FROM Cars WHERE id = 101";
$manager->executeQuery($sql);

// Deleting multiple rows
$sql = "DELETE FROM Cars WHERE id > 100";
$manager->executeQuery($sql);

// Using placeholders
$sql = "DELETE FROM Cars WHERE id BETWEEN :initial: AND :final:";
$manager->executeQuery(
    $sql,
    [
        "initial" => 1,
        "final"   => 100,
    ]
);
```

DELETE operations are also executed in two phases like UPDATES. To check if the deletion produces any validation messages you should check the status code returned:

```
<?php

// Deleting multiple rows
$sql = "DELETE FROM Cars WHERE id > 100";

$result = $manager->executeQuery($sql);
```

```

if ($result->success() === false) {
    $messages = $result->getMessages();

    foreach ($messages as $message) {
        echo $message->getMessage();
    }
}

```

Creating queries using the Query Builder

A builder is available to create PHQL queries without the need to write PHQL statements, also providing IDE facilities:

```

<?php

// Getting a whole set
$robots = $this->modelsManager->createBuilder()
    ->from("Robots")
    ->join("RobotsParts")
    ->orderBy("Robots.name")
    ->getQuery()
    ->execute();

// Getting the first row
$robots = $this->modelsManager->createBuilder()
    ->from("Robots")
    ->join("RobotsParts")
    ->orderBy("Robots.name")
    ->getQuery()
    ->getSingleResult();

```

That is the same as:

```

<?php

$phql = "SELECT Robots.* FROM Robots JOIN RobotsParts p ORDER BY Robots.name LIMIT 20";

$result = $manager->executeQuery($phql);

```

More examples of the builder:

```

<?php

// 'SELECT Robots.* FROM Robots';
$builder->from("Robots");

// 'SELECT Robots.*, RobotsParts.* FROM Robots, RobotsParts';
$builder->from(
    [
        "Robots",
        "RobotsParts",
    ]
);

// 'SELECT * FROM Robots';
$phql = $builder->columns("*");

```

```
        ->from("Robots");

// 'SELECT id FROM Robots';
$builder->columns("id")
    ->from("Robots");

// 'SELECT id, name FROM Robots';
$builder->columns(["id", "name"])
    ->from("Robots");

// 'SELECT Robots.* FROM Robots WHERE Robots.name = "Voltron"';
$builder->from("Robots")
    ->where("Robots.name = 'Voltron'");

// 'SELECT Robots.* FROM Robots WHERE Robots.id = 100';
$builder->from("Robots")
    ->where(100);

// 'SELECT Robots.* FROM Robots WHERE Robots.type = "virtual" AND Robots.id > 50';
$builder->from("Robots")
    ->where("type = 'virtual'")
    ->andWhere("id > 50");

// 'SELECT Robots.* FROM Robots WHERE Robots.type = "virtual" OR Robots.id > 50';
$builder->from("Robots")
    ->where("type = 'virtual'")
    ->orWhere("id > 50");

// 'SELECT Robots.* FROM Robots GROUP BY Robots.name';
$builder->from("Robots")
    ->groupBy("Robots.name");

// 'SELECT Robots.* FROM Robots GROUP BY Robots.name, Robots.id';
$builder->from("Robots")
    ->groupBy(["Robots.name", "Robots.id"]);

// 'SELECT Robots.name, SUM(Robots.price) FROM Robots GROUP BY Robots.name';
$builder->columns(["Robots.name", "SUM(Robots.price)"])
    ->from("Robots")
    ->groupBy("Robots.name");

// 'SELECT Robots.name, SUM(Robots.price) FROM Robots GROUP BY Robots.name HAVING
↳SUM(Robots.price) > 1000';
$builder->columns(["Robots.name", "SUM(Robots.price)"])
    ->from("Robots")
    ->groupBy("Robots.name")
    ->having("SUM(Robots.price) > 1000");

// 'SELECT Robots.* FROM Robots JOIN RobotsParts';
$builder->from("Robots")
    ->join("RobotsParts");

// 'SELECT Robots.* FROM Robots JOIN RobotsParts AS p';
$builder->from("Robots")
    ->join("RobotsParts", null, "p");

// 'SELECT Robots.* FROM Robots JOIN RobotsParts ON Robots.id = RobotsParts.robots_id
↳AS p';
```



```

$builder->from("Robots")
    ->join("RobotsParts", "Robots.id = RobotsParts.robots_id", "p");

// 'SELECT Robots.* FROM Robots
// JOIN RobotsParts ON Robots.id = RobotsParts.robots_id AS p
// JOIN Parts ON Parts.id = RobotsParts.parts_id AS t';
$builder->from("Robots")
    ->join("RobotsParts", "Robots.id = RobotsParts.robots_id", "p")
    ->join("Parts", "Parts.id = RobotsParts.parts_id", "t");

// 'SELECT r.* FROM Robots AS r';
$builder->addFrom("Robots", "r");

// 'SELECT Robots.*, p.* FROM Robots, Parts AS p';
$builder->from("Robots")
    ->addFrom("Parts", "p");

// 'SELECT r.*, p.* FROM Robots AS r, Parts AS p';
$builder->from(["r" => "Robots"])
    ->addFrom("Parts", "p");

// 'SELECT r.*, p.* FROM Robots AS r, Parts AS p';
$builder->from(["r" => "Robots", "p" => "Parts"]);

// 'SELECT Robots.* FROM Robots LIMIT 10';
$builder->from("Robots")
    ->limit(10);

// 'SELECT Robots.* FROM Robots LIMIT 10 OFFSET 5';
$builder->from("Robots")
    ->limit(10, 5);

// 'SELECT Robots.* FROM Robots WHERE id BETWEEN 1 AND 100';
$builder->from("Robots")
    ->betweenWhere("id", 1, 100);

// 'SELECT Robots.* FROM Robots WHERE id IN (1, 2, 3)';
$builder->from("Robots")
    ->inWhere("id", [1, 2, 3]);

// 'SELECT Robots.* FROM Robots WHERE id NOT IN (1, 2, 3)';
$builder->from("Robots")
    ->notInWhere("id", [1, 2, 3]);

// 'SELECT Robots.* FROM Robots WHERE name LIKE '%Art%';
$builder->from("Robots")
    ->where("name LIKE :name:", ["name" => "%" . $name . "%"]);

// 'SELECT r.* FROM Store\Robots WHERE r.name LIKE '%Art%';
$builder->from(["r" => 'Store\Robots'])
    ->where("r.name LIKE :name:", ["name" => "%" . $name . "%"]);

```

Bound Parameters

Bound parameters in the query builder can be set as the query is constructed or past all at once when executing:

```
<?php

// Passing parameters in the query construction
$robots = $this->modelsManager->createBuilder()
    ->from("Robots")
    ->where("name = :name:", ["name" => $name])
    ->andWhere("type = :type:", ["type" => $type])
    ->getQuery()
    ->execute();

// Passing parameters in query execution
$robots = $this->modelsManager->createBuilder()
    ->from("Robots")
    ->where("name = :name:")
    ->andWhere("type = :type:")
    ->getQuery()
    ->execute(["name" => $name, "type" => $type]);
```

Disallow literals in PHQL

Literals can be disabled in PHQL, this means that directly using strings, numbers and boolean values in PHQL strings will be disallowed. If PHQL statements are created embedding external data on them, this could open the application to potential SQL injections:

```
<?php

$login = 'voltron';

$phql = "SELECT * FROM Models\Users WHERE login = '$login'";

$result = $manager->executeQuery($phql);
```

If \$login is changed to ' OR '' = ', the produced PHQL is:

```
SELECT * FROM Models\Users WHERE login = '' OR '' = ''
```

Which is always true no matter what the login stored in the database is.

If literals are disallowed strings can be used as part of a PHQL statement, thus an exception will be thrown forcing the developer to use bound parameters. The same query can be written in a secure way like this:

```
<?php

$phql = "SELECT Robots.* FROM Robots WHERE Robots.name = :name:";

$result = $manager->executeQuery(
    $phql,
    [
        "name" => $name,
    ]
);
```

You can disallow literals in the following way:

```
<?php
```

```
use Phalcon\Mvc\Model;

Model::setup(
    [
        "phqlLiterals" => false
    ]
);
```

Bound parameters can be used even if literals are allowed or not. Disallowing them is just another security decision a developer could take in web applications.

Escaping Reserved Words

PHQL has a few reserved words, if you want to use any of them as attributes or models names, you need to escape those words using the cross-database escaping delimiters '[' and ']':

```
<?php

$phql = "SELECT * FROM [Update]";
$result = $manager->executeQuery($phql);

$phql = "SELECT id, [Like] FROM Posts";
$result = $manager->executeQuery($phql);
```

The delimiters are dynamically translated to valid delimiters depending on the database system where the application is currently running on.

PHQL Lifecycle

Being a high-level language, PHQL gives developers the ability to personalize and customize different aspects in order to suit their needs. The following is the life cycle of each PHQL statement executed:

- The PHQL is parsed and converted into an Intermediate Representation (IR) which is independent of the SQL implemented by database system
- The IR is converted to valid SQL according to the database system associated to the model
- PHQL statements are parsed once and cached in memory. Further executions of the same statement result in a slightly faster execution

Using Raw SQL

A database system could offer specific SQL extensions that aren't supported by PHQL, in this case, a raw SQL can be appropriate:

```
<?php

use Phalcon\Mvc\Model;
use Phalcon\Mvc\Model\Resultset\Simple as Resultset;

class Robots extends Model
{
    public static function findByCreateInterval()
    {
        // A raw SQL statement
```

```
$sql = "SELECT * FROM robots WHERE id > 0";

// Base model
$robot = new Robots();

// Execute the query
return new Resultset(
    null,
    $robot,
    $robot->getReadConnection()->query($sql)
);
}
```

If Raw SQL queries are common in your application a generic method could be added to your model:

```
<?php

use Phalcon\Mvc\Model;
use Phalcon\Mvc\Model\Resultset\Simple as Resultset;

class Robots extends Model
{
    public static function findByRawSql($conditions, $params = null)
    {
        // A raw SQL statement
        $sql = "SELECT * FROM robots WHERE $conditions";

        // Base model
        $robot = new Robots();

        // Execute the query
        return new Resultset(
            null,
            $robot,
            $robot->getReadConnection()->query($sql, $params)
        );
    }
}
```

The above `findByRawSql` could be used as follows:

```
<?php

$robots = Robots::findByRawSql(
    "id > ?",
    [
        10
    ]
);
```

Troubleshooting

Some things to keep in mind when using PHQL:

- Classes are case-sensitive, if a class is not defined with the same name as it was created this could lead to an unexpected behavior in operating systems with case-sensitive file systems such as Linux.

- Correct charset must be defined in the connection to bind parameters with success.
- Aliased classes aren't replaced by full namespaced classes since this only occurs in PHP code and not inside strings.
- If column renaming is enabled avoid using column aliases with the same name as columns to be renamed, this may confuse the query resolver.

2.3.13 Caching in the ORM

Every application is different, we could have models whose data change frequently and others that rarely change. Accessing database systems is often one of the most common bottlenecks in terms of performance. This is due to the complex connection/communication processes that PHP must do in each request to obtain data from the database. Therefore, if we want to achieve good performance we need to add some layers of caching where the application requires it.

This chapter explains the possible points where it is possible to implement caching to improve performance. The framework gives you the tools to implement the cache where you demand of it according to the architecture of your application.

Caching Resultsets

A well established technique to avoid continuously accessing to the database is to cache resultsets that don't change frequently using a system with faster access (usually memory).

When *Phalcon\Mvc\Model* requires a service to cache resultsets, it will request it to the Dependency Injector Container with the convention name "modelsCache".

As Phalcon provides a component to *cache* any kind of data, we'll explain how to integrate it with Models. First, you must register it as a service in the services container:

```
<?php

use Phalcon\Cache\Frontend\Data as FrontendData;
use Phalcon\Cache\Backend\Memcache as BackendMemcache;

// Set the models cache service
$di->set(
    "modelsCache",
    function () {
        // Cache data for one day by default
        $frontCache = new FrontendData(
            [
                "lifetime" => 86400,
            ]
        );

        // Memcached connection settings
        $cache = new BackendMemcache(
            $frontCache,
            [
                "host" => "localhost",
                "port" => "11211",
            ]
        );

        return $cache;
    }
);
```

```
    }  
);
```

You have complete control in creating and customizing the cache before being used by registering the service as an anonymous function. Once the cache setup is properly defined you could cache resultsets as follows:

```
<?php  
  
// Get products without caching  
$products = Products::find();  
  
// Just cache the resultset. The cache will expire in 1 hour (3600 seconds)  
$products = Products::find(  
    [  
        "cache" => [  
            "key" => "my-cache",  
        ],  
    ],  
);  
  
// Cache the resultset for only for 5 minutes  
$products = Products::find(  
    [  
        "cache" => [  
            "key"      => "my-cache",  
            "lifetime" => 300,  
        ],  
    ],  
);  
  
// Use the 'cache' service from the DI instead of 'modelsCache'  
$products = Products::find(  
    [  
        "cache" => [  
            "key"      => "my-cache",  
            "service" => "cache",  
        ],  
    ],  
);
```

Caching could be also applied to resultsets generated using relationships:

```
<?php  
  
// Query some post  
$post = Post::findFirst();  
  
// Get comments related to a post, also cache it  
$comments = $post->getComments(  
    [  
        "cache" => [  
            "key" => "my-key",  
        ],  
    ],  
);  
  
// Get comments related to a post, setting lifetime  
$comments = $post->getComments(  
    [  
        "cache" => [  
            "key"      => "my-key",  
            "lifetime" => 300,  
        ],  
    ],  
);
```

```
[
    "cache" => [
        "key"      => "my-key",
        "lifetime" => 3600,
    ],
]
);
```

When a cached resultset needs to be invalidated, you can simply delete it from the cache using the previously specified key.

Note that not all resultsets should be cached. Results that change very frequently should not be cached since they are invalidated very quickly and caching in that case impacts performance. Additionally, large datasets that do not change frequently could be cached, but that is a decision that the developer has to make based on the available caching mechanism and whether the performance impact to simply retrieve that data in the first place is acceptable.

Forcing Cache

Earlier we saw how *Phalcon\Mvc\Model* integrates with the caching component provided by the framework. To make a record/resultset cacheable we pass the key 'cache' in the array of parameters:

```
<?php

// Cache the resultset for only for 5 minutes
$product = Products::find(
    [
        "cache" => [
            "key"      => "my-cache",
            "lifetime" => 300,
        ],
    ]
);
```

This gives us the freedom to cache specific queries, however if we want to cache globally every query performed over the model, we can override the `find() / :code:`findFirst()` method to force every query to be cached:

```
<?php

use Phalcon\Mvc\Model;

class Robots extends Model
{
    /**
     * Implement a method that returns a string key based
     * on the query parameters
     */
    protected static function _createKey($parameters)
    {
        $uniqueKey = [];

        foreach ($parameters as $key => $value) {
            if (is_scalar($value)) {
                $uniqueKey[] = $key . ":" . $value;
            } elseif (is_array($value)) {
                $uniqueKey[] = $key . ":" . self::_createKey($value) . " ";
            }
        }
    }
}
```

```
        return join(",", $uniqueKey);
    }

    public static function find($parameters = null)
    {
        // Convert the parameters to an array
        if (!is_array($parameters)) {
            $parameters = [$parameters];
        }

        // Check if a cache key wasn't passed
        // and create the cache parameters
        if (!isset($parameters["cache"])) {
            $parameters["cache"] = [
                "key" => self::_createKey($parameters),
                "lifetime" => 300,
            ];
        }

        return parent::find($parameters);
    }

    public static function findFirst($parameters = null)
    {
        // ...
    }
}
```

Accessing the database is several times slower than calculating a cache key. You're free to implement any key generation strategy you find to better for your needs. Note that a good key avoids collisions as much as possible - meaning that different keys should return unrelated records.

This gives you full control on how the cache should be implemented for each model. If this strategy is common to several models you can create a base class for all of them:

```
<?php

use Phalcon\Mvc\Model;

class CacheableModel extends Model
{
    protected static function _createKey($parameters)
    {
        // ... Create a cache key based on the parameters
    }

    public static function find($parameters = null)
    {
        // ... Custom caching strategy
    }

    public static function findFirst($parameters = null)
    {
        // ... Custom caching strategy
    }
}
```


Then use this class as base class for each ‘Cacheable’ model:

```
<?php

class Robots extends CacheableModel
{
}

```

Caching PHQL Queries

Regardless of the syntax we used to create them, all queries in the ORM are handled internally using PHQL. This language gives you much more freedom to create all kinds of queries. Of course these queries can be cached:

```
<?php

$phql = "SELECT * FROM Cars WHERE name = :name:";

$query = $this->modelsManager->createQuery($phql);

$query->cache([
    [
        "key"      => "cars-by-name",
        "lifetime" => 300,
    ]
]);

$cars = $query->execute([
    [
        "name" => "Audi",
    ]
]);

```

Reusable Related Records

Some models may have relationships with other models. This allows us to easily check the records that relate to instances in memory:

```
<?php

// Get some invoice
$invoice = Invoices::findFirst();

// Get the customer related to the invoice
$customer = $invoice->customer;

// Print his/her name
echo $customer->name, "\n";

```

This example is very simple, a customer is queried and can be used as required, for example, to show its name. This also applies if we retrieve a set of invoices to show customers that correspond to these invoices:

```
<?php

// Get a set of invoices

```

```
// SELECT * FROM invoices;
$invoices = Invoices::find();

foreach ($invoices as $invoice) {
    // Get the customer related to the invoice
    // SELECT * FROM customers WHERE id = ?;
    $customer = $invoice->customer;

    // Print his/her name
    echo $customer->name, "\n";
}
```

A customer may have one or more bills so, in this example, the same customer record may be unnecessarily queried several times. To avoid this, we could mark the relationship as reusable; by doing so, we tell the ORM to automatically reuse the records from memory instead of re-querying them again and again:

```
<?php

use Phalcon\Mvc\Model;

class Invoices extends Model
{
    public function initialize()
    {
        $this->belongsTo(
            "customers_id",
            "Customer",
            "id",
            [
                "reusable" => true,
            ]
        );
    }
}
```

Note that this type of cache works in memory only, this means that cached data are released when the request is terminated.

Caching Related Records

When a related record is queried, the ORM internally builds the appropriate condition and gets the required records using `find()`/`findFirst()` in the target model according to the following table:

Type	Description	Implicit Method
Belongs-To	Returns a model instance of the related record directly	<code>findFirst()</code>
Has-One	Returns a model instance of the related record directly	<code>findFirst()</code>
Has-Many	Returns a collection of model instances of the referenced model	<code>find()</code>

This means that when you get a related record you could intercept how the data is obtained by implementing the corresponding method:

```
<?php

// Get some invoice
$invoice = Invoices::findFirst();

// Get the customer related to the invoice
```

```
$customer = $invoice->customer; // Invoices::findFirst("...");

// Same as above
$customer = $invoice->getCustomer(); // Invoices::findFirst("...");
```

Accordingly, we could replace the `findFirst()` method in the `Invoices` model and implement the cache we consider most appropriate:

```
<?php

use Phalcon\Mvc\Model;

class Invoices extends Model
{
    public static function findFirst($parameters = null)
    {
        // ... Custom caching strategy
    }
}
```

Caching Related Records Recursively

In this scenario, we assume that every time we query a result we also retrieve their associated records. If we store the records found together with their related entities perhaps we could reduce a bit the overhead required to obtain all entities:

```
<?php

use Phalcon\Mvc\Model;

class Invoices extends Model
{
    protected static function _createKey($parameters)
    {
        // ... Create a cache key based on the parameters
    }

    protected static function _getCache($key)
    {
        // Returns data from a cache
    }

    protected static function _setCache($key, $results)
    {
        // Stores data in the cache
    }

    public static function find($parameters = null)
    {
        // Create a unique key
        $key = self::_createKey($parameters);

        // Check if there are data in the cache
        $results = self::_getCache($key);

        // Valid data is an object
    }
}
```

```
        if (is_object($results)) {
            return $results;
        }

        $results = [];

        $invoices = parent::find($parameters);

        foreach ($invoices as $invoice) {
            // Query the related customer
            $customer = $invoice->customer;

            // Assign it to the record
            $invoice->customer = $customer;

            $results[] = $invoice;
        }

        // Store the invoices in the cache + their customers
        self::_setCache($key, $results);

        return $results;
    }

    public function initialize()
    {
        // Add relations and initialize other stuff
    }
}
```

Getting the invoices from the cache already obtains the customer data in just one hit, reducing the overall overhead of the operation. Note that this process can also be performed with PHQL following an alternative solution:

```
<?php

use Phalcon\Mvc\Model;

class Invoices extends Model
{
    public function initialize()
    {
        // Add relations and initialize other stuff
    }

    protected static function _createKey($conditions, $params)
    {
        // ... Create a cache key based on the parameters
    }

    public function getInvoicesCustomers($conditions, $params = null)
    {
        $phql = "SELECT Invoices.*, Customers.* FROM Invoices JOIN Customers WHERE " .
        ↪ $conditions;

        $query = $this->getModelsManager()->executeQuery($phql);

        $query->cache(
            [
```

```

        "key"      => self::_createKey($conditions, $params),
        "lifetime" => 300,
    ];

    );

    return $query->execute($params);
}
}

```

Caching based on Conditions

In this scenario, the cache is implemented differently depending on the conditions received. We might decide that the cache backend should be determined by the primary key:

Type	Cache Backend
1 - 10000	mongo1
10000 - 20000	mongo2
> 20000	mongo3

The easiest way is adding a static method to the model that chooses the right cache to be used:

```

<?php

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public static function queryCache($initial, $final)
    {
        if ($initial >= 1 && $final < 10000) {
            $service = "mongo1";
        } elseif ($initial >= 10000 && $final <= 20000) {
            $service = "mongo2";
        } elseif ($initial > 20000) {
            $service = "mongo3";
        }

        return self::find(
            [
                "id >= " . $initial . " AND id <= " . $final,
                "cache" => [
                    "service" => $service,
                ],
            ]
        );
    }
}

```

This approach solves the problem, however, if we want to add other parameters such orders or conditions we would have to create a more complicated method. Additionally, this method does not work if the data is obtained using related records or a `find()`/`findFirst()`:

```

<?php

$robots = Robots::find("id < 1000");
$robots = Robots::find("id > 100 AND type = 'A'");

```

```
$robots = Robots::find("(id > 100 AND type = 'A') AND id < 2000");

$robots = Robots::find(
    [
        "(id > ?0 AND type = 'A') AND id < ?1",
        "bind" => [100, 2000],
        "order" => "type",
    ]
);
```

To achieve this we need to intercept the intermediate representation (IR) generated by the PHQL parser and thus customize the cache everything possible:

The first is create a custom builder, so we can generate a totally customized query:

```
<?php

use Phalcon\Mvc\Model\Query\Builder as QueryBuilder;

class CustomQueryBuilder extends QueryBuilder
{
    public function getQuery()
    {
        $query = new CustomQuery($this->getPhql());

        $query->setDI($this->getDI());

        return $query;
    }
}
```

Instead of directly returning a *Phalcon\Mvc\Model\Query*, our custom builder returns a CustomQuery instance, this class looks like:

```
<?php

use Phalcon\Mvc\Model\Query as ModelQuery;

class CustomQuery extends ModelQuery
{
    /**
     * The execute method is overridden
     */
    public function execute($params = null, $types = null)
    {
        // Parse the intermediate representation for the SELECT
        $ir = $this->parse();

        // Check if the query has conditions
        if (isset($ir["where"])) {
            // The fields in the conditions can have any order
            // We need to recursively check the conditions tree
            // to find the info we're looking for
            $visitor = new CustomNodeVisitor();

            // Recursively visits the nodes
            $visitor->visit($ir["where"]);
        }
    }
}
```

```

        $initial = $visitor->getInitial();
        $final   = $visitor->getFinal();

        // Select the cache according to the range
        // ...

        // Check if the cache has data
        // ...
    }

    // Execute the query
    $result = $this->_executeSelect($ir, $params, $types);

    // Cache the result
    // ...

    return $result;
}
}

```

Implementing a helper (CustomNodeVisitor) that recursively checks the conditions looking for fields that tell us the possible range to be used in the cache:

```

<?php

class CustomNodeVisitor
{
    protected $_initial = 0;

    protected $_final = 25000;

    public function visit($node)
    {
        switch ($node["type"]) {
            case "binary-op":
                $left  = $this->visit($node["left"]);
                $right = $this->visit($node["right"]);

                if (!$left || !$right) {
                    return false;
                }

                if ($left === "id") {
                    if ($node["op"] === ">") {
                        $this->_initial = $right;
                    }

                    if ($node["op"] === "=") {
                        $this->_initial = $right;
                    }

                    if ($node["op"] === ">=") {
                        $this->_initial = $right;
                    }

                    if ($node["op"] === "<") {
                        $this->_final = $right;
                    }
                }
            }
        }
    }
}

```

```
        if ($node["op"] === "<=") {
            $this->_final = $right;
        }

        break;

    case "qualified":
        if ($node["name"] === "id") {
            return "id";
        }

        break;

    case "literal":
        return $node["value"];

    default:
        return false;
    }
}

public function getInitial()
{
    return $this->_initial;
}

public function getFinal()
{
    return $this->_final;
}
}
```

Finally, we can replace the find method in the Robots model to use the custom classes we've created:

```
<?php

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public static function find($parameters = null)
    {
        if (!is_array($parameters)) {
            $parameters = [$parameters];
        }

        $builder = new CustomQueryBuilder($parameters);

        $builder->from(get_called_class());

        $query = $builder->getQuery();

        if (isset($parameters["bind"])) {
            return $query->execute($parameters["bind"]);
        } else {
            return $query->execute();
        }
    }
}
```



```

    }
}
}

```

Caching of PHQL planning

As well as most moderns database systems PHQL internally caches the execution plan, if the same statement is executed several times PHQL reuses the previously generated plan improving performance, for a developer to take better advantage of this is highly recommended build all your SQL statements passing variable parameters as bound parameters:

```

<?php

for ($i = 1; $i <= 10; $i++) {
    $phql = "SELECT * FROM Store\Robots WHERE id = " . $i;

    $robots = $this->modelsManager->executeQuery($phql);

    // ...
}

```

In the above example, ten plans were generated increasing the memory usage and processing in the application. Rewriting the code to take advantage of bound parameters reduces the processing by both ORM and database system:

```

<?php

$phql = "SELECT * FROM Store\Robots WHERE id = ?0";

for ($i = 1; $i <= 10; $i++) {
    $robots = $this->modelsManager->executeQuery(
        $phql,
        [
            $i,
        ]
    );

    // ...
}

```

Performance can be also improved reusing the PHQL query:

```

<?php

$phql = "SELECT * FROM Store\Robots WHERE id = ?0";

$query = $this->modelsManager->createQuery($phql);

for ($i = 1; $i <= 10; $i++) {
    $robots = $query->execute(
        $phql,
        [
            $i,
        ]
    );
}

```

```
// ...  
}
```

Execution plans for queries involving [prepared statements](#) are also cached by most database systems reducing the overall execution time, also protecting your application against [SQL Injections](#).

2.3.14 ODM (Object-Document Mapper)

In addition to its ability to [map tables](#) in relational databases, Phalcon can map documents from NoSQL databases. The ODM offers a CRUD functionality, events, validations among other services.

Due to the absence of SQL queries and planners, NoSQL databases can see real improvements in performance using the Phalcon approach. Additionally, there are no SQL building reducing the possibility of SQL injections.

The following NoSQL databases are supported:

Name	Description
MongoDB	MongoDB is a scalable, high-performance, open source NoSQL database.

Creating Models

A model is a class that extends from [Phalcon\Mvc\Collection](#). It must be placed in the models directory. A model file must contain a single class; its class name should be in camel case notation:

```
<?php  
  
use Phalcon\Mvc\Collection;  
  
class Robots extends Collection  
{  
  
}
```

If you're using PHP 5.4/5.5 is recommended declare each column that makes part of the model in order to save memory and reduce the memory allocation.

By default model “Robots” will refer to the collection “robots”. If you want to manually specify another name for the mapping collection, you can use the `setSource()` method:

```
<?php  
  
use Phalcon\Mvc\Collection;  
  
class Robots extends Collection  
{  
    public function initialize()  
    {  
        $this->setSource("the_robots");  
    }  
}
```

Understanding Documents To Objects

Every instance of a model represents a document in the collection. You can easily access collection data by reading object properties. For example, for a collection “robots” with the documents:

```
$ mongo test
MongoDB shell version: 1.8.2
connecting to: test
> db.robots.find()
{ "_id" : ObjectId("508735512d42b8c3d15ec4e1"), "name" : "Astro Boy", "year" : 1952,
  "type" : "mechanical" }
{ "_id" : ObjectId("5087358f2d42b8c3d15ec4e2"), "name" : "Bender", "year" : 1999,
  "type" : "mechanical" }
{ "_id" : ObjectId("508735d32d42b8c3d15ec4e3"), "name" : "Wall-E", "year" : 2008 }
>
```

Models in Namespaces

Namespaces can be used to avoid class name collision. In this case it is necessary to indicate the name of the related collection using the `setSource()` method:

```
<?php

namespace Store\Toys;

use Phalcon\Mvc\Collection;

class Robots extends Collection
{
    public function initialize()
    {
        $this->setSource("robots");
    }
}
```

You could find a certain document by its ID and then print its name:

```
<?php

// Find record with _id = "5087358f2d42b8c3d15ec4e2"
$robot = Robots::findById("5087358f2d42b8c3d15ec4e2");

// Prints "Bender"
echo $robot->name;
```

Once the record is in memory, you can make modifications to its data and then save changes:

```
<?php

$robot = Robots::findFirst(
[
    [
        "name" => "Astro Boy",
    ]
]
);

$robot->name = "Voltron";

$robot->save();
```

Setting a Connection

Connections are retrieved from the services container. By default, Phalcon tries to find the connection in a service called “mongo”:

```
<?php

// Simple database connection to localhost
$di->set(
    "mongo",
    function () {
        $mongo = new MongoClient();

        return $mongo->selectDB("store");
    },
    true
);

// Connecting to a domain socket, falling back to localhost connection
$di->set(
    "mongo",
    function () {
        $mongo = new MongoClient(
            "mongodb://tmp/mongodb-27017.sock,localhost:27017"
        );

        return $mongo->selectDB("store");
    },
    true
);
```

Finding Documents

As *Phalcon\Mvc\Collection* relies on the Mongo PHP extension you have the same facilities to query documents and convert them transparently to model instances:

```
<?php

// How many robots are there?
$robots = Robots::find();
echo "There are ", count($robots), "\n";

// How many mechanical robots are there?
$robots = Robots::find(
    [
        [
            "type" => "mechanical",
        ]
    ]
);
echo "There are ", count($robots), "\n";

// Get and print mechanical robots ordered by name upward
$robots = Robots::find(
    [
        [
            "type" => "mechanical",
```

```

        ],
        "sort" => [
            "name" => 1,
        ],
    ],
];

foreach ($robots as $robot) {
    echo $robot->name, "\n";
}

// Get first 100 mechanical robots ordered by name
$robots = Robots::find(
    [
        [
            "type" => "mechanical",
        ],
        "sort" => [
            "name" => 1,
        ],
        "limit" => 100,
    ]
);

foreach ($robots as $robot) {
    echo $robot->name, "\n";
}

```

You could also use the `findFirst()` method to get only the first record matching the given criteria:

```

<?php

// What's the first robot in robots collection?
$robot = Robots::findFirst();
echo "The robot name is ", $robot->name, "\n";

// What's the first mechanical robot in robots collection?
$robot = Robots::findFirst(
    [
        [
            "type" => "mechanical",
        ]
    ]
);
echo "The first mechanical robot name is ", $robot->name, "\n";

```

Both `find()` and `findFirst()` methods accept an associative array specifying the search criteria:

```

<?php

// First robot where type = "mechanical" and year = "1999"
$robot = Robots::findFirst(
    [
        "conditions" => [
            "type" => "mechanical",
            "year" => "1999",
        ],
    ]
);

```

```
);

// All virtual robots ordered by name downward
$robots = Robots::find(
    [
        "conditions" => [
            "type" => "virtual",
        ],
        "sort" => [
            "name" => -1,
        ],
    ]
);
```

The available query options are:

Pa- rame- ter	Description	Example
conditions	Search conditions for the find operation. Is used to extract only those records that fulfill a specified criterion. By default Phalcon_model assumes the first parameter are the conditions.	"conditions" => array('gt' => 1990)
fields	Returns specific columns instead of the full fields in the collection. When using this option an incomplete object is returned	"fields" => array('name' => true)
sort	It's used to sort the resultset. Use one or more fields as each element in the array, 1 means ordering upwards, -1 downward	"sort" => array("name" => -1, "status" => 1)
limit	Limit the results of the query to results to certain range	"limit" => 10
skip	Skips a number of results	"skip" => 50

If you have experience with SQL databases, you may want to check the [SQL to Mongo Mapping Chart](#).

Aggregations

A model can return calculations using [aggregation framework](#) provided by Mongo. The aggregated values are calculate without having to use MapReduce. With this option is easy perform tasks such as totaling or averaging field values:

```
<?php

$data = Article::aggregate(
    [
        [
            "\$project" => [
                "category" => 1,
            ],
        ],
        [
            "\$group" => [
                "_id" => [
                    "category" => "\$category"
                ],
                "id" => [
                    "\$max" => "\$_id",
                ],
            ],
        ],
    ],
);
```

```
        ],
    ]
);
```

Creating Updating/Records

The `Phalcon\Mvc\Collection::save()` method allows you to create/update documents according to whether they already exist in the collection associated with a model. The `save()` method is called internally by the create and update methods of *Phalcon\Mvc\Collection*.

Also the method executes associated validators and events that are defined in the model:

```
<?php

$robot = new Robots();

$robot->type = "mechanical";
$robot->name = "Astro Boy";
$robot->year = 1952;

if ($robot->save() === false) {
    echo "Umh, We can't store robots right now: \n";

    $messages = $robot->getMessages();

    foreach ($messages as $message) {
        echo $message, "\n";
    }
} else {
    echo "Great, a new robot was saved successfully!";
}
```

The “`_id`” property is automatically updated with the `MongoId` object created by the driver:

```
<?php

$robot->save();

echo "The generated id is: ", $robot->getId();
```

Validation Messages

Phalcon\Mvc\Collection has a messaging subsystem that provides a flexible way to output or store the validation messages generated during the insert/update processes.

Each message consists of an instance of the class *Phalcon\Mvc\Model\Message*. The set of messages generated can be retrieved with the method `getMessages()`. Each message provides extended information like the field name that generated the message or the message type:

```
<?php

if ($robot->save() === false) {
    $messages = $robot->getMessages();

    foreach ($messages as $message) {
```

```

        echo "Message: ", $message->getMessage();
        echo "Field: ", $message->getField();
        echo "Type: ", $message->getType();
    }
}

```

Validation Events and Events Manager

Models allow you to implement events that will be thrown when performing an insert or update. They help define business rules for a certain model. The following are the events supported by *Phalcon\Mvc\Collection* and their order of execution:

Operation	Name	Opération stoppée ?	Explanation
Inserting/Updating	beforeValidation	Oui	Is executed before the validation process and the final insert/update to the database
Inserting	beforeValidationOnCreate	Oui	Is executed before the validation process only when an insertion operation is being made
Updating	beforeValidationOnUpdate	Oui	Is executed before the fields are validated for not nulls or foreign keys when an updating operation is being made
Inserting/Updating	onValidationFail	Oui (already stopped)	Is executed before the validation process only when an insertion operation is being made
Inserting	afterValidationOnCreate	Oui	Is executed after the validation process when an insertion operation is being made
Updating	afterValidationOnUpdate	Oui	Is executed after the validation process when an updating operation is being made
Inserting/Updating	afterValidation	Oui	Is executed after the validation process
Inserting/Updating	beforeSave	Oui	Runs before the required operation over the database system
Updating	beforeUpdate	Oui	Runs before the required operation over the database system only when an updating operation is being made
Inserting	beforeCreate	Oui	Runs before the required operation over the database system only when an inserting operation is being made
Updating	afterUpdate	Non	Runs after the required operation over the database system only when an updating operation is being made
Inserting	afterCreate	Non	Runs after the required operation over the database system only when an inserting operation is being made
Inserting/Updating	afterSave	Non	Runs after the required operation over the database system

To make a model to react to an event, we must to implement a method with the same name of the event:

```

<?php

use Phalcon\Mvc\Collection;

class Robots extends Collection
{
    public function beforeValidationOnCreate()
    {
        echo "This is executed before creating a Robot!";
    }
}

```



```
}
}
```

Events can be useful to assign values before performing an operation, for example:

```
<?php
use Phalcon\Mvc\Collection;

class Products extends Collection
{
    public function beforeCreate()
    {
        // Set the creation date
        $this->created_at = date("Y-m-d H:i:s");
    }

    public function beforeUpdate()
    {
        // Set the modification date
        $this->modified_in = date("Y-m-d H:i:s");
    }
}
```

Additionally, this component is integrated with *Phalcon\Events\Manager*, this means we can create listeners that run when an event is triggered.

```
<?php
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;

$eventsManager = new EventsManager();

// Attach an anonymous function as a listener for "model" events
$eventsManager->attach(
    "collection:beforeSave",
    function (Event $event, $robot) {
        if ($robot->name === "Scooby Doo") {
            echo "Scooby Doo isn't a robot!";

            return false;
        }

        return true;
    }
);

$robot = new Robots();

$robot->setEventsManager($eventsManager);

$robot->name = "Scooby Doo";
$robot->year = 1969;

$robot->save();
```

In the example given above the EventsManager only acted as a bridge between an object and a listener (the anonymous

function). If we want all objects created in our application use the same EventsManager, then we need to assign this to the Models Manager:

```
<?php

use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;
use Phalcon\Mvc\Collection\Manager as CollectionManager;

// Registering the collectionManager service
$di->set(
    "collectionManager",
    function () {
        $eventsManager = new EventsManager();

        // Attach an anonymous function as a listener for "model" events
        $eventsManager->attach(
            "collection:beforeSave",
            function (Event $event, $model) {
                if (get_class($model) === "Robots") {
                    if ($model->name === "Scooby Doo") {
                        echo "Scooby Doo isn't a robot!";

                        return false;
                    }
                }

                return true;
            }
        );

        // Setting a default EventsManager
        $modelsManager = new CollectionManager();

        $modelsManager->setEventsManager($eventsManager);

        return $modelsManager;
    },
    true
);
```

Implementing a Business Rule

When an insert, update or delete is executed, the model verifies if there are any methods with the names of the events listed in the table above.

We recommend that validation methods are declared protected to prevent that business logic implementation from being exposed publicly.

The following example implements an event that validates the year cannot be smaller than 0 on update or insert:

```
<?php

use Phalcon\Mvc\Collection;

class Robots extends Collection
{
```

```

public function beforeSave()
{
    if ($this->year < 0) {
        echo "Year cannot be smaller than zero!";

        return false;
    }
}

```

Some events return false as an indication to stop the current operation. If an event doesn't return anything, *Phalcon\Mvc\Collection* will assume a true value.

Validating Data Integrity

Phalcon\Mvc\Collection provides several events to validate data and implement business rules. The special “validation” event allows us to call built-in validators over the record. Phalcon exposes a few built-in validators that can be used at this stage of validation.

The following example shows how to use it:

```

<?php

use Phalcon\Mvc\Collection;
use Phalcon\Validation;
use Phalcon\Validation\Validator\InclusionIn;
use Phalcon\Validation\Validator\Numericality;

class Robots extends Collection
{
    public function validation()
    {
        $validation = new Validation();

        $validation->add(
            "type",
            new InclusionIn(
                [
                    "message" => "Type must be: mechanical or virtual",
                    "domain" => [
                        "Mechanical",
                        "Virtual",
                    ],
                ]
            )
        );

        $validation->add(
            "price",
            new Numericality(
                [
                    "message" => "Price must be numeric"
                ]
            )
        );

        return $this->validate($validation);
    }
}

```

```
}  
}
```

The example given above performs a validation using the built-in validator “InclusionIn”. It checks the value of the field “type” in a domain list. If the value is not included in the method, then the validator will fail and return false.

For more information on validators, see the [Validation documentation](#).

Deleting Records

The `Phalcon\Mvc\Collection::delete()` method allows you to delete a document. You can use it as follows:

```
<?php  
  
$robot = Robots::findFirst();  
  
if ($robot !== false) {  
    if ($robot->delete() === false) {  
        echo "Sorry, we can't delete the robot right now: \n";  
  
        $messages = $robot->getMessages();  
  
        foreach ($messages as $message) {  
            echo $message, "\n";  
        }  
    } else {  
        echo "The robot was deleted successfully!";  
    }  
}
```

You can also delete many documents by traversing a resultset with a `foreach` loop:

```
<?php  
  
$robots = Robots::find(  
    [  
        [  
            "type" => "mechanical",  
        ],  
    ],  
);  
  
foreach ($robots as $robot) {  
    if ($robot->delete() === false) {  
        echo "Sorry, we can't delete the robot right now: \n";  
  
        $messages = $robot->getMessages();  
  
        foreach ($messages as $message) {  
            echo $message, "\n";  
        }  
    } else {  
        echo "The robot was deleted successfully!";  
    }  
}
```

The following events are available to define custom business rules that can be executed when a delete operation is performed:

Operation	Name	Opération stoppée ?	Explanation
Deleting	beforeDelete	Oui	Runs before the delete operation is made
Deleting	afterDelete	Non	Runs after the delete operation was made

Validation Failed Events

Another type of events is available when the data validation process finds any inconsistency:

Operation	Name	Explanation
Insert or Update	notSave	Triggered when the insert/update operation fails for any reason
Insert, Delete or Update	onValidationFails	Triggered when any data manipulation operation fails

Implicit Ids vs. User Primary Keys

By default *Phalcon\Mvc\Collection* assumes that the `_id` attribute is automatically generated using *MongoIds*. If a model uses custom primary keys this behavior can be overridden:

```
<?php
use Phalcon\Mvc\Collection;

class Robots extends Collection
{
    public function initialize()
    {
        $this->useImplicitObjectIds(false);
    }
}
```

Setting multiple databases

In Phalcon, all models can belong to the same database connection or have an individual one. Actually, when *Phalcon\Mvc\Collection* needs to connect to the database it requests the “mongo” service in the application’s services container. You can overwrite this service setting it in the initialize method:

```
<?php
// This service returns a mongo database at 192.168.1.100
$di->set(
    "mongo1",
    function () {
        $mongo = new MongoClient(
            "mongodb://scott:nekhen@192.168.1.100"
        );

        return $mongo->selectDB("management");
    },
    true
);
```

```
// This service returns a mongo database at localhost
$di->set(
    "mongo2",
    function () {
        $mongo = new MongoClient(
            "mongodb://localhost"
        );

        return $mongo->selectDB("invoicing");
    },
    true
);
```

Then, in the `initialize()` method, we define the connection service for the model:

```
<?php
use Phalcon\Mvc\Collection;

class Robots extends Collection
{
    public function initialize()
    {
        $this->setConnectionService("mongo1");
    }
}
```

Injecting services into Models

You may be required to access the application services within a model, the following example explains how to do that:

```
<?php
use Phalcon\Mvc\Collection;

class Robots extends Collection
{
    public function notSave()
    {
        // Obtain the flash service from the DI container
        $flash = $this->getDI()->getShared("flash");

        $messages = $this->getMessages();

        // Show validation messages
        foreach ($messages as $message) {
            $flash->error(
                (string) $message
            );
        }
    }
}
```

The “notSave” event is triggered whenever a “creating” or “updating” action fails. We’re flashing the validation messages obtaining the “flash” service from the DI container. By doing this, we don’t have to print messages after

each saving.

2.3.15 Using Views

Views represent the user interface of your application. Views are often HTML files with embedded PHP code that perform tasks related solely to the presentation of the data. Views handle the job of providing data to the web browser or other tool that is used to make requests from your application.

Phalcon\Mvc\View and *Phalcon\Mvc\View\Simple* are responsible for the managing the view layer of your MVC application.

Integrating Views with Controllers

Phalcon automatically passes the execution to the view component as soon as a particular controller has completed its cycle. The view component will look in the views folder for a folder named as the same name of the last controller executed and then for a file named as the last action executed. For instance, if a request is made to the URL *http://127.0.0.1/blog/posts/show/301*, Phalcon will parse the URL as follows:

Server Address	127.0.0.1
Phalcon Directory	blog
Controller	posts
Action	show
Parameter	301

The dispatcher will look for a “PostsController” and its action “showAction”. A simple controller file for this example:

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {

    }

    public function showAction($postId)
    {
        // Pass the $postId parameter to the view
        $this->view->postId = $postId;
    }
}
```

The `setVar()` method allows us to create view variables on demand so that they can be used in the view template. The example above demonstrates how to pass the `$postId` parameter to the respective view template.

Hierarchical Rendering

Phalcon\Mvc\View supports a hierarchy of files and is the default component for view rendering in Phalcon. This hierarchy allows for common layout points (commonly used views), as well as controller named folders defining respective view templates.

This component uses by default PHP itself as the template engine, therefore views should have the `.phtml` extension. If the views directory is *app/views* then view component will find automatically for these 3 view files.

Name	File	Description
Action View	app/views/posts/show.phtml	This is the view related to the action. It only will be shown when the “show” action was executed.
Controller Layout	app/views/layouts/posts.phtml	This is the view related to the controller. It only will be shown for every action executed within the controller “posts”. All the code implemented in the layout will be reused for all the actions in this controller.
Main Layout	app/views/index.phtml	This is main action it will be shown for every controller or action executed within the application.

You are not required to implement all of the files mentioned above. *Phalcon\Mvc\View* will simply move to the next view level in the hierarchy of files. If all three view files are implemented, they will be processed as follows:

```
<!-- app/views/posts/show.phtml -->

<h3>This is show view!</h3>

<p>I have received the parameter <?php echo $postId; ?></p>
```

```
<!-- app/views/layouts/posts.phtml -->

<h2>This is the "posts" controller layout!</h2>

<?php echo $this->getContent(); ?>
```

```
<!-- app/views/index.phtml -->
<html>
  <head>
    <title>Example</title>
  </head>
  <body>

    <h1>This is main layout!</h1>

    <?php echo $this->getContent(); ?>

  </body>
</html>
```

Note the lines where the method `$this->getContent()` was called. This method instructs *Phalcon\Mvc\View* on where to inject the contents of the previous view executed in the hierarchy. For the example above, the output will be:

The generated HTML by the request will be:

```
<!-- app/views/index.phtml -->
<html>
  <head>
    <title>Example</title>
  </head>
  <body>

    <h1>This is main layout!</h1>

    <!-- app/views/layouts/posts.phtml -->

    <h2>This is the "posts" controller layout!</h2>

    <!-- app/views/posts/show.phtml -->
```




```

    <h3>This is show view!</h3>

    <p>I have received the parameter 101</p>

  </body>
</html>

```

Using Templates

Templates are views that can be used to share common view code. They act as controller layouts, so you need to place them in the layouts directory.

Templates can be rendered before the layout (using `$this->view->setTemplateBefore()`) or they can be rendered after the layout (using `this->view->setTemplateAfter()`). In the following example the template (layouts/common.phtml) is rendered after the main layout (layouts/posts.phtml):

```

<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function initialize()
    {
        $this->view->setTemplateAfter("common");
    }
}

```

```
}

public function lastAction()
{
    $this->flash->notice(
        "These are the latest posts"
    );
}
}
```

```
<!-- app/views/index.phtml -->
<!DOCTYPE html>
<html>
    <head>
        <title>Blog's title</title>
    </head>
    <body>
        <?php echo $this->getContent(); ?>
    </body>
</html>
```

```
<!-- app/views/layouts/common.phtml -->

<ul class="menu">
    <li><a href="/">Home</a></li>
    <li><a href="/articles">Articles</a></li>
    <li><a href="/contact">Contact us</a></li>
</ul>

<div class="content"><?php echo $this->getContent(); ?></div>
```

```
<!-- app/views/layouts/posts.phtml -->

<h1>Blog Title</h1>

<?php echo $this->getContent(); ?>
```

```
<!-- app/views/posts/last.phtml -->

<article>
    <h2>This is a title</h2>
    <p>This is the post content</p>
</article>

<article>
    <h2>This is another title</h2>
    <p>This is another post content</p>
</article>
```

The final output will be the following:

```
<!-- app/views/index.phtml -->
<!DOCTYPE html>
<html>
    <head>
        <title>Blog's title</title>
```

```

</head>
<body>

    <!-- app/views/layouts/common.phtml -->

    <ul class="menu">
        <li><a href="/">Home</a></li>
        <li><a href="/articles">Articles</a></li>
        <li><a href="/contact">Contact us</a></li>
    </ul>

    <div class="content">

        <!-- app/views/layouts/posts.phtml -->

        <h1>Blog Title</h1>

        <!-- app/views/posts/last.phtml -->

        <article>
            <h2>This is a title</h2>
            <p>This is the post content</p>
        </article>

        <article>
            <h2>This is another title</h2>
            <p>This is another post content</p>
        </article>

    </div>

</body>
</html>

```

If we had used `$this->view->setTemplateBefore("common")`, this would be the final output:

```

<!-- app/views/index.phtml -->
<!DOCTYPE html>
<html>
    <head>
        <title>Blog's title</title>
    </head>
    <body>

        <!-- app/views/layouts/posts.phtml -->

        <h1>Blog Title</h1>

        <!-- app/views/layouts/common.phtml -->

        <ul class="menu">
            <li><a href="/">Home</a></li>
            <li><a href="/articles">Articles</a></li>
            <li><a href="/contact">Contact us</a></li>
        </ul>

        <div class="content">

```

```
        <!-- app/views/posts/last.phtml -->

        <article>
            <h2>This is a title</h2>
            <p>This is the post content</p>
        </article>

        <article>
            <h2>This is another title</h2>
            <p>This is another post content</p>
        </article>

    </div>

</body>
</html>
```

Control Rendering Levels

As seen above, *Phalcon\Mvc\View* supports a view hierarchy. You might need to control the level of rendering produced by the view component. The method `Phalcon\Mvc\View::setRenderLevel()` offers this functionality.

This method can be invoked from the controller or from a superior view layer to interfere with the rendering process.

```
<?php

use Phalcon\Mvc\View;
use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {

    }

    public function findAction()
    {
        // This is an Ajax response so it doesn't generate any kind of view
        $this->view->setRenderLevel(
            View::LEVEL_NO_RENDER
        );

        // ...
    }

    public function showAction($postId)
    {
        // Shows only the view related to the action
        $this->view->setRenderLevel(
            View::LEVEL_ACTION_VIEW
        );
    }
}
```

The available render levels are:

Class Constant	Description	Order
LEVEL_NO_RENDER	Indicates to avoid generating any kind of presentation.	
LEVEL_ACTION_VIEW	Generates the presentation to the view associated to the action.	1
LEVEL_BEFORE_TEMPLATE	Generates presentation templates prior to the controller layout.	2
LEVEL_LAYOUT	Generates the presentation to the controller layout.	3
LEVEL_AFTER_TEMPLATE	Generates the presentation to the templates after the controller layout.	4
LEVEL_MAIN_LAYOUT	Generates the presentation to the main layout. File views/index.phtml	5

Disabling render levels

You can permanently or temporarily disable render levels. A level could be permanently disabled if it isn't used at all in the whole application:

```
<?php
use Phalcon\Mvc\View;

$di->set(
    "view",
    function () {
        $view = new View();

        // Disable several levels
        $view->disableLevel(
            [
                View::LEVEL_LAYOUT => true,
                View::LEVEL_MAIN_LAYOUT => true,
            ]
        );

        return $view;
    },
    true
);
```

Or disable temporarily in some part of the application:

```
<?php
use Phalcon\Mvc\View;
use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {

    }

    public function findAction()
    {
```

```
$this->view->disableLevel(  
    View::LEVEL_MAIN_LAYOUT  
);  
}  
}
```

Picking Views

As mentioned above, when *Phalcon\Mvc\View* is managed by *Phalcon\Mvc\Application* the view rendered is the one related with the last controller and action executed. You could override this by using the *Phalcon\Mvc\View::pick()* method:

```
<?php  
  
use Phalcon\Mvc\Controller;  
  
class ProductsController extends Controller  
{  
    public function listAction()  
    {  
        // Pick "views-dir/products/search" as view to render  
        $this->view->pick("products/search");  
  
        // Pick "views-dir/books/list" as view to render  
        $this->view->pick(  
            [  
                "books",  
            ]  
        );  
  
        // Pick "views-dir/products/search" as view to render  
        $this->view->pick(  
            [  
                1 => "search",  
            ]  
        );  
    }  
}
```

Disabling the view

If your controller doesn't produce any output in the view (or not even have one) you may disable the view component avoiding unnecessary processing:

```
<?php  
  
use Phalcon\Mvc\Controller;  
  
class UsersController extends Controller  
{  
    public function closeSessionAction()  
    {  
        // Close session  
        // ...  
    }  
}
```

```

        // Disable the view to avoid rendering
        $this->view->disable();
    }
}

```

Alternatively, you can return `false` to produce the same effect:

```

<?php

use Phalcon\Mvc\Controller;

class UsersController extends Controller
{
    public function closeSessionAction()
    {
        // ...

        // Disable the view to avoid rendering
        return false;
    }
}

```

You can return a ‘response’ object to avoid disable the view manually:

```

<?php

use Phalcon\Mvc\Controller;

class UsersController extends Controller
{
    public function closeSessionAction()
    {
        // Close session
        // ...

        // A HTTP Redirect
        return $this->response->redirect("index/index");
    }
}

```

Simple Rendering

Phalcon\Mvc\View\Simple is an alternative component to *Phalcon\Mvc\View*. It keeps most of the philosophy of *Phalcon\Mvc\View* but lacks of a hierarchy of files which is, in fact, the main feature of its counterpart.

This component allows the developer to have control of when a view is rendered and its location. In addition, this component can leverage of view inheritance available in template engines such as *Volt* and others.

The default component must be replaced in the service container:

```

<?php

use Phalcon\Mvc\View\Simple as SimpleView;

$di->set(
    "view",

```

```
function () {  
    $view = new SimpleView();  
  
    $view->setViewsDir("../app/views/");  
  
    return $view;  
},  
true  
);
```

Automatic rendering must be disabled in *Phalcon\Mvc\Application* (if needed):

```
<?php  
  
use Exception;  
use Phalcon\Mvc\Application;  
  
try {  
    $application = new Application($di);  
  
    $application->useImplicitView(false);  
  
    $response = $application->handle();  
  
    $response->send();  
} catch (Exception $e) {  
    echo $e->getMessage();  
}
```

To render a view it's necessary to call the render method explicitly indicating the relative path to the view you want to display:

```
<?php  
  
use Phalcon\Mvc\Controller;  
  
class PostsController extends \Controller  
{  
    public function indexAction()  
    {  
        // Render 'views-dir/index.phtml'  
        echo $this->view->render("index");  
  
        // Render 'views-dir/posts/show.phtml'  
        echo $this->view->render("posts/show");  
  
        // Render 'views-dir/index.phtml' passing variables  
        echo $this->view->render(  
            "index",  
            [  
                "posts" => Posts::find(),  
            ],  
        );  
  
        // Render 'views-dir/posts/show.phtml' passing variables  
        echo $this->view->render(  
            "posts/show",  
            [  

```



```

        "posts" => Posts::find(),
    ]
    );
}
}

```

This is different to *Phalcon\Mvc\View* who's `render()` method uses controllers and actions as parameters:

```

<?php

$params = [
    "posts" => Posts::find(),
];

// Phalcon\Mvc\View
$view = new \Phalcon\Mvc\View();
echo $view->render("posts", "show", $params);

// Phalcon\Mvc\View\Simple
$simpleView = new \Phalcon\Mvc\View\Simple();
echo $simpleView->render("posts/show", $params);

```

Using Partial

Partial templates are another way of breaking the rendering process into simpler more manageable chunks that can be reused by different parts of the application. With a partial, you can move the code for rendering a particular piece of a response to its own file.

One way to use partials is to treat them as the equivalent of subroutines: as a way to move details out of a view so that your code can be more easily understood. For example, you might have a view that looks like this:

```

<div class="top"><?php $this->partial("shared/ad_banner"); ?></div>

<div class="content">
    <h1>Robots</h1>

    <p>Check out our specials for robots:</p>
    ...
</div>

<div class="footer"><?php $this->partial("shared/footer"); ?></div>

```

The `partial()` method does accept a second parameter as an array of variables/parameters that only will exists in the scope of the partial:

```

<?php $this->partial("shared/ad_banner", ["id" => $site->id, "size" => "big"]); ?>

```

Transfer values from the controller to views

Phalcon\Mvc\View is available in each controller using the view variable (`$this->view`). You can use that object to set variables directly to the view from a controller action by using the `setVar()` method.

```

<?php

use Phalcon\Mvc\Controller;

```

```
class PostsController extends Controller
{
    public function indexAction()
    {

    }

    public function showAction()
    {
        $user = Users::findFirst();
        $posts = $user->getPosts();

        // Pass all the username and the posts to the views
        $this->view->setVar("username", $user->username);
        $this->view->setVar("posts", $posts);

        // Using the magic setter
        $this->view->username = $user->username;
        $this->view->posts = $posts;

        // Passing more than one variable at the same time
        $this->view->setVars(
            [
                "username" => $user->username,
                "posts" => $posts,
            ]
        );
    }
}
```

A variable with the name of the first parameter of `setVar()` will be created in the view, ready to be used. The variable can be of any type, from a simple string, integer etc. variable to a more complex structure such as array, collection etc.

```
<h1>
    {{ username }}'s Posts
</h1>

<div class="post">
    <?php

        foreach ($posts as $post) {
            echo "<h2>", $post->title, "</h2>";
        }

    ?>
</div>
```

Caching View Fragments

Sometimes when you develop dynamic websites and some areas of them are not updated very often, the output is exactly the same between requests. *Phalcon\Mvc\View* offers caching a part or the whole rendered output to increase performance.

Phalcon\Mvc\View integrates with *Phalcon\Cache* to provide an easier way to cache output fragments. You could manually set the cache handler or set a global handler:

```

<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function showAction()
    {
        // Cache the view using the default settings
        $this->view->cache(true);
    }

    public function showArticleAction()
    {
        // Cache this view for 1 hour
        $this->view->cache(
            [
                "lifetime" => 3600,
            ]
        );
    }

    public function resumeAction()
    {
        // Cache this view for 1 day with the key "resume-cache"
        $this->view->cache(
            [
                "lifetime" => 86400,
                "key"       => "resume-cache",
            ]
        );
    }

    public function downloadAction()
    {
        // Passing a custom service
        $this->view->cache(
            [
                "service"  => "myCache",
                "lifetime" => 86400,
                "key"       => "resume-cache",
            ]
        );
    }
}

```

When we do not define a key to the cache, the component automatically creates one using an MD5 hash of the name of the controller and view currently being rendered in the format of “controller/view”. It is a good practice to define a key for each action so you can easily identify the cache associated with each view.

When the View component needs to cache something it will request a cache service from the services container. The service name convention for this service is “viewCache”:

```

<?php

use Phalcon\Cache\Frontend\Output as OutputFrontend;
use Phalcon\Cache\Backend\Memcache as MemcacheBackend;

```

```
// Set the views cache service
$di->set(
    "viewCache",
    function () {
        // Cache data for one day by default
        $frontCache = new OutputFrontend(
            [
                "lifetime" => 86400,
            ]
        );

        // Memcached connection settings
        $cache = new MemcacheBackend(
            $frontCache,
            [
                "host" => "localhost",
                "port" => "11211",
            ]
        );

        return $cache;
    }
);
```

The frontend must always be *Phalcon\Cache\Frontend\Output* and the service ‘viewCache’ must be registered as always open (not shared) in the services container (DI).

When using views, caching can be used to prevent controllers from needing to generate view data on each request.

To achieve this we must identify uniquely each cache with a key. First we verify that the cache does not exist or has expired to make the calculations/queries to display data in the view:

```
<?php

use Phalcon\Mvc\Controller;

class DownloadController extends Controller
{
    public function indexAction()
    {
        // Check whether the cache with key "downloads" exists or has expired
        if ($this->view->getCache()->exists("downloads")) {
            // Query the latest downloads
            $latest = Downloads::find(
                [
                    "order" => "created_at DESC",
                ]
            );

            $this->view->latest = $latest;
        }

        // Enable the cache with the same key "downloads"
        $this->view->cache(
            [
                "key" => "downloads",
            ]
        );
    }
};
```

```
}
}
```

The [PHP alternative site](#) is an example of implementing the caching of fragments.

Template Engines

Template Engines help designers to create views without the use of a complicated syntax. Phalcon includes a powerful and fast templating engine called *Volt*.

Additionally, *Phalcon\Mvc\View* allows you to use other template engines instead of plain PHP or Volt.

Using a different template engine, usually requires complex text parsing using external PHP libraries in order to generate the final output for the user. This usually increases the number of resources that your application will use.

If an external template engine is used, *Phalcon\Mvc\View* provides exactly the same view hierarchy and it's still possible to access the API inside these templates with a little more effort.

This component uses adapters, these help Phalcon to speak with those external template engines in a unified way, let's see how to do that integration.

Creating your own Template Engine Adapter

There are many template engines, which you might want to integrate or create one of your own. The first step to start using an external template engine is create an adapter for it.

A template engine adapter is a class that acts as bridge between *Phalcon\Mvc\View* and the template engine itself. Usually it only needs two methods implemented: `__construct()` and `render()`. The first one receives the *Phalcon\Mvc\View* instance that creates the engine adapter and the DI container used by the application.

The method `render()` accepts an absolute path to the view file and the view parameters set using `$this->view->setVar()`. You could read or require it when it's necessary.

```
<?php

use Phalcon\DiInterface;
use Phalcon\Mvc\Engine;

class MyTemplateAdapter extends Engine
{
    /**
     * Adapter constructor
     *
     * @param \Phalcon\Mvc\View $view
     * @param \Phalcon\Di $di
     */
    public function __construct($view, DiInterface $di)
    {
        // Initialize here the adapter
        parent::__construct($view, $di);
    }

    /**
     * Renders a view using the template engine
     *
     * @param string $path
     * @param array $params
     */
}
```

```
*/
public function render($path, $params)
{
    // Access view
    $view = $this->_view;

    // Access options
    $options = $this->_options;

    // Render the view
    // ...
}
```

Changing the Template Engine

You can replace the template engine completely or use more than one template engine at the same time. The method `Phalcon\Mvc\View::registerEngines()` accepts an array containing data that define the template engines. The key of each engine is an extension that aids in distinguishing one from another. Template files related to the particular engine must have those extensions.

The order that the template engines are defined with `Phalcon\Mvc\View::registerEngines()` defines the relevance of execution. If *Phalcon\Mvc\View* finds two views with the same name but different extensions, it will only render the first one.

If you want to register a template engine or a set of them for each request in the application. You could register it when the view service is created:

```
<?php

use Phalcon\Mvc\View;

// Setting up the view component
$di->set(
    "view",
    function () {
        $view = new View();

        // A trailing directory separator is required
        $view->setViewsDir("../app/views/");

        // Set the engine
        $view->registerEngines(
            [
                ".my-html" => "MyTemplateAdapter",
            ]
        );

        // Using more than one template engine
        $view->registerEngines(
            [
                ".my-html" => "MyTemplateAdapter",
                ".phtml"   => "Phalcon\\Mvc\\View\\Engine\\Php",
            ]
        );
    }
);
```

```

        return $view;
    },
    true
);

```

There are adapters available for several template engines on the [Phalcon Incubator](#)

Injecting services in View

Every view executed is included inside a *Phalcon\Di\Injectable* instance, providing easy access to the application's service container.

The following example shows how to write a jQuery [ajax request](#) using a URL with the framework conventions. The service "url" (usually *Phalcon\Mvc\Url*) is injected in the view by accessing a property with the same name:

```

<script type="text/javascript">

$.ajax({
    url: "<?php echo $this->url->get("cities/get"); ?>"
})
.done(function () {
    alert("Done!");
});

</script>

```

Stand-Alone Component

All the components in Phalcon can be used as *glue* components individually because they are loosely coupled to each other:

Hierarchical Rendering

Using *Phalcon\Mvc\View* in a stand-alone mode can be demonstrated below:

```

<?php

use Phalcon\Mvc\View;

$view = new View();

// A trailing directory separator is required
$view->setViewsDir("../app/views/");

// Passing variables to the views, these will be created as local variables
$view->setVar("someProducts", $products);
$view->setVar("someFeatureEnabled", true);

// Start the output buffering
$view->start();

// Render all the view hierarchy related to the view products/list.phtml
$view->render("products", "list");

```

```
// Finish the output buffering
$view->finish();

echo $view->getContent();
```

A short syntax is also available:

```
<?php

use Phalcon\Mvc\View;

$view = new View();

echo $view->getRender(
    "products",
    "list",
    [
        "someProducts" => $products,
        "someFeatureEnabled" => true,
    ],
    function ($view) {
        // Set any extra options here

        $view->setViewsDir("../app/views/");

        $view->setRenderLevel(
            View::LEVEL_LAYOUT
        );
    }
);
```

Simple Rendering

Using *Phalcon\Mvc\View\Simple* in a stand-alone mode can be demonstrated below:

```
<?php

use Phalcon\Mvc\View\Simple as SimpleView;

$view = new SimpleView();

// A trailing directory separator is required
$view->setViewsDir("../app/views/");

// Render a view and return its contents as a string
echo $view->render("templates/welcomeMail");

// Render a view passing parameters
echo $view->render(
    "templates/welcomeMail",
    [
        "email" => $email,
        "content" => $content,
    ]
);
```


View Events

Phalcon\Mvc\View and *Phalcon\Mvc\View\Simple* are able to send events to an *EventsManager* if it is present. Events are triggered using the type “view”. Some events when returning boolean false could stop the active operation. The following events are supported:

Nom d'évt	Triggered	Opération stoppée ?
beforeRender	Triggered before starting the render process	Oui
beforeRenderView	Triggered before rendering an existing view	Oui
afterRenderView	Triggered after rendering an existing view	Non
afterRender	Triggered after completing the render process	Non
notFoundView	Triggered when a view was not found	Non

The following example demonstrates how to attach listeners to this component:

```
<?php

use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;
use Phalcon\Mvc\View;

$di->set(
    "view",
    function () {
        // Create an events manager
        $eventsManager = new EventsManager();

        // Attach a listener for type "view"
        $eventsManager->attach(
            "view",
            function (Event $event, $view) {
                echo $event->getType(), " - ", $view->getActiveRenderPath(), PHP_EOL;
            }
        );

        $view = new View();

        $view->setViewsDir("../app/views/");

        // Bind the eventsManager to the view component
        $view->setEventsManager($eventsManager);

        return $view;
    },
    true
);
```

The following example shows how to create a plugin that clean/repair the HTML produced by the render process using Tidy:

```
<?php

use Phalcon\Events\Event;

class TidyPlugin
{
    public function afterRender(Event $event, $view)
    {
```

```
$tidyConfig = [
    "clean" => true,
    "output-xhtml" => true,
    "show-body-only" => true,
    "wrap" => 0,
];

$tidy = tidy_parse_string(
    $view->getContent(),
    $tidyConfig,
    "UTF8"
);

$tidy->cleanRepair();

$view->setContent(
    (string) $tidy
);
}

// Attach the plugin as a listener
$eventsManager->attach(
    "view:afterRender",
    new TidyPlugin()
);
```

2.3.16 View Helpers (Tags)

Writing and maintaining HTML markup can quickly become a tedious task because of the naming conventions and numerous attributes that have to be taken into consideration. Phalcon deals with this complexity by offering *Phalcon\Tag*, which in turn offers view helpers to generate HTML markup.

This component can be used in a plain HTML+PHP view or in a *Volt* template.

This guide is not intended to be a complete documentation of available helpers and their arguments. Please visit the *Phalcon\Tag* page in the API for a complete reference.

Document Type of Content

Phalcon provides `Phalcon\Tag::setDoctype()` helper to set document type of the content. Document type setting may affect HTML output produced by other tag helpers. For example, if you set XHTML document type family, helpers that return or output HTML tags will produce self-closing tags to follow valid XHTML standard.

Available document type constants in *Phalcon\Tag* namespace are:

Constant	Document type
HTML32	HTML 3.2
HTML401_STRICT	HTML 4.01 Strict
HTML401_TRANSITIONAL	HTML 4.01 Transitional
HTML401_FRAMESET	HTML 4.01 Frameset
HTML5	HTML 5
XHTML10_STRICT	XHTML 1.0 Strict
XHTML10_TRANSITIONAL	XHTML 1.0 Transitional
XHTML10_FRAMESET	XHTML 1.0 Frameset
XHTML11	XHTML 1.1
XHTML20	XHTML 2.0
XHTML5	XHTML 5

Setting document type.

```
<?php
use Phalcon\Tag;

$this->tag->setDoctype(Tag::HTML401_STRICT);

?>
```

Getting document type.

```
<?= $this->tag->getDoctype() ?>
<html>
<!-- your HTML code -->
</html>
```

The following HTML will be produced.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
<!-- your HTML code -->
</html>
```

Volt syntax:

```
{{ get_doctype() }}
<html>
<!-- your HTML code -->
</html>
```

Generating Links

A real common task in any web application or website is to produce links that allow us to navigate from one page to another. When they are internal URLs we can create them in the following manner:

```
<!-- for the default route -->
<?= $this->tag->linkTo("products/search", "Search") ?>

<!-- with CSS attributes -->
<?= $this->tag->linkTo(["products/edit/10", "Edit", "class" => "edit-btn"]) ?>
```

```
<!-- for a named route -->
<?= $this->tag->linkTo(["for" => "show-product", "title" => 123, "name" => "carrots"
↪], "Show"]) ?>
```

Actually, all produced URLs are generated by the component *Phalcon\Mvc\Url* (or service “url” failing)

Same links generated with Volt:

```
<!-- for the default route -->
{{ link_to("products/search", "Search") }}

<!-- for a named route -->
{{ link_to(["for": "show-product", "id": 123, "name": "carrots"], "Show") }}

<!-- for a named route with a HTML class -->
{{ link_to(["for": "show-product", "id": 123, "name": "carrots"], "Show", "class":
↪"edit-btn") }}
```

Creating Forms

Forms in web applications play an essential part in retrieving user input. The following example shows how to implement a simple search form using view helpers:

```
<!-- Sending the form by method POST -->
<?= $this->tag->form("products/search") ?>
    <label for="q">Search:</label>

    <?= $this->tag->textField("q") ?>

    <?= $this->tag->submitButton("Search") ?>
<?= $this->tag->endForm() ?>

<!-- Specifying another method or attributes for the FORM tag -->
<?= $this->tag->form(["products/search", "method" => "get"]); ?>
    <label for="q">Search:</label>

    <?= $this->tag->textField("q"); ?>

    <?= $this->tag->submitButton("Search"); ?>
<?= $this->tag->endForm() ?>
```

This last code will generate the following HTML:

```
<form action="/store/products/search/" method="get">
    <label for="q">Search:</label>

    <input type="text" id="q" value="" name="q" />

    <input type="submit" value="Search" />
</form>
```

Same form generated in Volt:

```
<!-- Specifying another method or attributes for the FORM tag -->
{{ form("products/search", "method": "get") }}
    <label for="q">Search:</label>
```

```

    {{ text_field("q") }}

    {{ submit_button("Search") }}
{{ endForm() }}

```

Phalcon also provides a *form builder* to create forms in an object-oriented manner.

Helpers to Generate Form Elements

Phalcon provides a series of helpers to generate form elements such as text fields, buttons and more. The first parameter of each helper is always the name of the element to be generated. When the form is submitted, the name will be passed along with the form data. In a controller you can get these values using the same name by using the `getPost()` and `getQuery()` methods on the request object (`$this->request`).

```

<?php echo $this->tag->textField("username") ?>

<?php echo $this->tag->textArea(
    [
        "comment",
        "This is the content of the text-area",
        "cols" => "6",
        "rows" => 20,
    ]
) ?>

<?php echo $this->tag->passwordField(
    [
        "password",
        "size" => 30,
    ]
) ?>

<?php echo $this->tag->hiddenField(
    [
        "parent_id",
        "value" => "5",
    ]
) ?>

```

Volt syntax:

```

{{ text_field("username") }}

{{ text_area("comment", "This is the content", "cols": "6", "rows": 20) }}

{{ password_field("password", "size": 30) }}

{{ hidden_field("parent_id", "value": "5") }}

```

Making Select Boxes

Generating select boxes (select box) is easy, especially if the related data is stored in PHP associative arrays. The helpers for select elements are `Phalcon\Tag::select()` and `Phalcon\Tag::selectStatic()`. `Phalcon\Tag::select()` has been specifically designed to work with *Phalcon\Mvc\Model*, while `Phalcon\Tag::selectStatic()` can work with PHP arrays.

```
<?php

$products = Products::find("type = 'vegetables'");

// Using data from a resultset
echo $this->tag->select(
    [
        "productId",
        $products,
        "using" => [
            "id",
            "name",
        ]
    ]
);

// Using data from an array
echo $this->tag->selectStatic(
    [
        "status",
        [
            "A" => "Active",
            "I" => "Inactive",
        ]
    ]
);
```

The following HTML will generated:

```
<select id="productId" name="productId">
  <option value="101">Tomato</option>
  <option value="102">Lettuce</option>
  <option value="103">Beans</option>
</select>

<select id="status" name="status">
  <option value="A">Active</option>
  <option value="I">Inactive</option>
</select>
```

You can add an “empty” option to the generated HTML:

```
<?php

$products = Products::find("type = 'vegetables'");

// Creating a Select Tag with an empty option
echo $this->tag->select(
    [
        "productId",
        $products,
        "using" => [
            "id",
            "name",
        ],
        "useEmpty" => true,
    ]
);
```

Produces this HTML:

```
<select id="productId" name="productId">
  <option value="">Choose..</option>
  <option value="101">Tomato</option>
  <option value="102">Lettuce</option>
  <option value="103">Beans</option>
</select>
```

```
<?php

$products = Products::find("type = 'vegetables'");

// Creating a Select Tag with an empty option with default text
echo $this->tag->select(
    [
        "productId",
        $products,
        "using"      => [
            "id",
            "name",
        ],
        "useEmpty"   => true,
        "emptyText"  => "Please, choose one...",
        "emptyValue" => "@",
    ]
);
```

```
<select id="productId" name="productId">
  <option value="@">Please, choose one..</option>
  <option value="101">Tomato</option>
  <option value="102">Lettuce</option>
  <option value="103">Beans</option>
</select>
```

Volt syntax for above example:

```
{# Creating a Select Tag with an empty option with default text #}
{{ select('productId', products, 'using': ['id', 'name'],
    'useEmpty': true, 'emptyText': 'Please, choose one...', 'emptyValue': '@') }}
```

Assigning HTML attributes

All the helpers accept an array as their first parameter which can contain additional HTML attributes for the element generated.

```
<?php $this->tag->textField(
    [
        "price",
        "size"      => 20,
        "maxlength" => 30,
        "placeholder" => "Enter a price",
    ]
) ?>
```

or using Volt:

```
{{ text_field("price", "size": 20, "maxlength": 30, "placeholder": "Enter a price") }}
```

The following HTML is generated:

```
<input type="text" name="price" id="price" size="20" maxlength="30"
      placeholder="Enter a price" />
```

Setting Helper Values

From Controllers

It is a good programming principle for MVC frameworks to set specific values for form elements in the view. You can set those values directly from the controller using `Phalcon\Tag::setDefault()`. This helper preloads a value for any helpers present in the view. If any helper in the view has a name that matches the preloaded value, it will use it, unless a value is directly assigned on the helper in the view.

```
<?php
use Phalcon\Mvc\Controller;

class ProductsController extends Controller
{
    public function indexAction()
    {
        $this->tag->setDefault("color", "Blue");
    }
}
```

At the view, a `selectStatic` helper matches the same index used to preset the value. In this case “color”:

```
<?php
echo $this->tag->selectStatic(
    [
        "color",
        [
            "Yellow" => "Yellow",
            "Blue"   => "Blue",
            "Red"    => "Red",
        ]
    ]
);
```

This will generate the following select tag with the value “Blue” selected:

```
<select id="color" name="color">
    <option value="Yellow">Yellow</option>
    <option value="Blue" selected="selected">Blue</option>
    <option value="Red">Red</option>
</select>
```


From the Request

A special feature that the *Phalcon\Tag* helpers have is that they keep the values of form helpers between requests. This way you can easily show validation messages without losing entered data.

Specifying values directly

Every form helper supports the parameter “value”. With it you can specify a value for the helper directly. When this parameter is present, any preset value using `setDefault()` or via request will be ignored.

Changing dynamically the Document Title

Phalcon\Tag offers helpers to change dynamically the document title from the controller. The following example demonstrates just that:

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function initialize()
    {
        $this->tag->setTitle("Your Website");
    }

    public function indexAction()
    {
        $this->tag->prependTitle("Index of Posts - ");
    }
}
```

```
<html>
  <head>
    <?php echo $this->tag->getTitle(); ?>
  </head>

  <body>

  </body>
</html>
```

The following HTML will generated:

```
<html>
  <head>
    <title>Index of Posts - Your Website</title>
  </head>

  <body>

  </body>
</html>
```

Static Content Helpers

Phalcon\Tag also provide helpers to generate tags such as script, link or img. They aid in quick and easy generation of the static resources of your application

Images

```
<?php

// Generate 
echo $this->tag->image("img/hello.gif");

// Generate 
echo $this->tag->image(
    [
        "img/hello.gif",
        "alt" => "alternative text",
    ]
);
```

Volt syntax:

```
{# Generate  #}
{{ image("img/hello.gif") }}

{# Generate  #}
{{ image("img/hello.gif", "alt": "alternative text") }}
```

Stylesheets

```
<?php

// Generate <link rel="stylesheet" href="http://fonts.googleapis.com/css?
↳family=Rosario" type="text/css">
echo $this->tag->stylesheetLink("http://fonts.googleapis.com/css?family=Rosario",
↳false);

// Generate <link rel="stylesheet" href="/your-app/css/styles.css" type="text/css">
echo $this->tag->stylesheetLink("css/styles.css");
```

Volt syntax:

```
{# Generate <link rel="stylesheet" href="http://fonts.googleapis.com/css?
↳family=Rosario" type="text/css"> #}
{{ stylesheet_link("http://fonts.googleapis.com/css?family=Rosario", false) }}

{# Generate <link rel="stylesheet" href="/your-app/css/styles.css" type="text/css"> #}
{{ stylesheet_link("css/styles.css") }}
```

Javascript

```
<?php

// Generate <script src="http://localhost/javascript/jquery.min.js" type="text/
↪javascript"></script>
echo $this->tag->javascriptInclude("http://localhost/javascript/jquery.min.js", ↪
↪false);

// Generate <script src="/your-app/javascript/jquery.min.js" type="text/javascript"></
↪script>
echo $this->tag->javascriptInclude("javascript/jquery.min.js");
```

Volt syntax:

```
{# Generate <script src="http://localhost/javascript/jquery.min.js" type="text/
↪javascript"></script> #}
{{ javascript_include("http://localhost/javascript/jquery.min.js", false) }}

{# Generate <script src="/your-app/javascript/jquery.min.js" type="text/javascript"></
↪script> #}
{{ javascript_include("javascript/jquery.min.js") }}
```

HTML5 elements - generic HTML helper

Phalcon offers a generic HTML helper that allows the generation of any kind of HTML element. It is up to the developer to produce a valid HTML element name to the helper.

```
<?php

// Generate
// <canvas id="canvas1" width="300" class="cnvclass">
// This is my canvas
// </canvas>
echo $this->tag->tagHtml("canvas", ["id" => "canvas1", "width" => "300", "class" =>
↪"cnvclass"], false, true, true);
echo "This is my canvas";
echo $this->tag->tagHtmlClose("canvas");
```

Volt syntax:

```
{# Generate
<canvas id="canvas1" width="300" class="cnvclass">
This is my canvas
</canvas> #}
{{ tag_html("canvas", ["id": "canvas1", "width": "300", "class": "cnvclass"], false, ↪
↪true, true) }}
    This is my canvas
{{ tag_html_close("canvas") }}
```

Tag Service

Phalcon\Tag is available via the ‘tag’ service, this means you can access it from any part of the application where the services container is located:

```
<?php echo $this->tag->linkTo("pages/about", "About") ?>
```

You can easily add new helpers to a custom component replacing the service ‘tag’ in the services container:

```
<?php

use Phalcon\Tag;

class MyTags extends Tag
{
    // ...

    // Create a new helper
    public static function myAmazingHelper($parameters)
    {
        // ...
    }

    // Override an existing method
    public static function textField($parameters)
    {
        // ...
    }
}
```

Then change the definition of the service ‘tag’:

```
<?php

$di["tag"] = function () {
    return new MyTags();
};
```

Creating your own helpers

You can easily create your own helpers. First, start by creating a new folder within the same directory as your controllers and models. Give it a title that is relative to what you are creating. For our example here, we can call it “customhelpers”. Next we will create a new file titled `MyTags.php` within this new directory. At this point, we have a structure that looks similar to : `/app/customhelpers/MyTags.php`. In `MyTags.php`, we will extend the *Phalcon\Tag* and implement your own helper. Below is a simple example of a custom helper:

```
<?php

use Phalcon\Tag;

class MyTags extends Tag
{
    /**
     * Generates a widget to show a HTML5 audio tag
     *
     * @param array
     * @return string
     */
    public static function audioField($parameters)
    {
        // Converting parameters to array if it is not
```

```

        if (!is_array($parameters)) {
            $parameters = [$parameters];
        }

        // Determining attributes "id" and "name"
        if (!isset($parameters[0])) {
            $parameters[0] = $parameters["id"];
        }

        $id = $parameters[0];

        if (!isset($parameters["name"])) {
            $parameters["name"] = $id;
        } else {
            if (!$parameters["name"]) {
                $parameters["name"] = $id;
            }
        }

        // Determining widget value,
        // \Phalcon\Tag::setDefault() allows to set the widget value
        if (isset($parameters["value"])) {
            $value = $parameters["value"];

            unset($parameters["value"]);
        } else {
            $value = self::getValue($id);
        }

        // Generate the tag code
        $code = '<audio id="' . $id . '" value="' . $value . '" ';

        foreach ($parameters as $key => $attributeValue) {
            if (!is_integer($key)) {
                $code .= $key . '=' . $attributeValue . ' ';
            }
        }

        $code .= ">";

        return $code;
    }
}

```

After creating our custom helper, we will autoload the new directory that contains our helper class from our “index.php” located in the public directory.

```

<?php

use Phalcon\Loader;
use Phalcon\Mvc\Application;
use Phalcon\Di\FactoryDefault();
use Phalcon\Exception as PhalconException;

try {
    $loader = new Loader();

    $loader->registerDirs(

```

```
[
    "../app/controllers",
    "../app/models",
    "../app/customhelpers", // Add the new helpers folder
]
);

$loader->register();

$di = new FactoryDefault();

// Assign our new tag a definition so we can call it
$di->set(
    "MyTags",
    function () {
        return new MyTags();
    }
);

$app = new Application($di);

$response = $app->handle();

$response->send();
} catch (PhalconException $e) {
    echo "PhalconException: ", $e->getMessage();
}
```

Now you are ready to use your new helper within your views:

```
<body>

    <?php

    echo MyTags::audioField(
        [
            "name" => "test",
            "id"   => "audio_test",
            "src"  => "/path/to/audio.mp3",
        ]
    );

    ?>

</body>
```

In next chapter, we'll talk about *Volt* a faster template engine for PHP, where you can use a more friendly syntax for using helpers provided by *Phalcon\Tag*.

2.3.17 Assets Management

Phalcon\Assets is a component that allows you to manage static resources such as CSS stylesheets or JavaScript libraries in a web application.

Phalcon\Assets\Manager is available in the services container, so you can add resources from any part of the application where the container is available.

Adding Resources

Assets supports two built-in resources: CSS and JavaScripts. You can create other resources if you need. The assets manager internally stores two default collections of resources - one for JavaScript and another for CSS.

You can easily add resources to these collections like follows:

```
<?php
use Phalcon\Mvc\Controller;

class IndexController extends Controller
{
    public function index()
    {
        // Add some local CSS resources
        $this->assets->addCss("css/style.css");
        $this->assets->addCss("css/index.css");

        // And some local JavaScript resources
        $this->assets->addJs("js/jquery.js");
        $this->assets->addJs("js/bootstrap.min.js");
    }
}
```

Then in a view, these resources can be printed:

```
<html>
  <head>
    <title>Some amazing website</title>

    <?php $this->assets->outputCss(); ?>
  </head>

  <body>
    <!-- ... -->

    <?php $this->assets->outputJs(); ?>
  </body>
</html>
```

Volt syntax:

```
<html>
  <head>
    <title>Some amazing website</title>

    {{ assets.outputCss() }}
  </head>

  <body>
    <!-- ... -->

    {{ assets.outputJs() }}
  </body>
</html>
```

For better pageload performance, it is recommended to place JavaScript at the end of the HTML instead of in the <head>.

Local/Remote resources

Local resources are those who are provided by the same application and they're located in the document root of the application. URLs in local resources are generated by the 'url' service, usually *Phalcon\Mvc\Url*.

Remote resources are those such as common libraries like jQuery, Bootstrap, etc. that are provided by a CDN.

The second parameter of `addCss()` and `addJs()` says whether the resource is local or not (`true` is local, `false` is remote). By default, the assets manager will assume the resource is local:

```
<?php

public function indexAction()
{
    // Add some local CSS resources
    $this->assets->addCss("//netdna.bootstrapcdn.com/twitter-bootstrap/2.3.1/css/
    ↪bootstrap-combined.min.css", false);
    $this->assets->addCss("css/style.css", true);
    $this->assets->addCss("css/extra.css");
}
```

Collections

Collections group resources of the same type. The assets manager implicitly creates two collections: `css` and `js`. You can create additional collections to group specific resources to make it easier to place those resources in the views:

```
<?php

// Javascripts in the header
$headerCollection = $this->assets->collection("header");

$headerCollection->addJs("js/jquery.js");
$headerCollection->addJs("js/bootstrap.min.js");

// Javascripts in the footer
$footerCollection = $this->assets->collection("footer");

$footerCollection->addJs("js/jquery.js");
$footerCollection->addJs("js/bootstrap.min.js");
```

Then in the views:

```
<html>
  <head>
    <title>Some amazing website</title>

    <?php $this->assets->outputJs("header"); ?>
  </head>

  <body>
    <!-- ... -->

    <?php $this->assets->outputJs("footer"); ?>
  </body>
</html>
```

Volt syntax:


```
<html>
  <head>
    <title>Some amazing website</title>

    {{ assets.outputCss("header") }}
  </head>

  <body>
    <!-- ... -->

    {{ assets.outputJs("footer") }}
  </body>
</html>
```

URL Prefixes

Collections can be URL-prefixed, this enables you to easily change from one server to another at any moment:

```
<?php

$footerCollection = $this->assets->collection("footer");

if ($config->environment === "development") {
    $footerCollection->setPrefix("/");
} else {
    $footerCollection->setPrefix("http://cdn.example.com/");
}

$footerCollection->addJs("js/jquery.js");
$footerCollection->addJs("js/bootstrap.min.js");
```

A chainable syntax is available too:

```
<?php

$headerCollection = $assets
    ->collection("header")
    ->setPrefix("http://cdn.example.com/")
    ->setLocal(false)
    ->addJs("js/jquery.js")
    ->addJs("js/bootstrap.min.js");
```

Minification/Filtering

Phalcon\Assets provides built-in minification of JavaScript and CSS resources. You can create a collection of resources instructing the Assets Manager which ones must be filtered and which ones must be left as they are. In addition to the above, Jsmin by Douglas Crockford is part of the core extension offering minification of JavaScript files for maximum performance. In the CSS land, CSSMin by Ryan Day is also available to minify CSS files:

The following example shows how to minify a collection of resources:

```
<?php

$manager
```

```
// These JavaScripts are located in the page's bottom
->collection("jsFooter")

// The name of the final output
->setTargetPath("final.js")

// The script tag is generated with this URI
->setTargetUri("production/final.js")

// This is a remote resource that does not need filtering
->addJs("code.jquery.com/jquery-1.10.0.min.js", false, false)

// These are local resources that must be filtered
->addJs("common-functions.js")
->addJs("page-functions.js")

// Join all the resources in a single file
->join(true)

// Use the built-in Jsmin filter
->addFilter(
    new Phalcon\Assets\Filters\Jsmin()
)

// Use a custom filter
->addFilter(
    new MyApp\Assets\Filters\LicenseStamper()
);
```

A collection can contain JavaScript or CSS resources but not both. Some resources may be remote, that is, they're obtained by HTTP from a remote source for further filtering. It is recommended to convert the external resources to local for better performance.

As seen above, the `addJs()` method is used to add resources to the collection, the second parameter indicates whether the resource is external or not and the third parameter indicates whether the resource should be filtered or left as is:

```
<?php

// These Javascripts are located in the page's bottom
$jsFooterCollection = $manager->collection("jsFooter");

// This a remote resource that does not need filtering
$jsFooterCollection->addJs("code.jquery.com/jquery-1.10.0.min.js", false, false);

// These are local resources that must be filtered
$jsFooterCollection->addJs("common-functions.js");
$jsFooterCollection->addJs("page-functions.js");
```

Filters are registered in the collection, multiple filters are allowed, content in resources are filtered in the same order as filters were registered:

```
<?php

// Use the built-in Jsmin filter
$jsFooterCollection->addFilter(
    new Phalcon\Assets\Filters\Jsmin()
);
```

```
// Use a custom filter
$jsFooterCollection->addFilter(
    new MyApp\Assets\Filters\LicenseStamper()
);
```

Note that both built-in and custom filters can be transparently applied to collections. The last step is to decide if all the resources in the collection must be joined into a single file or serve each of them individually. To tell the collection that all resources must be joined you can use the `join()` method.

If resources are going to be joined, we need also to define which file will be used to store the resources and which URI will be used to show it. These settings are set up with `setTargetPath()` and `setTargetUri()`:

```
<?php

$jsFooterCollection->join(true);

// The name of the final file path
$jsFooterCollection->setTargetPath("public/production/final.js");

// The script HTML tag is generated with this URI
$jsFooterCollection->setTargetUri("production/final.js");
```

If resources are going to be joined, we need also to define which file will be used to store the resources and which URI will be used to show it. These settings are set up with `setTargetPath()` and `setTargetUri()`.

Built-In Filters

Phalcon provides 2 built-in filters to minify both JavaScript and CSS, their C-backend provide the minimum overhead to perform this task:

Filter	Description
<i>Phalcon\Assets\Filters\Jsmin</i>	Minifies JavaScript by removing unnecessary characters that are ignored by Javascript interpreters/compilers
<i>Phalcon\Assets\Filters\Cssmin</i>	Minifies CSS by removing unnecessary characters that are already ignored by browsers

Custom Filters

In addition to the built-in filters, you can create your own filters. These can take advantage of existing and more advanced tools like [YUI](#), [Sass](#), [Closure](#), etc.:

```
<?php

use Phalcon\Assets\FilterInterface;

/**
 * Filters CSS content using YUI
 *
 * @param string $contents
 * @return string
 */
class CssYUICompressor implements FilterInterface
{
    protected $_options;
```

```
/**
 * CssYUICompressor constructor
 *
 * @param array $options
 */
public function __construct(array $options)
{
    $this->_options = $options;
}

/**
 * Do the filtering
 *
 * @param string $contents
 *
 * @return string
 */
public function filter($contents)
{
    // Write the string contents into a temporal file
    file_put_contents("temp/my-temp-1.css", $contents);

    system(
        $this->_options["java-bin"] .
        " -jar " .
        $this->_options["yui"] .
        " --type css " .
        "temp/my-temp-file-1.css " .
        $this->_options["extra-options"] .
        " -o temp/my-temp-file-2.css"
    );

    // Return the contents of file
    return file_get_contents("temp/my-temp-file-2.css");
}
}
```

Usage:

```
<?php

// Get some CSS collection
$css = $this->assets->get("head");

// Add/Enable the YUI compressor filter in the collection
$css->addFilter(
    new CssYUICompressor(
        [
            "java-bin"      => "/usr/local/bin/java",
            "yui"           => "/some/path/yuicompressor-x.y.z.jar",
            "extra-options" => "--charset utf8",
        ]
    )
);
```

In a previous example, we used a custom filter called `LicenseStamper`:

```

<?php

use Phalcon\Assets\FILTERInterface;

/**
 * Adds a license message to the top of the file
 *
 * @param string $contents
 *
 * @return string
 */
class LicenseStamper implements FILTERInterface
{
    /**
     * Do the filtering
     *
     * @param string $contents
     * @return string
     */
    public function filter($contents)
    {
        $license = "/* (c) 2015 Your Name Here */";

        return $license . PHP_EOL . PHP_EOL . $contents;
    }
}

```

Custom Output

The `outputJs()` and `outputCss()` methods are available to generate the necessary HTML code according to each type of resources. You can override this method or print the resources manually in the following way:

```

<?php

use Phalcon\Tag;

$jsCollection = $this->assets->collection("js");

foreach ($jsCollection as $resource) {
    echo Tag::javascriptInclude(
        $resource->getPath()
    );
}

```

2.3.18 Volt: Template Engine

Volt is an ultra-fast and designer friendly templating language written in C for PHP. It provides you a set of helpers to write views in an easy way. Volt is highly integrated with other components of Phalcon, just as you can use it as a stand-alone component in your applications.

Volt is inspired by [Jinja](#), originally created by [Armin Ronacher](#). Therefore many developers will be in familiar territory using the same syntax they have been using with similar template engines. Volt's syntax and features have been enhanced with more elements and of course with the performance that developers have been accustomed to while working with Phalcon.



Introduction

Volt views are compiled to pure PHP code, so basically they save the effort of writing PHP code manually:

```

{# app/views/products/show.volt #}

{% block last_products %}

{% for product in products %}
    * Name: {{ product.name|e }}
    {% if product.status == "Active" %}
        Price: {{ product.price + product.taxes/100 }}
    {% endif %}
{% endfor %}

{% endblock %}

```

Activating Volt

As with other templating engines, you may register Volt in the view component, using a new extension or reusing the standard .phtml:

```

<?php

use Phalcon\Mvc\View;
use Phalcon\Mvc\View\Engine\Volt;

// Register Volt as a service
$di->set(

```

```

    "voltService",
    function ($view, $di) {
        $volt = new Volt($view, $di);

        $volt->setOptions(
            [
                "compiledPath"      => "../app/compiled-templates/",
                "compiledExtension" => ".compiled",
            ]
        );

        return $volt;
    }
);

// Register Volt as template engine
$di->set(
    "view",
    function () {
        $view = new View();

        $view->setViewsDir("../app/views/");

        $view->registerEngines(
            [
                ".volt" => "voltService",
            ]
        );

        return $view;
    }
);

```

Use the standard ".phtml" extension:

```

<?php

$view->registerEngines(
    [
        ".phtml" => "voltService",
    ]
);

```

You don't have to specify the Volt Service in the DI; you can also use the Volt engine with the default settings:

```

<?php

$view->registerEngines(
    [
        ".volt" => "Phalcon\\Mvc\\View\\Engine\\Volt",
    ]
);

```

If you do not want to reuse Volt as a service, you can pass an anonymous function to register the engine instead of a service name:

```

<?php

```

```
use Phalcon\Mvc\View;
use Phalcon\Mvc\View\Engine\Volt;

// Register Volt as template engine with an anonymous function
$di->set(
    "view",
    function () {
        $view = new \Phalcon\Mvc\View();

        $view->setViewsDir("../app/views/");

        $view->registerEngines(
            [
                ".volt" => function ($view, $di) {
                    $volt = new Volt($view, $di);

                    // Set some options here

                    return $volt;
                }
            ]
        );

        return $view;
    }
);
```

The following options are available in Volt:

Option	Description	De- fault
compiledPath	A writable path where the compiled PHP templates will be placed	./
compiledExtension	An additional extension appended to the compiled PHP file	.php
compiledSeparator	Volt replaces the directory separators / and \ by this separator in order to create a single file in the compiled directory	%%
stat	Whether Phalcon must check if exists differences between the template file and its compiled path	true
compileAlways	Tell Volt if the templates must be compiled in each request or only when they change	false
prefix	Allows to prepend a prefix to the templates in the compilation path	null
autoescape	Enables globally autoescape of HTML	false

The compilation path is generated according to the above options, if the developer wants total freedom defining the compilation path, an anonymous function can be used to generate it, this function receives the relative path to the template in the views directory. The following examples show how to change the compilation path dynamically:

```
<?php

// Just append the .php extension to the template path
// leaving the compiled templates in the same directory
$volt->setOptions(
    [
        "compiledPath" => function ($templatePath) {
            return $templatePath . ".php";
        }
    ]
);
```



```
// Recursively create the same structure in another directory
$volt->setOptions(
[
    "compiledPath" => function ($templatePath) {
        $dirName = dirname($templatePath);

        if (!is_dir("cache/" . $dirName)) {
            mkdir("cache/" . $dirName);
        }

        return "cache/" . $dirName . "/" . $templatePath . ".php";
    }
]
);
```

Basic Usage

A view consists of Volt code, PHP and HTML. A set of special delimiters is available to enter into Volt mode. `{% ... %}` is used to execute statements such as for-loops or assign values and `{{ ... }}`, prints the result of an expression to the template.

Below is a minimal template that illustrates a few basics:

```
{# app/views/posts/show.phtml #}
<!DOCTYPE html>
<html>
    <head>
        <title>{{ title }} - An example blog</title>
    </head>
    <body>

        {% if show_navigation %}
            <ul id="navigation">
                {% for item in menu %}
                    <li>
                        <a href="{{ item.href }}">
                            {{ item.caption }}
                        </a>
                    </li>
                {% endfor %}
            </ul>
        {% endif %}

        <h1>{{ post.title }}</h1>

        <div class="content">
            {{ post.content }}
        </div>

    </body>
</html>
```

Using *Phalcon\Mvc\View* you can pass variables from the controller to the views. In the above example, four variables were passed to the view: `show_navigation`, `menu`, `title` and `post`:

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function showAction()
    {
        $post = Post::findFirst();
        $menu = Menu::findFirst();

        $this->view->show_navigation = true;
        $this->view->menu              = $menu;
        $this->view->title             = $post->title;
        $this->view->post              = $post;

        // Or...

        $this->view->setVar("show_navigation", true);
        $this->view->setVar("menu",           $menu);
        $this->view->setVar("title",          $post->title);
        $this->view->setVar("post",           $post);
    }
}
```

Variables

Object variables may have attributes which can be accessed using the syntax: `foo.bar`. If you are passing arrays, you have to use the square bracket syntax: `foo['bar']`

```
{{ post.title }} {# for $post->title #}
{{ post['title'] }} {# for $post['title'] #}
```

Filters

Variables can be formatted or modified using filters. The pipe operator `|` is used to apply filters to variables:

```
{{ post.title|e }}
{{ post.content|striptags }}
{{ name|capitalize|trim }}
```

The following is the list of available built-in filters in Volt:

Filter	Description
e	Applies <code>Phalcon\Escaper->escapeHtml()</code> to the value
escape	Applies <code>Phalcon\Escaper->escapeHtml()</code> to the value
escape_css	Applies <code>Phalcon\Escaper->escapeCss()</code> to the value
escape_js	Applies <code>Phalcon\Escaper->escapeJs()</code> to the value
escape_attr	Applies <code>Phalcon\Escaper->escapeHtmlAttr()</code> to the value
trim	Applies the <code>trim</code> PHP function to the value. Removing extra spaces
left_trim	Applies the <code>ltrim</code> PHP function to the value. Removing extra spaces
right_trim	Applies the <code>rtrim</code> PHP function to the value. Removing extra spaces
striptags	Applies the <code>striptags</code> PHP function to the value. Removing HTML tags
slashes	Applies the <code>slashes</code> PHP function to the value. Escaping values
stripslashes	Applies the <code>stripslashes</code> PHP function to the value. Removing escaped quotes
capitalize	Capitalizes a string by applying the <code>ucwords</code> PHP function to the value
lower	Change the case of a string to lowercase
upper	Change the case of a string to uppercase
length	Counts the string length or how many items are in an array or object
nl2br	Changes newlines <code>\n</code> by line breaks (<code>
</code>). Uses the PHP function <code>nl2br</code>
sort	Sorts an array using the PHP function <code>asort</code>
keys	Returns the array keys using <code>array_keys</code>
join	Joins the array parts using a separator <code>join</code>
format	Formats a string using <code>sprintf</code> .
json_encode	Converts a value into its JSON representation
json_decode	Converts a value from its JSON representation to a PHP representation
abs	Applies the <code>abs</code> PHP function to a value.
url_encode	Applies the <code>urlencode</code> PHP function to the value
default	Sets a default value in case that the evaluated expression is empty (is not set or evaluates to a falsy value)
convert_encoding	Converts a string from one charset to another

Examples:

```
{# e or escape filter #}
{{ "<h1>Hello<h1>"|e }}
{{ "<h1>Hello<h1>"|escape }}
```

```
{# trim filter #}
{{ "  hello  "|trim }}
```

```
{# striptags filter #}
{{ "<h1>Hello<h1>"|striptags }}
```

```
{# slashes filter #}
{{ "'this is a string'"|slashes }}
```

```
{# stripslashes filter #}
{{ "'this is a string'"|stripslashes }}
```

```
{# capitalize filter #}
{{ "hello"|capitalize }}
```

```
{# lower filter #}
{{ "HELLO"|lower }}
```

```
{# upper filter #}
{{ "hello"|upper }}
```

```
{# length filter #}
{{ "robots"|length }}
{{ [1, 2, 3]|length }}

{# nl2br filter #}
{{ "some\ntext"|nl2br }}

{# sort filter #}
{% set sorted = [3, 1, 2]|sort %}

{# keys filter #}
{% set keys = ['first': 1, 'second': 2, 'third': 3]|keys %}

{# join filter #}
{% set joined = "a".. "z"|join(",") %}

{# format filter #}
{{ "My real name is %s"|format(name) }}

{# json_encode filter #}
{% set encoded = robots|json_encode %}

{# json_decode filter #}
{% set decoded = '{"one":1,"two":2,"three":3}'|json_decode %}

{# url_encode filter #}
{{ post.permanent_link|url_encode }}

{# convert_encoding filter #}
{{ "désolé"|convert_encoding('utf8', 'latin1') }}
```

Comments

Comments may also be added to a template using the `{# ... #}` delimiters. All text inside them is just ignored in the final output:

```
{# note: this is a comment
    {% set price = 100; %}
#}
```

List of Control Structures

Volt provides a set of basic but powerful control structures for use in templates:

For

Loop over each item in a sequence. The following example shows how to traverse a set of “robots” and print his/her name:

```
<h1>Robots</h1>
<ul>
    {% for robot in robots %}
        <li>
```

```

        {{ robot.name|e }}
    </li>
    {% endfor %}
</ul>

```

for-loops can also be nested:

```

<h1>Robots</h1>
{% for robot in robots %}
    {% for part in robot.parts %}
        Robot: {{ robot.name|e }} Part: {{ part.name|e }} <br />
    {% endfor %}
{% endfor %}

```

You can get the element “keys” as in the PHP counterpart using the following syntax:

```

{% set numbers = ['one': 1, 'two': 2, 'three': 3] %}

{% for name, value in numbers %}
    Name: {{ name }} Value: {{ value }}
{% endfor %}

```

An “if” evaluation can be optionally set:

```

{% set numbers = ['one': 1, 'two': 2, 'three': 3] %}

{% for value in numbers if value < 2 %}
    Value: {{ value }}
{% endfor %}

{% for name, value in numbers if name != 'two' %}
    Name: {{ name }} Value: {{ value }}
{% endfor %}

```

If an ‘else’ is defined inside the ‘for’, it will be executed if the expression in the iterator result in zero iterations:

```

<h1>Robots</h1>
{% for robot in robots %}
    Robot: {{ robot.name|e }} Part: {{ part.name|e }} <br />
{% else %}
    There are no robots to show
{% endfor %}

```

Alternative syntax:

```

<h1>Robots</h1>
{% for robot in robots %}
    Robot: {{ robot.name|e }} Part: {{ part.name|e }} <br />
{% elseif %}
    There are no robots to show
{% endfor %}

```

Loop Controls

The ‘break’ and ‘continue’ statements can be used to exit from a loop or force an iteration in the current block:

```
{# skip the even robots #}
{% for index, robot in robots %}
    {% if index is even %}
        {% continue %}
    {% endif %}
    ...
{% endfor %}
```

```
{# exit the foreach on the first even robot #}
{% for index, robot in robots %}
    {% if index is even %}
        {% break %}
    {% endif %}
    ...
{% endfor %}
```

If

As PHP, an “if” statement checks if an expression is evaluated as true or false:

```
<h1>Cyborg Robots</h1>
<ul>
    {% for robot in robots %}
        {% if robot.type === "cyborg" %}
            <li>{{ robot.name|e }}</li>
        {% endif %}
    {% endfor %}
</ul>
```

The else clause is also supported:

```
<h1>Robots</h1>
<ul>
    {% for robot in robots %}
        {% if robot.type === "cyborg" %}
            <li>{{ robot.name|e }}</li>
        {% else %}
            <li>{{ robot.name|e }} (not a cyborg)</li>
        {% endif %}
    {% endfor %}
</ul>
```

The ‘elseif’ control flow structure can be used together with if to emulate a ‘switch’ block:

```
{% if robot.type === "cyborg" %}
    Robot is a cyborg
{% elseif robot.type === "virtual" %}
    Robot is virtual
{% elseif robot.type === "mechanical" %}
    Robot is mechanical
{% endif %}
```

Loop Context

A special variable is available inside ‘for’ loops providing you information about

Variable	Description
<code>loop.index</code>	The current iteration of the loop. (1 indexed)
<code>loop.index0</code>	The current iteration of the loop. (0 indexed)
<code>loop.revindex</code>	The number of iterations from the end of the loop (1 indexed)
<code>loop.revindex0</code>	The number of iterations from the end of the loop (0 indexed)
<code>loop.first</code>	True if in the first iteration.
<code>loop.last</code>	True if in the last iteration.
<code>loop.length</code>	The number of items to iterate

```
{% for robot in robots %}
    {% if loop.first %}
        <table>
            <tr>
                <th>#</th>
                <th>Id</th>
                <th>Name</th>
            </tr>
        {% endif %}
        <tr>
            <td>{{ loop.index }}</td>
            <td>{{ robot.id }}</td>
            <td>{{ robot.name }}</td>
        </tr>
    {% if loop.last %}
        </table>
    {% endif %}
{% endfor %}
```

Assignments

Variables may be changed in a template using the instruction “set”:

```
{% set fruits = ['Apple', 'Banana', 'Orange'] %}

{% set name = robot.name %}
```

Multiple assignments are allowed in the same instruction:

```
{% set fruits = ['Apple', 'Banana', 'Orange'], name = robot.name, active = true %}
```

Additionally, you can use compound assignment operators:

```
{% set price += 100.00 %}

{% set age *= 5 %}
```

The following operators are available:

Operator	Description
=	Standard Assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment

Expressions

Volt provides a basic set of expression support, including literals and common operators.

A expression can be evaluated and printed using the '{{' and '}}' delimiters:

```
{{ (1 + 1) * 2 }}
```

If an expression needs to be evaluated without be printed the 'do' statement can be used:

```
{% do (1 + 1) * 2 %}
```

Literals

The following literals are supported:

Filter	Description
"this is a string"	Text between double quotes or single quotes are handled as strings
100.25	Numbers with a decimal part are handled as doubles/floats
100	Numbers without a decimal part are handled as integers
false	Constant "false" is the boolean false value
true	Constant "true" is the boolean true value
null	Constant "null" is the Null value

Arrays

Whether you're using PHP 5.3 or >= 5.4 you can create arrays by enclosing a list of values in square brackets:

```
{# Simple array #}  
{{ ['Apple', 'Banana', 'Orange'] }}
```

```
{# Other simple array #}  
{{ ['Apple', 1, 2.5, false, null] }}
```

```
{# Multi-Dimensional array #}  
{{ [[1, 2], [3, 4], [5, 6]] }}
```

```
{# Hash-style array #}  
{{ ['first': 1, 'second': 4/2, 'third': '3'] }}
```

Curly braces also can be used to define arrays or hashes:

```
{% set myArray = {'Apple', 'Banana', 'Orange'} %}  
{% set myHash = {'first': 1, 'second': 4/2, 'third': '3'} %}
```


Math

You may make calculations in templates using the following operators:

Operator	Description
+	Perform an adding operation. <code>{{ 2 + 3 }}</code> returns 5
-	Perform a subtraction operation <code>{{ 2 - 3 }}</code> returns -1
*	Perform a multiplication operation <code>{{ 2 * 3 }}</code> returns 6
/	Perform a division operation <code>{{ 10 / 2 }}</code> returns 5
%	Calculate the remainder of an integer division <code>{{ 10 % 3 }}</code> returns 1

Comparisons

The following comparison operators are available:

Operator	Description
==	Check whether both operands are equal
!=	Check whether both operands aren't equal
<>	Check whether both operands aren't equal
>	Check whether left operand is greater than right operand
<	Check whether left operand is less than right operand
<=	Check whether left operand is less or equal than right operand
>=	Check whether left operand is greater or equal than right operand
===	Check whether both operands are identical
!==	Check whether both operands aren't identical

Logic

Logic operators are useful in the “if” expression evaluation to combine multiple tests:

Operator	Description
or	Return true if the left or right operand is evaluated as true
and	Return true if both left and right operands are evaluated as true
not	Negates an expression
(expr)	Parenthesis groups expressions

Other Operators

Additional operators seen the following operators are available:

Operator	Description
~	Concatenates both operands <code>{{ "hello " ~ "world" }}</code>
	Applies a filter in the right operand to the left <code>{{ "hello" uppercase }}</code>
..	Creates a range <code>{{ 'a'..'z' }}</code> <code>{{ 1..10 }}</code>
is	Same as == (equals), also performs tests
in	To check if an expression is contained into other expressions if "a" in "abc"
is not	Same as != (not equals)
'a' ? 'b' : 'c'	Ternary operator. The same as the PHP ternary operator
++	Increments a value
--	Decrements a value

The following example shows how to use operators:

```
{% set robots = ['Voltron', 'Astro Boy', 'Terminator', 'C3PO'] %}

{% for index in 0..robots|length %}
    {% if robots[index] is defined %}
        {{ "Name: " ~ robots[index] }}
    {% endif %}
{% endfor %}
```

Tests

Tests can be used to test if a variable has a valid expected value. The operator “is” is used to perform the tests:

```
{% set robots = ['1': 'Voltron', '2': 'Astro Boy', '3': 'Terminator', '4': 'C3PO'] %}

{% for position, name in robots %}
    {% if position is odd %}
        {{ name }}
    {% endif %}
{% endfor %}
```

The following built-in tests are available in Volt:

Test	Description
defined	Checks if a variable is defined (<code>isset()</code>)
empty	Checks if a variable is empty
even	Checks if a numeric value is even
odd	Checks if a numeric value is odd
numeric	Checks if value is numeric
scalar	Checks if value is scalar (not an array or object)
iterable	Checks if a value is iterable. Can be traversed by a “for” statement
divisibleby	Checks if a value is divisible by other value
sameas	Checks if a value is identical to other value
type	Checks if a value is of the specified type

More examples:

```
{% if robot is defined %}
    The robot variable is defined
{% endif %}

{% if robot is empty %}
    The robot is null or isn't defined
{% endif %}

{% for key, name in [1: 'Voltron', 2: 'Astro Boy', 3: 'Bender'] %}
    {% if key is even %}
        {{ name }}
    {% endif %}
{% endfor %}

{% for key, name in [1: 'Voltron', 2: 'Astro Boy', 3: 'Bender'] %}
    {% if key is odd %}
        {{ name }}
    {% endif %}
{% endfor %}
```

```
{% for key, name in [1: 'Voltron', 2: 'Astroy Boy', 'third': 'Bender'] %}
    {% if key is numeric %}
        {{ name }}
    {% endif %}
{% endfor %}

{% set robots = [1: 'Voltron', 2: 'Astroy Boy'] %}
{% if robots is iterable %}
    {% for robot in robots %}
        ...
    {% endfor %}
{% endif %}

{% set world = "hello" %}
{% if world is sameas("hello") %}
    {{ "it's hello" }}
{% endif %}

{% set external = false %}
{% if external is type('boolean') %}
    {{ "external is false or true" }}
{% endif %}
```

Macros

Macros can be used to reuse logic in a template, they act as PHP functions, can receive parameters and return values:

```
{# Macro "display a list of links to related topics" #}
{%- macro related_bar(related_links) %}
    <ul>
        {%- for link in related_links %}
            <li>
                <a href="{{ url(link.url) }}" title="{{ link.title|striptags }}">
                    {{ link.text }}
                </a>
            </li>
        {%- endfor %}
    </ul>
{%- endmacro %}

{# Print related links #}
{{ related_bar(links) }}

<div>This is the content</div>

{# Print related links again #}
{{ related_bar(links) }}
```

When calling macros, parameters can be passed by name:

```
{%- macro error_messages(message, field, type) %}
    <div>
        <span class="error-type">{{ type }}</span>
        <span class="error-field">{{ field }}</span>
        <span class="error-message">{{ message }}</span>
    </div>
```

```
{%- endmacro %}

{# Call the macro #}
{{ error_messages('type': 'Invalid', 'message': 'The name is invalid', 'field': 'name'
→) }}
}}
```

Macros can return values:

```
{%- macro my_input(name, class) %}
    {% return text_field(name, 'class': class) %}
{%- endmacro %}

{# Call the macro #}
{{ '<p>' ~ my_input('name', 'input-text') ~ '</p>' }}
```

And receive optional parameters:

```
{%- macro my_input(name, class="input-text") %}
    {% return text_field(name, 'class': class) %}
{%- endmacro %}

{# Call the macro #}
{{ '<p>' ~ my_input('name') ~ '</p>' }}
{{ '<p>' ~ my_input('name', 'input-text') ~ '</p>' }}
```

Using Tag Helpers

Volt is highly integrated with *Phalcon\Tag*, so it's easy to use the helpers provided by that component in a Volt template:

```
{{ javascript_include("js/jquery.js") }}

{{ form('products/save', 'method': 'post') }}

    <label for="name">Name</label>
    {{ text_field("name", "size": 32) }}

    <label for="type">Type</label>
    {{ select("type", productTypes, 'using': ['id', 'name']) }}

    {{ submit_button('Send') }}

{{ end_form() }}
```

The following PHP is generated:

```
<?php echo Phalcon\Tag::javascriptInclude("js/jquery.js") ?>

<?php echo Phalcon\Tag::form(array('products/save', 'method' => 'post')); ?>

    <label for="name">Name</label>
    <?php echo Phalcon\Tag::textField(array('name', 'size' => 32)); ?>

    <label for="type">Type</label>
    <?php echo Phalcon\Tag::select(array('type', $productTypes, 'using' => array('id',
→ 'name'))); ?>
```

```
<?php echo Phalcon\Tag::submitButton('Send'); ?>

{{ end_form() }}
```

To call a *Phalcon\Tag* helper, you only need to call an uncamelized version of the method:

Method	Volt function
Phalcon\Tag::linkTo	link_to
Phalcon\Tag::textField	text_field
Phalcon\Tag::passwordField	password_field
Phalcon\Tag::hiddenField	hidden_field
Phalcon\Tag::fileField	file_field
Phalcon\Tag::checkField	check_field
Phalcon\Tag::radioField	radio_field
Phalcon\Tag::dateField	date_field
Phalcon\Tag::emailField	email_field
Phalcon\Tag::numericField	numeric_field
Phalcon\Tag::submitButton	submit_button
Phalcon\Tag::selectStatic	select_static
Phalcon\Tag::select	select
Phalcon\Tag::textArea	text_area
Phalcon\Tag::form	form
Phalcon\Tag::endForm	end_form
Phalcon\Tag::getTitle	get_title
Phalcon\Tag::stylesheetLink	stylesheet_link
Phalcon\Tag::javascriptInclude	javascript_include
Phalcon\Tag::image	image
Phalcon\Tag::friendlyTitle	friendly_title

Functions

The following built-in functions are available in Volt:

Name	Description
content	Includes the content produced in a previous rendering stage
get_content	Same as content
partial	Dynamically loads a partial view in the current template
super	Render the contents of the parent block
time	Calls the PHP function with the same name
date	Calls the PHP function with the same name
dump	Calls the PHP function <code>var_dump()</code>
version	Returns the current version of the framework
constant	Reads a PHP constant
url	Generate a URL using the 'url' service

View Integration

Also, Volt is integrated with *Phalcon\Mvc\View*, you can play with the view hierarchy and include partials as well:

```
{{ content() }}
```

```
<!-- Simple include of a partial -->
<div id="footer">{{ partial("partials/footer") }}</div>

<!-- Passing extra variables -->
<div id="footer">{{ partial("partials/footer", ['links': links]) }}</div>
```

A partial is included in runtime, Volt also provides “include”, this compiles the content of a view and returns its contents as part of the view which was included:

```
{# Simple include of a partial #}
<div id="footer">
    {% include "partials/footer" %}
</div>

{# Passing extra variables #}
<div id="footer">
    {% include "partials/footer" with ['links': links] %}
</div>
```

Include

‘include’ has a special behavior that will help us improve performance a bit when using Volt, if you specify the extension when including the file and it exists when the template is compiled, Volt can inline the contents of the template in the parent template where it’s included. Templates aren’t inlined if the ‘include’ have variables passed with ‘with’:

```
{# The contents of 'partials/footer.volt' is compiled and inlined #}
<div id="footer">
    {% include "partials/footer.volt" %}
</div>
```

Partial vs Include

Keep the following points in mind when choosing to use the “partial” function or “include”:

- ‘Partial’ allows you to include templates made in Volt and in other template engines as well
- ‘Partial’ allows you to pass an expression like a variable allowing to include the content of other view dynamically
- ‘Partial’ is better if the content that you have to include changes frequently
- ‘Include’ copies the compiled content into the view which improves the performance
- ‘Include’ only allows to include templates made with Volt
- ‘Include’ requires an existing template at compile time

Template Inheritance

With template inheritance you can create base templates that can be extended by others templates allowing to reuse code. A base template define *blocks* than can be overridden by a child template. Let’s pretend that we have the following base template:

```
{# templates/base.volt #}
<!DOCTYPE html>
<html>
  <head>
    {% block head %}
      <link rel="stylesheet" href="style.css" />
    {% endblock %}

    <title>{% block title %}{% endblock %} - My Webpage</title>
  </head>

  <body>
    <div id="content">{% block content %}{% endblock %}</div>

    <div id="footer">
      {% block footer %}&copy; Copyright 2015, All rights reserved.{% endblock
→ %}
    </div>
  </body>
</html>
```

From other template we could extend the base template replacing the blocks:

```
{% extends "templates/base.volt" %}

{% block title %}Index{% endblock %}

{% block head %}<style type="text/css">.important { color: #336699; }</style>{%
→ endblock %}

{% block content %}
  <h1>Index</h1>
  <p class="important">Welcome on my awesome homepage.</p>
{% endblock %}
```

Not all blocks must be replaced at a child template, only those that are needed. The final output produced will be the following:

```
<!DOCTYPE html>
<html>
  <head>
    <style type="text/css">.important { color: #336699; }</style>

    <title>Index - My Webpage</title>
  </head>

  <body>
    <div id="content">
      <h1>Index</h1>
      <p class="important">Welcome on my awesome homepage.</p>
    </div>

    <div id="footer">
      &copy; Copyright 2015, All rights reserved.
    </div>
  </body>
</html>
```

Multiple Inheritance

Extended templates can extend other templates. The following example illustrates this:

```
{# main.volt #}  
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Title</title>  
  </head>  
  
  <body>  
    {% block content %}{% endblock %}  
  </body>  
</html>
```

Template “layout.volt” extends “main.volt”

```
{# layout.volt #}  
{% extends "main.volt" %}  
  
{% block content %}  
  
  <h1>Table of contents</h1>  
  
{% endblock %}
```

Finally a view that extends “layout.volt”:

```
{# index.volt #}  
{% extends "layout.volt" %}  
  
{% block content %}  
  
  {{ super() }}  
  
  <ul>  
    <li>Some option</li>  
    <li>Some other option</li>  
  </ul>  
  
{% endblock %}
```

Rendering “index.volt” produces:

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Title</title>  
  </head>  
  
  <body>  
  
    <h1>Table of contents</h1>  
  
    <ul>  
      <li>Some option</li>  
      <li>Some other option</li>
```



```

        </ul>

    </body>
</html>

```

Note the call to the function `super()`. With that function it's possible to render the contents of the parent block.

As partials, the path set to “extends” is a relative path under the current views directory (i.e. `app/views/`).

By default, and for performance reasons, Volt only checks for changes in the children templates to know when to re-compile to plain PHP again, so it is recommended initialize Volt with the option `'compileAlways' => true`. Thus, the templates are compiled always taking into account changes in the parent templates.

Autoescape mode

You can enable auto-escaping of all variables printed in a block using the autoescape mode:

```

Manually escaped: {{ robot.name|e }}

{% autoescape true %}
    Autoescaped: {{ robot.name }}
{% autoescape false %}
    No Autoescaped: {{ robot.name }}
{% endautoescape %}
{% endautoescape %}

```

Extending Volt

Unlike other template engines, Volt itself is not required to run the compiled templates. Once the templates are compiled there is no dependence on Volt. With performance independence in mind, Volt only acts as a compiler for PHP templates.

The Volt compiler allow you to extend it adding more functions, tests or filters to the existing ones.

Functions

Functions act as normal PHP functions, a valid string name is required as function name. Functions can be added using two strategies, returning a simple string or using an anonymous function. Always is required that the chosen strategy returns a valid PHP string expression:

```

<?php

use Phalcon\Mvc\View\Engine\Volt;

$volt = new Volt($view, $di);

$compiler = $volt->getCompiler();

// This binds the function name 'shuffle' in Volt to the PHP function 'str_shuffle'
$compiler->addFunction("shuffle", "str_shuffle");

```

Register the function with an anonymous function. This case we use `$resolvedArgs` to pass the arguments exactly as were passed in the arguments:

```
<?php

$compiler->addFunction(
    "widget",
    function ($resolvedArgs, $exprArgs) {
        return "MyLibrary\\Widgets::get(" . $resolvedArgs . ")";
    }
);
```

Treat the arguments independently and unresolved:

```
<?php

$compiler->addFunction(
    "repeat",
    function ($resolvedArgs, $exprArgs) use ($compiler) {
        // Resolve the first argument
        $firstArgument = $compiler->expression($exprArgs[0]['expr']);

        // Checks if the second argument was passed
        if (isset($exprArgs[1])) {
            $secondArgument = $compiler->expression($exprArgs[1]['expr']);
        } else {
            // Use '10' as default
            $secondArgument = '10';
        }

        return "str_repeat(" . $firstArgument . ", " . $secondArgument . ")";
    }
);
```

Generate the code based on some function availability:

```
<?php

$compiler->addFunction(
    "contains_text",
    function ($resolvedArgs, $exprArgs) {
        if (function_exists("mb_stripos")) {
            return "mb_stripos(" . $resolvedArgs . ")";
        } else {
            return "stripos(" . $resolvedArgs . ")";
        }
    }
);
```

Built-in functions can be overridden adding a function with its name:

```
<?php

// Replace built-in function dump
$compiler->addFunction("dump", "print_r");
```

Filters

A filter has the following form in a template: `leftExprName(optional-args)`. Adding new filters is similar as seen with the functions:

```
<?php

// This creates a filter 'hash' that uses the PHP function 'md5'
$compiler->addFilter("hash", "md5");
```

```
<?php

$compiler->addFilter(
    "int",
    function ($resolvedArgs, $exprArgs) {
        return "intval(" . $resolvedArgs . ")";
    }
);
```

Built-in filters can be overridden adding a function with its name:

```
<?php

// Replace built-in filter 'capitalize'
$compiler->addFilter("capitalize", "lcfirst");
```

Extensions

With extensions the developer has more flexibility to extend the template engine, and override the compilation of a specific instruction, change the behavior of an expression or operator, add functions/filters, and more.

An extension is a class that implements the events triggered by Volt as a method of itself.

For example, the class below allows to use any PHP function in Volt:

```
<?php

class PhpFunctionExtension
{
    /**
     * This method is called on any attempt to compile a function call
     */
    public function compileFunction($name, $arguments)
    {
        if (function_exists($name)) {
            return $name . "(" . $arguments . ")";
        }
    }
}
```

The above class implements the method ‘`compileFunction`’ which is invoked before any attempt to compile a function call in any template. The purpose of the extension is to verify if a function to be compiled is a PHP function allowing to call it from the template. Events in extensions must return valid PHP code, this will be used as result of the compilation instead of the one generated by Volt. If an event doesn’t return a string the compilation is done using the default behavior provided by the engine.

The following compilation events are available to be implemented in extensions:

Event/Method	Description
compileFunction	Triggered before trying to compile any function call in a template
compileFilter	Triggered before trying to compile any filter call in a template
resolveExpression	Triggered before trying to compile any expression. This allows the developer to override operators
compileStatement	Triggered before trying to compile any expression. This allows the developer to override any statement

Volt extensions must be in registered in the compiler making them available in compile time:

```
<?php

// Register the extension in the compiler
$compiler->addExtension(
    new PhpFunctionExtension()
);
```

Caching view fragments

With Volt it's easy cache view fragments. This caching improves performance preventing that the contents of a block from being executed by PHP each time the view is displayed:

```
{% cache "sidebar" %}
    <!-- generate this content is slow so we are going to cache it -->
{% endcache %}
```

Setting a specific number of seconds:

```
{# cache the sidebar by 1 hour #}
{% cache "sidebar" 3600 %}
    <!-- generate this content is slow so we are going to cache it -->
{% endcache %}
```

Any valid expression can be used as cache key:

```
{% cache ("article-" ~ post.id) 3600 %}

    <h1>{{ post.title }}</h1>

    <p>{{ post.content }}</p>

{% endcache %}
```

The caching is done by the *Phalcon\Cache* component via the view component. Learn more about how this integration works in the section “*Caching View Fragments*”.

Inject Services into a Template

If a service container (DI) is available for Volt, you can use the services by only accessing the name of the service in the template:

```
{# Inject the 'flash' service #}
<div id="messages">{{ flash.output() }}</div>

{# Inject the 'security' service #}
<input type="hidden" name="token" value="{{ security.getToken() }}">
```

Stand-alone component

Using Volt in a stand-alone mode can be demonstrated below:

```
<?php

use Phalcon\Mvc\View\Engine\Volt\Compiler as VoltCompiler;

// Create a compiler
$compiler = new VoltCompiler();

// Optionally add some options
$compiler->setOptions(
    [
        // ...
    ]
);

// Compile a template string returning PHP code
echo $compiler->compileString(
    "{{ 'hello' }}"
);

// Compile a template in a file specifying the destination file
$compiler->compileFile(
    "layouts/main.volt",
    "cache/layouts/main.volt.php"
);

// Compile a template in a file based on the options passed to the compiler
$compiler->compile(
    "layouts/main.volt"
);

// Require the compiled templated (optional)
require $compiler->getCompiledTemplatePath();
```

External Resources

- A bundle for Sublime/Textmate is available [here](#)
- Album-O-Rama is a sample application using Volt as template engine, [[Album-O-Rama on Github](#)]
- Our website is running using Volt as template engine, [[Our website on Github](#)]
- Phosphorum, the Phalcon's forum, also uses Volt, [[Phosphorum on Github](#)]
- Vökuró, is another sample application that use Volt, [[Vökuró on Github](#)]

2.3.19 MVC Applications

All the hard work behind orchestrating the operation of MVC in Phalcon is normally done by *Phalcon\MvcApplication*. This component encapsulates all the complex operations required in the background, instantiating every component needed and integrating it with the project, to allow the MVC pattern to operate as desired.

Single or Multi Module Applications

With this component you can run various types of MVC structures:

Single Module

Single MVC applications consist of one module only. Namespaces can be used but are not necessary. An application like this would have the following file structure:

```
single/  
  app/  
    controllers/  
    models/  
    views/  
  public/  
    css/  
    img/  
    js/
```

If namespaces are not used, the following bootstrap file could be used to orchestrate the MVC flow:

```
<?php  
  
use Phalcon\Loader;  
use Phalcon\Mvc\View;  
use Phalcon\Mvc\Application;  
use Phalcon\Di\FactoryDefault;  
  
$loader = new Loader();  
  
$loader->registerDirs(  
    [  
        "../apps/controllers/",  
        "../apps/models/",  
    ]  
);  
  
$loader->register();  
  
$di = new FactoryDefault();  
  
// Registering the view component  
$di->set(  
    "view",  
    function () {  
        $view = new View();  
  
        $view->setViewsDir("../apps/views/");  
  
        return $view;  
    }  
);  
  
$application = new Application($di);  
  
try {  
    $response = $application->handle();
```

```

        $response->send();
    } catch (\Exception $e) {
        echo $e->getMessage();
    }

```

If namespaces are used, the following bootstrap can be used:

```

<?php

use Phalcon\Loader;
use Phalcon\Mvc\View;
use Phalcon\Mvc\Dispatcher;
use Phalcon\Mvc\Application;
use Phalcon\Di\FactoryDefault;

$loader = new Loader();

// Use autoloading with namespaces prefixes
$loader->registerNamespaces(
    [
        "Single\\Controllers" => "../apps/controllers/",
        "Single\\Models"      => "../apps/models/",
    ]
);

$loader->register();

$di = new FactoryDefault();

// Register the default dispatcher's namespace for controllers
$di->set(
    "dispatcher",
    function () {
        $dispatcher = new Dispatcher();

        $dispatcher->setDefaultNamespace("Single\\Controllers");

        return $dispatcher;
    }
);

// Register the view component
$di->set(
    "view",
    function () {
        $view = new View();

        $view->setViewsDir("../apps/views/");

        return $view;
    }
);

$application = new Application($di);

try {
    $response = $application->handle();
}

```

```
$response->send();
} catch (\Exception $e) {
    echo $e->getMessage();
}
```

Multi Module

A multi-module application uses the same document root for more than one module. In this case the following file structure can be used:

```
multiple/
  apps/
    frontend/
      controllers/
      models/
      views/
      Module.php
    backend/
      controllers/
      models/
      views/
      Module.php
  public/
    css/
    img/
    js/
```

Each directory in apps/ have its own MVC structure. A Module.php is present to configure specific settings of each module like autoloaders or custom services:

```
<?php

namespace Multiple\Backend;

use Phalcon\Loader;
use Phalcon\Mvc\View;
use Phalcon\DiInterface;
use Phalcon\Mvc\Dispatcher;
use Phalcon\Mvc\ModuleDefinitionInterface;

class Module implements ModuleDefinitionInterface
{
    /**
     * Register a specific autoloader for the module
     */
    public function registerAutoloaders(DiInterface $di = null)
    {
        $loader = new Loader();

        $loader->registerNamespaces(
            [
                "Multiple\\Backend\\Controllers" => "../apps/backend/controllers/",
                "Multiple\\Backend\\Models"       => "../apps/backend/models/",
            ]
        );
    }
}
```



```

        $loader->register();
    }

    /**
     * Register specific services for the module
     */
    public function registerServices(DiInterface $di)
    {
        // Registering a dispatcher
        $di->set(
            "dispatcher",
            function () {
                $dispatcher = new Dispatcher();

                $dispatcher->setDefaultNamespace("Multiple\\Backend\\Controllers");

                return $dispatcher;
            }
        );

        // Registering the view component
        $di->set(
            "view",
            function () {
                $view = new View();

                $view->setViewsDir("../apps/backend/views/");

                return $view;
            }
        );
    }
}

```

A special bootstrap file is required to load a multi-module MVC architecture:

```

<?php

use Phalcon\Mvc\Router;
use Phalcon\Mvc\Application;
use Phalcon\Di\FactoryDefault;

$di = new FactoryDefault();

// Specify routes for modules
// More information how to set the router up https://docs.phalconphp.com/fr/latest/reference/routing.html
$di->set(
    "router",
    function () {
        $router = new Router();

        $router->setDefaultModule("frontend");

        $router->add(
            "/login",
            [

```

```
        "module"      => "backend",
        "controller" => "login",
        "action"     => "index",
    ]
);

$router->add(
    "/admin/products/:action",
    [
        "module"      => "backend",
        "controller" => "products",
        "action"     => 1,
    ]
);

$router->add(
    "/products/:action",
    [
        "controller" => "products",
        "action"     => 1,
    ]
);

return $router;
}
);

// Create an application
$application = new Application($di);

// Register the installed modules
$application->registerModules(
    [
        "frontend" => [
            "className" => "Multiple\\Frontend\\Module",
            "path"      => "../apps/frontend/Module.php",
        ],
        "backend" => [
            "className" => "Multiple\\Backend\\Module",
            "path"      => "../apps/backend/Module.php",
        ]
    ]
);

try {
    // Handle the request
    $response = $application->handle();

    $response->send();
} catch (\Exception $e) {
    echo $e->getMessage();
}
```

If you want to maintain the module configuration in the bootstrap file you can use an anonymous function to register the module:

```
<?php
```

```

use Phalcon\Mvc\View;

// Creating a view component
$view = new View();

// Set options to view component
// ...

// Register the installed modules
$application->registerModules(
    [
        "frontend" => function ($di) use ($view) {
            $di->setShared(
                "view",
                function () use ($view) {
                    $view->setViewsDir("../apps/frontend/views/");

                    return $view;
                }
            );
        },
        "backend" => function ($di) use ($view) {
            $di->setShared(
                "view",
                function () use ($view) {
                    $view->setViewsDir("../apps/backend/views/");

                    return $view;
                }
            );
        }
    ]
);

```

When *Phalcon\Mvc\Application* have modules registered, always is necessary that every matched route returns a valid module. Each registered module has an associated class offering functions to set the module itself up. Each module class definition must implement two methods: *registerAutoloaders()* and *registerServices()*, they will be called by *Phalcon\Mvc\Application* according to the module to be executed.

Application Events

Phalcon\Mvc\Application is able to send events to the *EventsManager* (if it is present). Events are triggered using the type “application”. The following events are supported:

Nom d'évt	Triggered
boot	Executed when the application handles its first request
beforeStartModule	Before initialize a module, only when modules are registered
afterStartModule	After initialize a module, only when modules are registered
beforeHandleRequest	Before execute the dispatch loop
afterHandleRequest	After execute the dispatch loop

The following example demonstrates how to attach listeners to this component:

```

<?php

use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;

```

```
$eventsManager = new EventsManager();

$application->setEventsManager($eventsManager);

$eventsManager->attach(
    "application",
    function (Event $event, $application) {
        // ...
    }
);
```

External Resources

- [MVC examples on Github](#)

2.3.20 Routage

Le composant routeur vous permet de définir des routes qui correspondent à des contrôleurs ou des gestionnaires qui doivent recevoir la requête. Un routeur analyse l'URI pour extraire cette information. Le routeur dispose de deux modes: MVC, et correspondance seulement (match-only). Le premier mode est idéal pour travailler sur des applications MVC.

Définir des Routes

Phalcon\Mvc\Router fournit des possibilités de routage avancées. En mode MVC vous pouvez définir des routes et les faire correspondre à des contrôleurs ou des actions dont vous avez besoin. Une route est définie comme suit:

```
<?php

use Phalcon\Mvc\Router;

// Création du routeur
$router = new Router();

// Définition d'une route
$router->add(
    "/admin/users/my-profile",
    [
        "controller" => "users",
        "action"      => "profile",
    ]
);

// Une autre route
$router->add(
    "/admin/users/change-password",
    [
        "controller" => "users",
        "action"      => "changePassword",
    ]
);

$router->handle();
```

Le premier paramètre de la méthode `add()` est le motif recherché et, optionnellement, le second paramètre est un ensemble de chemins. Dans ce cas, si l'URI est `/admin/users/my-profile`, alors l'action "profile" du contrôleur "users" sera exécutée. Il faut se rappeler que le routeur n'exécute pas l'action du contrôleur, il récupère uniquement cette information pour en informer le bon composant (par ex. *Phalcon\Mvc\Dispatcher*) que c'est ce contrôleur ou cette action qui doit être exécutée.

Définir les routes une à une d'une application qui possède plusieurs chemins peut être une tâche pénible. Pour ces cas nous pouvons créer des routes plus flexibles:

```
<?php

use Phalcon\Mvc\Router;

// Création du routeur
$routeur = new Router();

// Définition de route
$routeur->add(
    "/admin/:controller/a/:action/:params",
    [
        "controller" => 1,
        "action"      => 2,
        "params"      => 3,
    ]
);
```

Dans l'exemple précédent nous utilisons des jokers pour rendre la route valide pour plusieurs URIs. Par exemple, cette URL (`/admin/users/a/delete/dave/301`) pourrait produire:

Contrôleur	users
Action	delete
Paramètre	dave
Paramètre	301

La méthode `add()` reçoit un motif qui peut optionnellement avoir des marqueurs et des expressions régulières. Tous les motifs de routage doivent commencer avec une barre oblique (/). La syntaxe utilisée pour les expressions régulières est la même que les [PCRE regular expressions](#). Notez qu'il n'est pas nécessaire d'ajouter les délimiteurs d'expression régulière. Tous les motifs de route sont insensibles à la casse.

Le second paramètre définit comment les parties reconnues sont reliées aux contrôleur/action/paramètre. Les parties à reconnaître sont des marqueurs ou des sous-motifs délimités par des parenthèses (round brackets). Dans l'exemple donné précédemment, le premier sous-motif correspondant (`:controller`) est partie contrôleur de la route, le deuxième est l'action, et ainsi de suite.

Ces marqueurs facilite l'écriture d'expression régulière qui sont plus lisible pour le développeur et facile à comprendre. Les marqueurs suivant sont supportés:

Marqueur	Expression régulière	Utilisation
/:module	/ ([a-zA-Z0-9_\\-])	Correspond à un module valide contenant seulement des caractères alphanumériques
/:controller	/ ([a-zA-Z0-9_\\-])	Correspond à un contrôleur valide contenant seulement des caractères alphanumériques
/:action	/ ([a-zA-Z0-9_\\-]+)	Correspond à une action valide contenant seulement des caractères alphanumériques
/:params	/ (/.*) *	Correspond à une liste de mots optionnels séparés par des slashes. A n'utiliser qu'en fin de route !
/:namespace	/ ([a-zA-Z0-9_\\-]+)	Correspond à un espace de nom à un seul niveau
/:int	/ ([0-9]+)	Correspond à un paramètre de type entier

Les noms de contrôleur sont “camélisés”. Ceci signifie que les caractères (-) et (_) sont retirés et que le caractère qui suit est mis en majuscule. Par exemple, un_controleur est converti en UnControleur.

Depuis que vous pouvez ajouter autant de routes que nécessaire grâce à la méthode `add()`, l'ordre d'ajout des routes indique leur pertinence, les dernières routes ajoutées étant plus pertinentes que les premières. En interne, toutes les routes sont parcourues dans l'ordre inverse jusqu'à ce que *Phalcon\MvcRouter* trouve celle qui correspond à l'URI fournie et la traite, ignorant alors le reste.

Paramètres avec des Noms

L'exemple ci-dessous démontre comment définir des noms pour les paramètres d'une route:

```
<?php

$router->add(
    "/news/([0-9]{4})/([0-9]{2})/([0-9]{2})/:params",
    [
        "controller" => "posts",
        "action"      => "show",
        "year"        => 1, // ([0-9]{4})
        "month"       => 2, // ([0-9]{2})
        "day"         => 3, // ([0-9]{2})
        "params"      => 4, // :params
    ]
);
```

Dans l'exemple précédent, la route ne contient aucune partie “contrôler” ou “action”. Ces parties sont remplacées par des valeurs constantes (“posts” et “show”). L'utilisateur ignore quel est le contrôleur qui est réellement concerné par la requête. Dans le contrôleur, on peut accéder à ces paramètres nommés de la manière suivante:

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {

    }

    public function showAction()
```

```

{
    // Get "year" parameter
    $year = $this->dispatcher->getParam("year");

    // Get "month" parameter
    $month = $this->dispatcher->getParam("month");

    // Get "day" parameter
    $day = $this->dispatcher->getParam("day");

    // ...
}

```

Notez que les valeurs des paramètres sont obtenues depuis le répartiteur. Ceci arrive parce que c'est le composant qui finalement interagit avec les pilotes de votre application. De plus, il existe une autre façon de créer des paramètres nommées à l'intérieur du motif:

```

<?php
$router->add(
    "/documentation/{chapter}/{name}.{type:[a-z]+}",
    [
        "controller" => "documentation",
        "action"      => "show",
    ]
);

```

Vous pouvez accéder aux valeurs de la même façon que précédemment:

```

<?php
use Phalcon\Mvc\Controller;

class DocumentationController extends Controller
{
    public function showAction()
    {
        // Get "name" parameter
        $name = $this->dispatcher->getParam("name");

        // Get "type" parameter
        $type = $this->dispatcher->getParam("type");

        // ...
    }
}

```

Syntaxe courte

Si vous n'aimez pas utiliser les tableaux pour définir des routes, une autre syntaxe est possible. L'exemple suivant produit le même résultat:

```

<?php

// Forme courte

```

```
$router->add(
    "/posts/{year:[0-9]+}/{title:[a-z\-\-]+}",
    "Posts::show"
);

// Forme tableau
$router->add(
    "/posts/([0-9]+)/([a-z\-\-]+)",
    [
        "controller" => "posts",
        "action"      => "show",
        "year"        => 1,
        "title"       => 2,
    ]
);
```

Mélanger les Syntaxes Tableau et Courtes

Les syntaxes tableau et courtes peuvent être mélangées pour définir une route. Dans ce cas, notez que les paramètres nommés sont ajoutés automatiquement aux chemins selon la position dans laquelle ils sont définis:

```
<?php

// La première position est ignorée parce qu'elle est utilisée
// pour le paramètre 'country'
$router->add(
    "/news/{country:[a-z]{2}}/([a-z\-\-]+)/([a-z\-\-]+)",
    [
        "section" => 2, // Positions start with 2
        "article" => 3,
    ]
);
```

Router vers des Modules

Vous pouvez définir des routes dont les chemins incluent des modules. Ceci est spécialement adapté aux application multi-modules. Il est possible de définir une route qui inclus un joker pour le module:

```
<?php

use Phalcon\Mvc\Router;

$router = new Router(false);

$router->add(
    "[:module]/:controller/:action/:params",
    [
        "module"      => 1,
        "controller"  => 2,
        "action"      => 3,
        "params"      => 4,
    ]
);
```


Dans le cas le nom de module sera toujours partie intégrante de l'URL. Par exemple, l'URL: /admin/users/edit/sonny sera traitée comme:

Module	admin
Contrôleur	users
Action	edit
Paramètre	sonny

Ou bien vous pouvez rattacher des routes spécifiques à des modules spécifiques:

```
<?php

$router->add(
    "/login",
    [
        "module"      => "backend",
        "controller" => "login",
        "action"      => "index",
    ]
);

$router->add(
    "/products/:action",
    [
        "module"      => "frontend",
        "controller" => "products",
        "action"      => 1,
    ]
);
```

Ou les rattacher à des espaces de noms spécifiques:

```
<?php

$router->add(
    "namespace/login",
    [
        "namespace" => 1,
        "controller" => "login",
        "action"     => "index",
    ]
);
```

Les noms d'espace de nom et de classe doivent être transmis séparément:

```
<?php

$router->add(
    "/login",
    [
        "namespace" => "Backend\\Controllers",
        "controller" => "login",
        "action"     => "index",
    ]
);
```

Restriction de la Méthode HTTP

Lorsque vous ajoutez une route en utilisant simplement `add()` la route est définie pour toutes les méthodes HTTP. De temps en temps, nous pouvons restreindre une route à une méthode en particulier. Ceci est spécialement utile lors de la création d'applications RESTful:

```
<?php

// Cette route correspondra seulement si la méthode HTTP est GET
$router->addGet(
    "/products/edit/{id}",
    "Products::edit"
);

// Cette route correspondra seulement si la méthode HTTP est POST
$router->addPost(
    "/products/save",
    "Products::save"
);

// Cette route correspondra seulement si la méthode HTTP est POST ou PUT
$router->add(
    "/products/update",
    "Products::update"
)->via(
    [
        "POST",
        "PUT",
    ]
);
```

Utilisation de Convertisseurs

Les convertisseurs vous permettent de transformer librement les paramètres d'une route avant de les transmettre au répartiteur. Les exemples qui suivent vous montre comment s'en servir:

```
<?php

// Le nom de l'action autorise les tirets. Une action peut être: /products/new-ipod-
↳ nano-4-generation
$route = $router->add(
    "/products/{slug:[a-z\-\-]+}",
    [
        "controller" => "products",
        "action"      => "show",
    ]
);

$route->convert(
    "slug",
    function ($slug) {
        // Transforme slug en supprimant les tirets
        return str_replace("-", "", $slug);
    }
);
```

Un autre cas d'utilisation des convertisseurs est de relier un modèle à une route. Ceci permet de transmettre directement le modèle à l'action:

```
<?php

// Cet exemple fonctionne en supposant que l'ID est transmis en paramètre dans l'url:
↳ /products/4
$route = $router->add(
    "/products/{id}",
    [
        "controller" => "products",
        "action"      => "show",
    ]
);

$route->convert(
    "id",
    function ($id) {
        // Fetch the model
        return Product::findFirstById($id);
    }
);
```

Groupe de Routes

Si un ensemble de route a des chemins communs, ils peuvent être regroupés pour les maintenir aisément:

```
<?php

use Phalcon\Mvc\Router;
use Phalcon\Mvc\Router\Group as RouterGroup;

$router = new Router();

// Création d'un groupe avec un module et un contrôleur communs
$blog = new RouterGroup(
    [
        "module"      => "blog",
        "controller" => "index",
    ]
);

// Toutes les routes commencent par /blog
$blog->setPrefix("/blog");

// Ajout d'une route au groupe
$blog->add(
    "/save",
    [
        "action" => "save",
    ]
);

// Ajout d'une autre route au groupe
$blog->add(
    "/edit/{id}",
    [
```

```
        "action" => "edit",
    ]
);

// Cette route est reliée à un autre contrôleur que celui par défaut
$blog->add(
    "/blog",
    [
        "controller" => "blog",
        "action"      => "index",
    ]
);

// Ajout du groupe au routeur
$router->mount($blog);
```

Vous pouvez placer les groupes de routes dans des fichiers distincts pour améliorer l'organisation et la réutilisation de code:

```
<?php

use Phalcon\Mvc\Router\Group as RouterGroup;

class BlogRoutes extends RouterGroup
{
    public function initialize()
    {
        // Default paths
        $this->setPaths(
            [
                "module"    => "blog",
                "namespace" => "Blog\\Controllers",
            ]
        );

        // Toutes les routes commencent par /blog
        $this->setPrefix("/blog");

        // Ajout d'une route au groupe
        $this->add(
            "/save",
            [
                "action" => "save",
            ]
        );

        // Ajout d'une autre route au groupe
        $this->add(
            "/edit/{id}",
            [
                "action" => "edit",
            ]
        );

        // Cette route est reliée à un autre contrôleur que celui par défaut
        $this->add(
            "/blog",
            [
```

```

        "controller" => "blog",
        "action"      => "index",
    ]
);
}
}

```

On monte le groupe dans le routeur:

```

<?php

// Ajout du groupe au routeur
$router->mount(
    new BlogRoutes()
);

```

Correspondance de Routes

Une URI valide doit être transmise au routeur pour qu'il puisse la traiter et trouver une route correspondante. Par défaut, l'URI à router est prise dans la variable `$_GET['_url']` qui est créée par le module de réécriture. Un ensemble de règles de réécriture qui fonctionne bien avec Phalcon est:

```

RewriteEngine On
RewriteCond    %{REQUEST_FILENAME} !-d
RewriteCond    %{REQUEST_FILENAME} !-f
RewriteRule    ^((?s).*)$ index.php?_url=/$1 [QSA,L]

```

Avec cette configuration, toutes les requêtes vers des fichiers ou des dossiers qui n'existent pas sont envoyés à `index.php`.

L'exemple suivant montre comment utiliser ce composant dans un mode autonome:

```

<?php

use Phalcon\Mvc\Router;

// Création du routeur
$router = new Router();

// Définition de routes s'il y a
// ...

// Récupère l'URI depuis $_GET["_url"]
$router->handle();

// Ou en définissant l'URI directement
$router->handle("/employees/edit/17");

// Récupération du contrôleur trouvé
echo $router->getControllerName();

// Récupération de l'action trouvée
echo $router->getActionName();

// Récupération de la route trouvée
$route = $router->getMatchedRoute();

```

Routes Nommées

Chaque route ajoutée au routeur est stockée en interne en tant qu'objet de *Phalcon\Mvc\Router\Route*. Cette classe encapsule tous les détails d'une route. Par exemple, nous pouvons donner un nom au chemin afin de l'identifier de manière unique dans notre application. Ceci est particulièrement utile lorsqu'il faut s'en servir pour créer des URLs.

```
<?php

$route = $router->add(
    "/posts/{year}/{title}",
    "Posts::show"
);

$route->setName("show-posts");
```

Ensuite en utilisant par exemple le composant *Phalcon\Mvc\Url* nous pouvons contruire des routes à partir de son nom:

```
<?php

// Retourne /posts/2012/phalcon-1-0-released
echo $url->get(
    [
        "for"    => "show-posts",
        "year"   => "2012",
        "title"  => "phalcon-1-0-released",
    ]
);
```

Exemple d'utilisation

Ce qui suit sont des exemples de routes personnalisées:

```
<?php

// Trouve "/system/admin/a/edit/7001"
$route->add(
    "/system/:controller/a/:action/:params",
    [
        "controller" => 1,
        "action"     => 2,
        "params"     => 3,
    ]
);

// Trouve "/es/news"
$route->add(
    "/([a-z]{2}):(controller)",
    [
        "controller" => 2,
        "action"     => "index",
        "language"   => 1,
    ]
);

// Trouve "/es/news"
$route->add(
```

```

       ("/{language:[a-z]{2}}/:controller",
        [
            "controller" => 2,
            "action"      => "index",
        ]
    );

// Trouve "/admin/posts/edit/100"
$router->add(
    "/admin/:controller/:action/:int",
    [
        "controller" => 1,
        "action"      => 2,
        "id"          => 3,
    ]
);

// Trouve "/posts/2015/02/some-cool-content"
$router->add(
    "/posts/([0-9]{4})/([0-9]{2})/([a-z\-\-]+)",
    [
        "controller" => "posts",
        "action"      => "show",
        "year"        => 1,
        "month"       => 2,
        "title"       => 4,
    ]
);

// Trouve "/manual/en/translate.adapter.html"
$router->add(
    "/manual/([a-z]{2})/([a-z\-.]+)\.html",
    [
        "controller" => "manual",
        "action"      => "show",
        "language"    => 1,
        "file"        => 2,
    ]
);

// Trouve /feed/fr/le-robots-hot-news.atom
$router->add(
    "/feed/{lang:[a-z]+}/{blog:[a-z\-\-]+}\.{type:[a-z\-\-]+}",
    "Feed::get"
);

// Trouve /api/v1/users/peter.json
$router->add(
    "/api/(v1|v2)/{method:[a-z]+}/{param:[a-z]+}\.{(json|xml) }",
    [
        "controller" => "api",
        "version"     => 1,
        "format"      => 4,
    ]
);

```

Prenez garde aux caractères autorisés dans les expressions régulières pour les contrôleurs et les espaces de noms. Comme ils deviennent des noms de classe, ils peuvent permettre à des attaquants d'atteindre

le système de fichiers et donc de lire des fichiers non autorisés. Une expression régulière sûre est /
([a-zA-Z0-9_\-] +)

Comportement par Défaut

Phalcon\Mvc\Router a un comportement par défaut qui fournit un routage très simple qui s'attend à ce que l'URI corresponde au motif: `/:controller/:action/:params`

Par exemple pour une URL du style `http://phalconphp.com/documentation/show/about.html`, le routeur transformera comme suit:

Contrôleur	documentation
Action	show
Paramètre	about.html

Si vous ne souhaitez pas que le routeur ait ce comportement, vous devez créer le routeur en passant `false` en premier paramètre:

```
<?php

use Phalcon\Mvc\Router;

// Création du routeur sans route par défaut
$router = new Router(false);
```

Définir la route par défaut

Quand votre application est accédée sans aucune route c'est la route `'/'` qui est utilisée pour déterminer quels sont les chemins à utiliser pour afficher la page initiale de votre site web ou de votre application:

```
<?php

$router->add(
    "/",
    [
        "controller" => "index",
        "action"      => "index",
    ]
);
```

Chemins Introuvables

Si aucune des routes spécifiées au routeur ne correspond, vous pouvez définir un groupe de chemin pour ce type de scénario:

```
<?php

// Set 404 paths
$router->notFound(
    [
        "controller" => "index",
        "action"      => "route404",
    ]
);
```


Ceci est typiquement pour une page d'Erreur 404.

Etablir des chemins par défaut

Il est possible de définir des valeurs par défaut pour le module, le contrôleur ou l'action. Lorsqu'il manque une route, n'importe lequel des ces chemin peut être automatiquement complété par le routeur:

```
<?php

// Définition d'un défaut spécifique
$router->setDefaultModule("backend");
$router->setDefaultNamespace("Backend\\Controllers");
$router->setDefaultController("index");
$router->setDefaultAction("index");

// Avec un tableau
$router->setDefaults(
    [
        "controller" => "index",
        "action"      => "index",
    ]
);
```

Traitement des slashes terminaux

Il arrive qu'une route soit accédée avec des slashes terminaux. Ces slashes en trop peuvent provoquer un état de non-trouvé dans le répartiteur. Vous pouvez paramétrer le routeur pour qu'il retire automatiquement les slashes qui se trouvent à la fin d'une route:

```
<?php

use Phalcon\Mvc\Router;

$router = new Router();

// Retrait automatique des slashes terminaux
$router->removeExtraSlashes(true);
```

Ou bien, vous pouvez modifier des routes en particulier pour qu'elles acceptent des slashes terminaux:

```
<?php

// The [/]{0,1} autorise cette route de terminer éventuellement avec un slash
$router->add(
    "{language:[a-z]{2}}/:controller[/]{0,1}",
    [
        "controller" => 2,
        "action"      => "index",
    ]
);
```

Rappel sur Correspondance

De temps en temps, des routes ne peuvent correspondre que si elle remplissent certaines conditions. Vous pouvez ajouter des conditions arbitraires aux routes en utilisant la fonction de rappel `beforeMatch()`. Si la fonction

retourne false, la route sera considérée comme ne pas correspondre:

```
<?php

$route = $router->add("/login",
    [
        "module"      => "admin",
        "controller" => "session",
    ]
);

$route->beforeMatch(
    function ($uri, $route) {
        // Vérifie qu'il s'agit d'une requête Ajax
        if (isset($_SERVER["HTTP_X_REQUESTED_WITH"]) && $_SERVER["HTTP_X_REQUESTED_
        ↪WITH"] === "XMLHttpRequest") {
            return false;
        }

        return true;
    }
);
```

Vous pouvez réutiliser des conditions complémentaires dans des classes:

```
<?php

class AjaxFilter
{
    public function check()
    {
        return $_SERVER["HTTP_X_REQUESTED_WITH"] === "XMLHttpRequest";
    }
}
```

Et exploiter cette classe au lieu d'une fonction anonyme:

```
<?php

$route = $router->add(
    "/get/info/{id}",
    [
        "controller" => "products",
        "action"      => "info",
    ]
);

$route->beforeMatch(
    [
        new AjaxFilter(),
        "check"
    ]
);
```

Depuis Phalcon 3, il existe une autre façon de vérifier:

```
<?php

$route = $router->add(
```

```

        "/login",
        [
            "module"      => "admin",
            "controller" => "session",
        ]
    );

    $route->beforeMatch(
        function ($uri, $route) {
            /**
             * @var string $uri
             * @var \Phalcon\Mvc\Router\Route $route
             * @var \Phalcon\DiInterface $this
             * @var \Phalcon\Http\Request $request
             */
            $request = $this->getShared("request");

            // Vérifie qu'il s'agit d'une requête Ajax
            return $request->isAjax();
        }
    );

```

Contraintes de Nom d'Hôte

Le routeur vous permet d'établir des contraintes selon le nom de l'hôte, ceci signifie que des routes spécifiques ou des groupes de routes peuvent être restreintes seulement si la route satisfait la contrainte du nom d'hôte;

```

<?php

$route = $router->add(
    "/login",
    [
        "module"      => "admin",
        "controller" => "session",
        "action"      => "login",
    ]
);

$route->setHostName("admin.company.com");

```

Le nom d'hôte peut également être transmis sous forme d'expression régulière:

```

<?php

$route = $router->add(
    "/login",
    [
        "module"      => "admin",
        "controller" => "session",
        "action"      => "login",
    ]
);

$route->setHostName("([a-z]+).company.com");

```

Vous pouvez faire en sorte qu'une contrainte de nom d'hôte s'applique à toutes les routes d'un groupe de routes:

```
<?php

use Phalcon\Mvc\Router\Group as RouterGroup;

// Création d'un groupe avec un module et un contrôleur communs
$blog = new RouterGroup(
    [
        "module"      => "blog",
        "controller" => "posts",
    ]
);

// Restriction sur le nom de l'hôte
$blog->setHostName("blog.mycompany.com");

// Toutes les routes commencent par /blog
$blog->setPrefix("/blog");

// Route par défaut
$blog->add(
    "/",
    [
        "action" => "index",
    ]
);

// Ajout d'une route au groupe
$blog->add(
    "/save",
    [
        "action" => "save",
    ]
);

// Ajout d'une autre route au groupe
$blog->add(
    "/edit/{id}",
    [
        "action" => "edit",
    ]
);

// Ajout du groupe au routeur
$router->mount($blog);
```

Sources d'URI

Par défaut l'URI est extraite de la variable `$_GET['_url']` qui est transmise à Phalcon par le moteur de réécriture. Vous pouvez également utiliser `$_SERVER['REQUEST_URI']` si c'est nécessaire:

```
<?php

use Phalcon\Mvc\Router;

// ...
```

```
// Use $_GET["_url"] (default)
$router->setUriSource(
    Router::URI_SOURCE_GET_URL
);

// Use $_SERVER["REQUEST_URI"]
$router->setUriSource(
    Router::URI_SOURCE_SERVER_REQUEST_URI
);
```

Ou bien vous pouvez transmettre manuellement l'URI à la méthode `handle()` :

```
<?php

$router->handle("/some/route/to/handle");
```

Test de vos routes

Tant que le composant n'a pas de dépendances, vous pouvez créer un fichier comme montré ci-dessous pour tester vos routes:

```
<?php

use Phalcon\Mvc\Router;

// Ces routes simulent de vrai URIs
$testRoutes = [
    "/",
    "/index",
    "/index/index",
    "/index/test",
    "/products",
    "/products/index/",
    "/products/show/101",
];

$router = new Router();

// Ajoutez ici vos propres routes
// ...

// Test de chaque route
foreach ($testRoutes as $testRoute) {
    // Gestion de la route
    $router->handle($testRoute);

    echo "Testing ", $testRoute, "<br>";

    // Vérifie que chaque route corresponde
    if ($router->wasMatched()) {
        echo 'Contrôleur: ', $router->getControllerName(), '<br>';
        echo 'Action: ', $router->getActionName(), '<br>';
    } else {
        echo 'La route n\'a pas de correspondance<br>';
    }
}
```

```
    echo "<br>";
}
```

Annotations du Routeur

Ce composant fournit une variante du service *annotations*. Avec cette stratégie vous pouvez écrire les routes directement dans les contrôleurs plutôt que les ajouter dans le service d'inscription:

```
<?php

use Phalcon\Mvc\Router\Annotations as RouterAnnotations;

$di["router"] = function () {
    // Utilise les annotations du routeur. Nous passons 'faux' si nous ne voulons pas
    // que le routeur ajoute son motif par défaut
    $router = new RouterAnnotations(false);

    // Lecture des annotations depuis ProductsController si l'URI commence par /api/
    // products
    $router->addResource("Products", "/api/products");

    return $router;
};
```

Les annotations peuvent être écrites de la façon suivante:

```
<?php

/**
 * @RoutePrefix("/api/products")
 */
class ProductsController
{
    /**
     * @Get (
     *     "/"
     * )
     */
    public function indexAction()
    {

    }

    /**
     * @Get (
     *     "/edit/{id:[0-9]+}",
     *     name="edit-robot"
     * )
     */
    public function editAction($id)
    {

    }

    /**
     * @Route (
```

```

*     "/save",
*     methods={"POST", "PUT"},
*     name="save-robot"
* )
*/
public function saveAction()
{

}

/**
 * @Route(
 *     "/delete/{id:[0-9]+}",
 *     methods="DELETE",
 *     converters={
 *         id="MyConvertors::checkId"
 *     }
 * )
 */
public function deleteAction($id)
{

}

public function infoAction($id)
{

}
}

```

Seules les méthodes marquées par une annotation valide sont utilisées comme routes. Voyez la liste des annotations supportées:

Nom	Description	Exemple de déclaration
RoutePrefix	Un préfixe qui sera placé devant chaque route URI. Cette annotation est à placer dans le docblock de la classe	@RoutePrefix("/api/products")
Route	Cette annotation associe une méthode à une route. Cette annotation est à placer dans le docblock d'une méthode	@Route("/api/products/show")
Get	Cette annotation associe une méthode à une route avec une restriction sur la méthode HTTP GET	@Get("/api/products/search")
Post	Cette annotation associe une méthode à une route avec une restriction sur la méthode HTTP POST	@Post("/api/products/save")
Put	Cette annotation associe une méthode à une route avec une restriction sur la méthode HTTP PUT	@Put("/api/products/save")
Delete	Cette annotation associe une méthode à une route avec une restriction sur la méthode HTTP DELETE	@Delete("/api/products/delete/{id}")
Options	Cette annotation associe une méthode à une route avec une restriction sur la méthode HTTP OPTIONS	@Option("/api/products/info")

Pour les annotations qui ajoutent des routes, les paramètres suivants sont supportés:

Nom	Description	Exemple de déclaration
meth-ods	Définit une ou plusieurs méthodes HTTP que la route doit respecter	<code>@Route ("/api/products", methods={"GET", "POST"})</code>
name	Définit le nom d'une route	<code>@Route ("/api/products", name="get-products")</code>
paths	Un tableau de chemins identiques à ceux passés à <code>Phalcon\Mvc\Router::add()</code>	<code>@Route ("/posts/{id}/{slug}", paths={module="backend"})</code>
con-vert-sors	Un ensemble de convertisseurs qui s'appliquent aux paramètres	<code>@Route ("/posts/{id}/{slug}", converters={id="MyConversor::getId"})</code>

Si vous utilisez des modules dans votre application, il vaut mieux utiliser la méthode `addModuleResource()` :

```
<?php

use Phalcon\Mvc\Router\Annotations as RouterAnnotations;

$di["router"] = function () {
    // Utilise les annotations de routage
    $router = new RouterAnnotations(false);

    // Lecture des annotations depuis Backend\Controllers\ProductsController si l'URI
    // commence par /api/products
    $router->addModuleResource("backend", "Products", "/api/products");

    return $router;
};
```

Inscription d'une Instance de Routeur

Vous pouvez inscrire le routeur lors de la procédure d'inscription du service dans l'injecteur de dépendance de Phalcon pour le rendre disponible aux contrôleurs.

Vous devez ajouter le code suivant dans votre fichier d'amorce (par exemple `index.php` ou `app/config/services.php` si vous utilisez [Phalcon Developer Tools](#))

```
<?php

/**
 * Ajout de la capacité de routage
 */
$di->set(
    "router",
    function () {
        require __DIR__ . "../app/config/routes.php";

        return $router;
    }
);
```

Vous devrez créer `app/config/routes.php` et d'ajouter du code d'initialisation du routeur, comme par exemple:

```
<?php

use Phalcon\Mvc\Router;

$router = new Router();
```



```

$router->add(
    "/login",
    [
        "controller" => "login",
        "action"      => "index",
    ]
);

$router->add(
    "/products/:action",
    [
        "controller" => "products",
        "action"      => 1,
    ]
);

return $router;

```

Ecriture de votre propre Routeur

L'interface *Phalcon\Mvc\RouterInterface* doit être implémentée pour créer un routeur en remplacement de celui fourni par Phalcon.

2.3.21 Dispatching Controllers

Phalcon\Mvc\Dispatcher is the component responsible for instantiating controllers and executing the required actions on them in an MVC application. Understanding its operation and capabilities helps us get more out of the services provided by the framework.

The Dispatch Loop

This is an important process that has much to do with the MVC flow itself, especially with the controller part. The work occurs within the controller dispatcher. The controller files are read, loaded, and instantiated. Then the required actions are executed. If an action forwards the flow to another controller/action, the controller dispatcher starts again. To better illustrate this, the following example shows approximately the process performed within *Phalcon\Mvc\Dispatcher*:

```

<?php

// Dispatch loop
while (!$finished) {
    $finished = true;

    $controllerClass = $controllerName . "Controller";

    // Instantiating the controller class via autoloaders
    $controller = new $controllerClass();

    // Execute the action
    call_user_func_array(
        [
            $controller,
            $actionName . "Action"

```

```
        ],
        $params
    );

    // '$finished' should be reloaded to check if the flow was forwarded to another_
    ↪controller
    $finished = true;
}
```

The code above lacks validations, filters and additional checks, but it demonstrates the normal flow of operation in the dispatcher.

Dispatch Loop Events

Phalcon\Mvc\Dispatcher is able to send events to an *EventsManager* if it is present. Events are triggered using the type “dispatch”. Some events when returning boolean false could stop the active operation. The following events are supported:

The *INVO* tutorial shows how to take advantage of dispatching events implementing a security filter with *Acl*

The following example demonstrates how to attach listeners to this component:

```
<?php

use Phalcon\Mvc\Dispatcher as MvcDispatcher;
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;

$di->set(
    "dispatcher",
    function () {
        // Create an event manager
        $eventsManager = new EventsManager();

        // Attach a listener for type "dispatch"
        $eventsManager->attach(
            "dispatch",
            function (Event $event, $dispatcher) {
                // ...
            }
        );

        $dispatcher = new MvcDispatcher();

        // Bind the eventsManager to the view component
        $dispatcher->setEventsManager($eventsManager);

        return $dispatcher;
    },
    true
);
```

An instantiated controller automatically acts as a listener for dispatch events, so you can implement methods as call-backs:

```
<?php
```

```
use Phalcon\Mvc\Controller;
use Phalcon\Mvc\Dispatcher;

class PostsController extends Controller
{
    public function beforeExecuteRoute(Dispatcher $dispatcher)
    {
        // Executed before every found action
    }

    public function afterExecuteRoute(Dispatcher $dispatcher)
    {
        // Executed after every found action
    }
}
```

Note: Methods on event listeners accept an *Phalcon\Events\Event* object as their first parameter - methods in controllers do not.

Forwarding to other actions

The dispatch loop allows us to forward the execution flow to another controller/action. This is very useful to check if the user can access to certain options, redirect users to other screens or simply reuse code.

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {
    }

    public function saveAction($year, $postTitle)
    {
        // ... Store some product and forward the user

        // Forward flow to the index action
        $this->dispatcher->forward([
            "controller" => "posts",
            "action"      => "index",
        ]);
    }
}
```

Keep in mind that making a “forward” is not the same as making a HTTP redirect. Although they apparently got the same result. The “forward” doesn’t reload the current page, all the redirection occurs in a single request, while the HTTP redirect needs two requests to complete the process.

More forwarding examples:

```
<?php

// Forward flow to another action in the current controller
$this->dispatcher->forward(
    [
        "action" => "search"
    ]
);

// Forward flow to another action in the current controller
// passing parameters
$this->dispatcher->forward(
    [
        "action" => "search",
        "params" => [1, 2, 3]
    ]
);
```

A forward action accepts the following parameters:

Parameter	Triggered
controller	A valid controller name to forward to.
action	A valid action name to forward to.
params	An array of parameters for the action
namespace	A valid namespace name where the controller is part of

Preparing Parameters

Thanks to the hooks points provided by *Phalcon\Mvc\Dispatcher* you can easily adapt your application to any URL schema:

For example, you want your URLs look like: <http://example.com/controller/key1/value1/key2/value>

Parameters by default are passed as they come in the URL to actions, you can transform them to the desired schema:

```
<?php

use Phalcon\Dispatcher;
use Phalcon\Mvc\Dispatcher as MvcDispatcher;
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;

$di->set(
    "dispatcher",
    function () {
        // Create an EventsManager
        $eventsManager = new EventsManager();

        // Attach a listener
        $eventsManager->attach(
            "dispatch:beforeDispatchLoop",
            function (Event $event, $dispatcher) {
                $params = $dispatcher->getParams();

                $keyParams = [];

                // Use odd parameters as keys and even as values
```

```

        foreach ($params as $i => $value) {
            if ($i & 1) {
                // Previous param
                $key = $params[$i - 1];

                $keyParams[$key] = $value;
            }
        }

        // Override parameters
        $dispatcher->setParams($keyParams);
    }
);

$dispatcher = new MvcDispatcher();

$dispatcher->setEventManager($eventsManager);

return $dispatcher;
}
);

```

If the desired schema is: <http://example.com/controller/key1:value1/key2:value>, the following code is required:

```

<?php

use Phalcon\Dispatcher;
use Phalcon\Mvc\Dispatcher as MvcDispatcher;
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;

$di->set(
    "dispatcher",
    function () {
        // Create an EventsManager
        $eventsManager = new EventsManager();

        // Attach a listener
        $eventsManager->attach(
            "dispatch:beforeDispatchLoop",
            function (Event $event, $dispatcher) {
                $params = $dispatcher->getParams();

                $keyParams = [];

                // Explode each parameter as key,value pairs
                foreach ($params as $number => $value) {
                    $parts = explode(":", $value);

                    $keyParams[$parts[0]] = $parts[1];
                }

                // Override parameters
                $dispatcher->setParams($keyParams);
            }
        );

        $dispatcher = new MvcDispatcher();
    }
);

```

```
        $dispatcher->setEventManager($eventsManager);

        return $dispatcher;
    }
};
```

Getting Parameters

When a route provides named parameters you can receive them in a controller, a view or any other component that extends *Phalcon\Di\Injectable*.

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {

    }

    public function saveAction()
    {
        // Get the post's title passed in the URL as parameter
        // or prepared in an event
        $title = $this->dispatcher->getParam("title");

        // Get the post's year passed in the URL as parameter
        // or prepared in an event also filtering it
        $year = $this->dispatcher->getParam("year", "int");

        // ...
    }
}
```

Preparing actions

You can also define an arbitrary schema for actions before be dispatched.

Camelize action names

If the original URL is: <http://example.com/admin/products/show-latest-products>, and for example you want to camelize ‘show-latest-products’ to ‘ShowLatestProducts’, the following code is required:

```
<?php

use Phalcon\Text;
use Phalcon\Mvc\Dispatcher as MvcDispatcher;
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;
```

```

$di->set(
    "dispatcher",
    function () {
        // Create an EventsManager
        $eventsManager = new EventsManager();

        // Camelize actions
        $eventsManager->attach(
            "dispatch:beforeDispatchLoop",
            function (Event $event, $dispatcher) {
                $dispatcher->setActionName(
                    Text::camelize($dispatcher->getActionName())
                );
            }
        );

        $dispatcher = new MvcDispatcher();

        $dispatcher->setEventsManager($eventsManager);

        return $dispatcher;
    }
);

```

Remove legacy extensions

If the original URL always contains a ‘.php’ extension:

<http://example.com/admin/products/show-latest-products.php> <http://example.com/admin/products/index.php>

You can remove it before dispatch the controller/action combination:

```

<?php

use Phalcon\Mvc\Dispatcher as MvcDispatcher;
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;

$di->set(
    "dispatcher",
    function () {
        // Create an EventsManager
        $eventsManager = new EventsManager();

        // Remove extension before dispatch
        $eventsManager->attach(
            "dispatch:beforeDispatchLoop",
            function (Event $event, $dispatcher) {
                $action = $dispatcher->getActionName();

                // Remove extension
                $action = preg_replace("/\.php$/", "", $action);

                // Override action
                $dispatcher->setActionName($action);
            }
        );
    }
);

```

```
        $dispatcher = new MvcDispatcher();

        $dispatcher->setEventManager($eventsManager);

        return $dispatcher;
    }
};
```

Inject model instances

In this example, the developer wants to inspect the parameters that an action will receive in order to dynamically inject model instances.

The controller looks like:

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    /**
     * Shows posts
     *
     * @param \Posts $post
     */
    public function showAction(Posts $post)
    {
        $this->view->post = $post;
    }
}
```

Method 'showAction' receives an instance of the model Posts, the developer could inspect this before dispatch the action preparing the parameter accordingly:

```
<?php

use Exception;
use Phalcon\Mvc\Model;
use Phalcon\Mvc\Dispatcher as MvcDispatcher;
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;
use ReflectionMethod;

$di->set(
    "dispatcher",
    function () {
        // Create an EventsManager
        $eventsManager = new EventsManager();

        $eventsManager->attach(
            "dispatch:beforeDispatchLoop",
            function (Event $event, $dispatcher) {
                // Possible controller class name
                $controllerName = $dispatcher->getControllerClass();
```



```

        // Possible method name
        $actionName = $dispatcher->getActiveMethod();

        try {
            // Get the reflection for the method to be executed
            $reflection = new ReflectionMethod($controllerName, $actionName);

            $parameters = $reflection->getParameters();

            // Check parameters
            foreach ($parameters as $parameter) {
                // Get the expected model name
                $className = $parameter->getClass()->name;

                // Check if the parameter expects a model instance
                if (is_subclass_of($className, Model::class)) {
                    $model = $className::findFirstById($dispatcher->
→getParams()[0]);

                    // Override the parameters by the model instance
                    $dispatcher->setParams([$model]);
                }
            }
        } catch (Exception $e) {
            // An exception has occurred, maybe the class or action does not
→exist?
        }
    }
    );

    $dispatcher = new MvcDispatcher();

    $dispatcher->setEventManager($eventsManager);

    return $dispatcher;
}
);

```

The above example has been simplified for academic purposes. A developer can improve it to inject any kind of dependency or model in actions before be executed.

From 3.1.x onwards the dispatcher also comes with an option to handle this internally for all models passed into a controller action by using `Phalcon\Mvc\Model\Binder`.

```

use Phalcon\Mvc\Dispatcher;
use Phalcon\Mvc\Model\Binder;

$dispatcher = new Dispatcher();

$dispatcher->setModelBinder(new Binder());

return $dispatcher;

```

Since Binder object is using internally Reflection Api which can be heavy there is ability to set cache. This can be done by using second argument in `setModelBinder()` which can also accept service name or just by passing cache instance to Binder constructor.

It also introduces a new interface `Phalcon\Mvc\Model\Binder\BindableInterface` which allows you to define the controllers associated models to allow models binding in base controllers.

For example, you have a base `CrudController` which your `PostsController` extends from. Your `CrudController` looks something like this:

```
use Phalcon\Mvc\Controller;
use Phalcon\Mvc\Model;

class CrudController extends Controller
{
    /**
     * Show action
     *
     * @param Model $model
     */
    public function showAction(Model $model)
    {
        $this->view->model = $model;
    }
}
```

In your `PostsController` you need to define which model the controller is associated with. This is done by implementing the `Phalcon\Mvc\Model\Binder\BindableInterface` which will add the `getModelName()` method from which you can return the model name. It can return string with just one model name or associative array where key is parameter name.

```
use Phalcon\Mvc\Model\Binder\BindableInterface;
use Models\Posts;

class PostsController extends CrudController implements BindableInterface
{
    public static function getModelName()
    {
        return Posts::class;
    }
}
```

By declaring the model associated with the `PostsController` the binder can check the controller for the `getModelName()` method before passing the defined model into the parent show action.

If your project structure does not use any parent controller you can of course still bind the model directly into the controller action:

```
use Phalcon\Mvc\Controller;
use Models\Posts;

class PostsController extends Controller
{
    /**
     * Shows posts
     *
     * @param Posts $post
     */
    public function showAction(Posts $post)
    {
        $this->view->post = $post;
    }
}
```

Currently the binder will only use the models primary key to perform a `findFirst()` on. An example route for the above would be `/posts/show/{1}`

Handling Not-Found Exceptions

Using the *EventsManager* it's possible to insert a hook point before the dispatcher throws an exception when the controller/action combination wasn't found:

```
<?php

use Exception;
use Phalcon\Dispatcher;
use Phalcon\Mvc\Dispatcher as MvcDispatcher;
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;
use Phalcon\Mvc\Dispatcher\Exception as DispatchException;

$di->setShared(
    "dispatcher",
    function () {
        // Create an EventsManager
        $eventsManager = new EventsManager();

        // Attach a listener
        $eventsManager->attach(
            "dispatch:beforeException",
            function (Event $event, $dispatcher, Exception $exception) {
                // Handle 404 exceptions
                if ($exception instanceof DispatchException) {
                    $dispatcher->forward(
                        [
                            "controller" => "index",
                            "action"      => "show404",
                        ]
                    );

                    return false;
                }

                // Alternative way, controller or action doesn't exist
                switch ($exception->getCode()) {
                    case Dispatcher::EXCEPTION_HANDLER_NOT_FOUND:
                    case Dispatcher::EXCEPTION_ACTION_NOT_FOUND:
                        $dispatcher->forward(
                            [
                                "controller" => "index",
                                "action"      => "show404",
                            ]
                        );

                        return false;
                    }
                }
            }
        );

        $dispatcher = new MvcDispatcher();
```

```
// Bind the EventsManager to the dispatcher
$dispatcher->setEventsManager($eventsManager);

return $dispatcher;
}
);
```

Of course, this method can be moved onto independent plugin classes, allowing more than one class take actions when an exception is produced in the dispatch loop:

```
<?php

use Exception;
use Phalcon\Events\Event;
use Phalcon\Mvc\Dispatcher;
use Phalcon\Mvc\Dispatcher\Exception as DispatchException;

class ExceptionsPlugin
{
    public function beforeException(Event $event, Dispatcher $dispatcher, Exception
    ↪$exception)
    {
        // Default error action
        $action = "show503";

        // Handle 404 exceptions
        if ($exception instanceof DispatchException) {
            $action = "show404";
        }

        $dispatcher->forward([
            "controller" => "index",
            "action"      => $action,
        ]);

        return false;
    }
}
```

Only exceptions produced by the dispatcher and exceptions produced in the executed action are notified in the ‘beforeException’ events. Exceptions produced in listeners or controller events are redirected to the latest try/catch.

Implementing your own Dispatcher

The *Phalcon\Mvc\DispatcherInterface* interface must be implemented to create your own dispatcher replacing the one provided by Phalcon.

2.3.22 Micro Applications

With Phalcon you can create “Micro-Framework like” applications. By doing this, you only need to write a minimal amount of code to create a PHP application. Micro applications are suitable to implement small applications, APIs

and prototypes in a practical way.

```
<?php

use Phalcon\Mvc\Micro;

$app = new Micro();

$app->get(
    "/say/welcome/{name}",
    function ($name) {
        echo "<h1>Welcome $name!</h1>";
    }
);

$app->handle();
```

Creating a Micro Application

Phalcon\Mvc\Micro is the class responsible for implementing a micro application.

```
<?php

use Phalcon\Mvc\Micro;

$app = new Micro();
```

Defining routes

After instantiating the object, you will need to add some routes. *Phalcon\Mvc\Router* manages routing internally. Routes must always start with /. A HTTP method constraint is optionally required when defining routes, so as to instruct the router to match only if the request also matches the HTTP methods. The following example shows how to define a route for the method GET:

```
<?php

$app->get(
    "/say/hello/{name}",
    function ($name) {
        echo "<h1>Hello! $name</h1>";
    }
);
```

The “get” method indicates that the associated HTTP method is GET. The route `/say/hello/{name}` also has a parameter `{ $name }` that is passed directly to the route handler (the anonymous function). Handlers are executed when a route is matched. A handler could be any callable item in the PHP userland. The following example shows how to define different types of handlers:

```
<?php

// With a function
function say_hello($name) {
    echo "<h1>Hello! $name</h1>";
}
```

```
$app->get(
    "/say/hello/{name}",
    "say_hello"
);

// With a static method
$app->get(
    "/say/hello/{name}",
    "SomeClass::someSayMethod"
);

// With a method in an object
$myController = new MyController();
$app->get(
    "/say/hello/{name}",
    [
        $myController,
        "someAction"
    ]
);

// Anonymous function
$app->get(
    "/say/hello/{name}",
    function ($name) {
        echo "<h1>Hello! $name</h1>";
    }
);
```

Phalcon\Mvc\Micro provides a set of methods to define the HTTP method (or methods) which the route is constrained for:

```
<?php

// Matches if the HTTP method is GET
$app->get(
    "/api/products",
    "get_products"
);

// Matches if the HTTP method is POST
$app->post(
    "/api/products/add",
    "add_product"
);

// Matches if the HTTP method is PUT
$app->put(
    "/api/products/update/{id}",
    "update_product"
);

// Matches if the HTTP method is DELETE
$app->delete(
    "/api/products/remove/{id}",
    "delete_product"
);
```

```
// Matches if the HTTP method is OPTIONS
$app->options(
    "/api/products/info/{id}",
    "info_product"
);

// Matches if the HTTP method is PATCH
$app->patch(
    "/api/products/update/{id}",
    "info_product"
);

// Matches if the HTTP method is GET or POST
$app->map(
    "/repos/store/refs",
    "action_product"
)->via(
    [
        "GET",
        "POST",
    ]
);
```

To access the HTTP method data `$app` needs to be passed into the closure:

```
<?php

// Matches if the HTTP method is POST
$app->post(
    "/api/products/add",
    function () use ($app) {
        echo $app->request->getPost("productID");
    }
);
```

Routes with Parameters

Defining parameters in routes is very easy as demonstrated above. The name of the parameter has to be enclosed in brackets. Parameter formatting is also available using regular expressions to ensure consistency of data. This is demonstrated in the example below:

```
<?php

// This route have two parameters and each of them have a format
$app->get(
    "/posts/{year:[0-9]+}/{title:[a-zA-Z\-\-]+}",
    function ($year, $title) {
        echo "<h1>Title: $title</h1>";
        echo "<h2>Year: $year</h2>";
    }
);
```

Starting Route

Normally, the starting route in an application is the route `/`, and it will more frequent to be accessed by the method GET. This scenario is coded as follows:

```
<?php

// This is the start route
$app->get(
    "/",
    function () {
        echo "<h1>Welcome!</h1>";
    }
);
```

Rewrite Rules

The following rules can be used together with Apache to rewrite the URIs:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^((?s).*)$ index.php?url=/$1 [QSA,L]
</IfModule>
```

Working with Responses

You are free to produce any kind of response in a handler: directly make an output, use a template engine, include a view, return a json, etc.:

```
<?php

// Direct output
$app->get(
    "/say/hello",
    function () {
        echo "<h1>Hello! $name</h1>";
    }
);

// Requiring another file
$app->get(
    "/show/results",
    function () {
        require "views/results.php";
    }
);

// Returning JSON
$app->get(
    "/get/some-json",
    function () {
        echo json_encode(
            [
                "some",
```



```

        "important",
        "data",
    ]
);
}
);

```

In addition to that, you have access to the service “*response*”, with which you can manipulate better the response:

```

<?php

$app->get(
    "/show/data",
    function () use ($app) {
        // Set the Content-Type header
        $app->response->setContentType("text/plain");

        $app->response->sendHeaders();

        // Print a file
        readfile("data.txt");
    }
);

```

Or create a response object and return it from the handler:

```

<?php

$app->get(
    "/show/data",
    function () {
        // Create a response
        $response = new Phalcon\Http\Response();

        // Set the Content-Type header
        $response->setContentType("text/plain");

        // Pass the content of a file
        $response->setContent(file_get_contents("data.txt"));

        // Return the response
        return $response;
    }
);

```

Making redirections

Redirections could be performed to forward the execution flow to another route:

```

<?php

// This route makes a redirection to another route
$app->post("/old/welcome",
    function () use ($app) {
        $app->response->redirect("new/welcome");
    }
);

```

```
        $app->response->sendHeaders();
    }
);

$app->post("/new/welcome",
    function () use ($app) {
        echo "This is the new Welcome";
    }
);
```

Generating URLs for Routes

Phalcon\Mvc\Url can be used to produce URLs based on the defined routes. You need to set up a name for the route; by this way the “url” service can produce the corresponding URL:

```
<?php

// Set a route with the name "show-post"
$app->get(
    "/blog/{year}/{title}",
    function ($year, $title) use ($app) {
        // ... Show the post here
    }
)->setName("show-post");

// Produce a URL somewhere
$app->get(
    "/",
    function () use ($app) {
        echo '<a href="', $app->url->get(
            [
                "for"    => "show-post",
                "title"   => "php-is-a-great-framework",
                "year"    => 2015
            ],
            ">Show the post</a>';
    }
);
```

Interacting with the Dependency Injector

In the micro application, a *Phalcon\Di\FactoryDefault* services container is created implicitly; additionally you can create outside the application a container to manipulate its services:

```
<?php

use Phalcon\Mvc\Micro;
use Phalcon\Di\FactoryDefault;
use Phalcon\Config\Adapter\Ini as IniConfig;

$di = new FactoryDefault();

$di->set(
    "config",
    function () {
```

```

        return new IniConfig("config.ini");
    }
);

$app = new Micro();

$app->setDI($di);

$app->get(
    "/",
    function () use ($app) {
        // Read a setting from the config
        echo $app->config->app_name;
    }
);

$app->post(
    "/contact",
    function () use ($app) {
        $app->flash->success("Yes!, the contact was made!");
    }
);

```

The array-syntax is allowed to easily set/get services in the internal services container:

```

<?php

use Phalcon\Mvc\Micro;
use Phalcon\Db\Adapter\Pdo\Mysql as MysqlAdapter;

$app = new Micro();

// Setup the database service
$app["db"] = function () {
    return new MysqlAdapter(
        [
            "host"      => "localhost",
            "username" => "root",
            "password" => "secret",
            "dbname"    => "test_db"
        ]
    );
};

$app->get(
    "/blog",
    function () use ($app) {
        $news = $app["db"]->query("SELECT * FROM news");

        foreach ($news as $new) {
            echo $new->title;
        }
    }
);

```

Not-Found Handler

When a user tries to access a route that is not defined, the micro application will try to execute the “Not-Found” handler. An example of that behavior is below:

```
<?php

$app->notFound(
    function () use ($app) {
        $app->response->setStatusCode(404, "Not Found");

        $app->response->sendHeaders();

        echo "This is crazy, but this page was not found!";
    }
);
```

Models in Micro Applications

Models can be used transparently in Micro Applications, only is required an autoloader to load models:

```
<?php

$loader = new \Phalcon\Loader();

$loader->registerDirs(
    [
        __DIR__ . "/models/"
    ]
)->register();

$app = new \Phalcon\Mvc\Micro();

$app->get(
    "/products/find",
    function () {
        $products = Products::find();

        foreach ($products as $product) {
            echo $product->name, "<br>";
        }
    }
);

$app->handle();
```

Inject model instances

By using class `Phalcon\Mvc\Model\Binder` you can inject model instances into your routes:

```
<?php

$loader = new \Phalcon\Loader();

$loader->registerDirs(
```

```

[
    __DIR__ . "/models/"
]
)->register();

$app = new \Phalcon\Mvc\Micro();
$app->setModelBinder(new \Phalcon\Mvc\Model\Binder());

$app->get(
    "/products/{product:[0-9]+}",
    function (Products $product) {
        // do anything with $product object
    }
);

$app->handle();

```

Since Binder object is using internally Reflection Api which can be heavy there is ability to set cache. This can be done by using second argument in `setModelBinder()` which can also accept service name or just by passing cache instance to Binder constructor.

Currently the binder will only use the models primary key to perform a `findFirst()` on. An example route for the above would be `/products/1`

Micro Application Events

Phalcon\Mvc\Micro is able to send events to the *EventsManager* (if it is present). Events are triggered using the type “micro”. The following events are supported:

Nom d'évt	Triggered	Opération stoppée ?
beforeHandleRoute	The main method is just called, at this point the application doesn't know if there is some matched route	Oui
beforeExecuteRoute	A route has been matched and it contains a valid handler, at this point the handler has not been executed	Oui
afterExecuteRoute	Triggered after running the handler	Non
beforeNotFound	Triggered when any of the defined routes match the requested URI	Oui
afterHandlerRoute	Triggered after completing the whole process in a successful way	Oui
afterBinding	Triggered after models are bound but before executing the handler	Oui

In the following example, we explain how to control the application security using events:

```

<?php

use Phalcon\Mvc\Micro;
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;

// Create a events manager
$eventsManager = new EventsManager();

$eventsManager->attach(
    "micro:beforeExecuteRoute",

```

```
function (Event $event, $app) {
    if ($app->session->get("auth") === false) {
        $app->flashSession->error("The user isn't authenticated");

        $app->response->redirect("/");

        $app->response->sendHeaders();

        // Return (false) stop the operation
        return false;
    }
}
);

$app = new Micro();

// Bind the events manager to the app
$app->setEventsManager($eventsManager);
```

Middleware events

In addition to the events manager, events can be added using the methods ‘before’, ‘after’ and ‘finish’:

```
<?php

$app = new Phalcon\Mvc\Micro();

// Executed before every route is executed
// Return false cancels the route execution
$app->before(
    function () use ($app) {
        if ($app["session"]->get("auth") === false) {
            $app["flashSession"]->error("The user isn't authenticated");

            $app["response"]->redirect("/error");

            // Return false stops the normal execution
            return false;
        }

        return true;
    }
);

$app->map(
    "/api/robots",
    function () {
        return [
            "status" => "OK",
        ];
    }
);

$app->after(
    function () use ($app) {
        // This is executed after the route is executed
        echo json_encode($app->getReturnedValue());
    }
);
```

```

    }
);

$app->finish(
    function () use ($app) {
        // This is executed when the request has been served
    }
);

```

You can call the methods several times to add more events of the same type:

```

<?php

$app->finish(
    function () use ($app) {
        // First 'finish' middleware
    }
);

$app->finish(
    function () use ($app) {
        // Second 'finish' middleware
    }
);

```

Code for middlewares can be reused using separate classes:

```

<?php

use Phalcon\Mvc\Micro\MiddlewareInterface;

/**
 * CacheMiddleware
 *
 * Caches pages to reduce processing
 */
class CacheMiddleware implements MiddlewareInterface
{
    public function call($application)
    {
        $cache = $application["cache"];
        $router = $application["router"];

        $key = preg_replace("/^[a-zA-Z0-9]/", "", $router->getRewriteUri());

        // Check if the request is cached
        if ($cache->exists($key)) {
            echo $cache->get($key);

            return false;
        }

        return true;
    }
}

```

Then add the instance to the application:

```
<?php

$app->before(
    new CacheMiddleware()
);
```

The following middleware events are available:

Nom d'évt	Triggered	Opération stoppée ?
before	Before executing the handler. It can be used to control the access to the application	Oui
after	Executed after the handler is executed. It can be used to prepare the response	Non
finish	Executed after sending the response. It can be used to perform clean-up	Non

finish | Executed after sending the response. It can be used to perform clean-up | Non |

Using Controllers as Handlers

Medium applications using the `Mvc\Micro` approach may require organize handlers in controllers. You can use *Phalcon\Mvc\Micro\Collection* to group handlers that belongs to controllers:

```
<?php

use Phalcon\Mvc\Micro\Collection as MicroCollection;

$posts = new MicroCollection();

// Set the main handler. ie. a controller instance
$posts->setHandler(
    new PostsController()
);

// Set a common prefix for all routes
$posts->setPrefix("/posts");

// Use the method 'index' in PostsController
$posts->get("/", "index");

// Use the method 'show' in PostsController
$posts->get("/show/{slug}", "show");

$app->mount($posts);
```

The controller 'PostsController' might look like this:

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
```



```
{
    public function index()
    {
        // ...
    }

    public function show($slug)
    {
        // ...
    }
}
```

In the above example the controller is directly instantiated, Collection also have the ability to lazy-load controllers, this option provide better performance loading controllers only if the related routes are matched:

```
<?php

$posts->setHandler("PostsController", true);
$posts->setHandler("Blog\Controllers\PostsController", true);
```

Returning Responses

Handlers may return raw responses using *Phalcon\Http\Response* or a component that implements the relevant interface. When responses are returned by handlers they are automatically sent by the application.

```
<?php

use Phalcon\Mvc\Micro;
use Phalcon\Http\Response;

$app = new Micro();

// Return a response
$app->get(
    "/welcome/index",
    function () {
        $response = new Response();

        $response->setStatusCode(401, "Unauthorized");

        $response->setContent("Access is not authorized");

        return $response;
    }
);
```

Rendering Views

Phalcon\Mvc\View\Simple can be used to render views, the following example shows how to do that:

```
<?php

$app = new Phalcon\Mvc\Micro();

$app["view"] = function () {
```

```
$view = new \Phalcon\Mvc\View\Simple();

$view->setViewsDir("app/views/");

return $view;
};

// Return a rendered view
$app->get(
    "/products/show",
    function () use ($app) {
        // Render app/views/products/show.phtml passing some variables
        echo $app["view"]->render(
            "products/show",
            [
                "id" => 100,
                "name" => "Artichoke"
            ]
        );
    }
);
```

Please note that this code block uses *Phalcon\Mvc\View\Simple* which uses relative paths instead of controllers and actions. If you would like to use *Phalcon\Mvc\View\Simple* instead, you will need to change the parameters of the `render()` method:

```
<?php

$app = new Phalcon\Mvc\Micro();

$app["view"] = function () {
    $view = new \Phalcon\Mvc\View();

    $view->setViewsDir("app/views/");

    return $view;
};

// Return a rendered view
$app->get(
    "/products/show",
    function () use ($app) {
        // Render app/views/products/show.phtml passing some variables
        echo $app["view"]->render(
            "products",
            "show",
            [
                "id" => 100,
                "name" => "Artichoke"
            ]
        );
    }
);
```

Error Handling

A proper response can be generated if an exception is raised in a micro handler:

```
<?php

$app = new Phalcon\Mvc\Micro();

$app->get(
    "/",
    function () {
        throw new \Exception("An error");
    }
);

$app->error(
    function ($exception) {
        echo "An error has occurred";
    }
);
```

If the handler returns “false” the exception is stopped.

Related Sources

- *Creating a Simple REST API* is a tutorial that explains how to create a micro application to implement a RESTful web service.
- *Stickers Store* is a very simple micro-application making use of the micro-mvc approach [[Github](#)].

2.3.23 Travailler avec les espaces de nom

Namespaces, c’est le nom anglais des espaces de nom. Ces derniers peuvent être utilisés pour éviter le conflit des noms de classe. Par exemple si vous avez deux contrôleurs dans votre application avec le même nom, un namespace peut être utilisé pour les différencier. Les espaces de nom sont aussi utiles pour créer des paquets ou des modules.

Mise en place

Utiliser les espaces de nom a quelques implications quand on charge le contrôleur approprié. Pour ajuster le comportement du framework aux namespaces, il est nécessaire de faire une (ou toutes) les tâches suivante:

Utiliser un autoloader qui prends en compte les espaces de nom, par exemple *Phalcon\Loader*:

```
<?php

$loader->registerNamespaces(
    [
        "Store\\Admin\\Controllers" => "../bundles/admin/controllers/",
        "Store\\Admin\\Models"      => "../bundles/admin/models/",
    ]
);
```

Vous pouvez le spécifier aux routes comme un paramètre séparé :

```
<?php

$router->add(
    "/admin/users/my-profile",
    [
        "namespace" => "Store\\Admin",
        "controller" => "Users",
        "action"     => "profile",
    ]
);
```

Le passer en tant que partie du chemin:

```
<?php

$router->add(
    "namespace/admin/users/my-profile",
    [
        "namespace" => 1,
        "controller" => "Users",
        "action"     => "profile",
    ]
);
```

Si vous travaillez avec le même namespace pour tous les contrôleurs de votre application, vous pouvez le définir en tant que namespace par défaut dans le dispatcher, ainsi, vous n'avez pas besoin de spécifier le nom complet de la classe dans le chemin du routeur:

```
<?php

use Phalcon\Mvc\Dispatcher;

// Registering a dispatcher
$di->set(
    "dispatcher",
    function () {
        $dispatcher = new Dispatcher();

        $dispatcher->setDefaultNamespace(
            "Store\\Admin\\Controllers"
        );

        return $dispatcher;
    }
);
```

Contrôleur avec namespace

L'exemple suivante montre comment implémenter un contrôleur qui utilise des espaces de nom:

```
<?php

namespace Store\Admin\Controllers;

use Phalcon\Mvc\Controller;
```

```
class UsersController extends Controller
{
    public function indexAction()
    {

    }

    public function profileAction()
    {

    }

}
```

Modèles et Espaces de Nom

Prenez en considération ce qui suit lors de l'utilisation de modèles dans les espaces de nom:

```
<?php

namespace Store\Models;

use Phalcon\Mvc\Model;

class Robots extends Model
{

}
```

Si des modèles contiennent des relations, ils doivent inclure aussi l'espace de nom:

```
<?php

namespace Store\Models;

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function initialize()
    {
        $this->hasMany(
            "id",
            "Store\\Models\\Parts",
            "robots_id",
            [
                "alias" => "parts",
            ]
        );
    }
}
```

En PHQL vous devez inclure les espaces de nom dans les instructions:

```
<?php

$phql = "SELECT r.* FROM Store\\Models\\Robots r JOIN Store\\Models\\Parts p";
```

2.3.24 Events Manager

The purpose of this component is to intercept the execution of most of the other components of the framework by creating “hook points”. These hook points allow the developer to obtain status information, manipulate data or change the flow of execution during the process of a component.

Naming Convention

Phalcon events use namespaces to avoid naming collisions. Each component in Phalcon occupies a different event namespace and you are free to create your own as you see fit. Event names are formatted as “component:event”. For example, as *Phalcon\Db* occupies the “db” namespace, its “afterQuery” event’s full name is “db:afterQuery”.

When attaching event listeners to the events manager, you can use “component” to catch all events from that component (eg. “db” to catch all of the *Phalcon\Db* events) or “component:event” to target a specific event (eg. “db:afterQuery”).

Usage Example

In the following example, we will use the EventsManager to listen for the “afterQuery” event produced in a MySQL connection managed by *Phalcon\Db*:

```
<?php

use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;
use Phalcon\Db\Adapter\Pdo\Mysql as DbAdapter;

$eventsManager = new EventsManager();

$eventsManager->attach(
    "db:afterQuery",
    function (Event $event, $connection) {
        echo $connection->getSQLStatement();
    }
);

$connection = new DbAdapter(
    [
        "host"      => "localhost",
        "username"  => "root",
        "password"  => "secret",
        "dbname"    => "invo",
    ]
);

// Assign the eventsManager to the db adapter instance
$connection->setEventsManager($eventsManager);

// Send a SQL command to the database server
$connection->query(
    "SELECT * FROM products p WHERE p.status = 1"
);
```

Now every time a query is executed, the SQL statement will be echoed out. The first parameter passed to the lambda function contains contextual information about the event that is running, the second is the source of the event (in this case: the connection itself). A third parameter may also be specified which will contain arbitrary data specific to the event.

You must explicitly set the Events Manager to a component using the `setEventManager()` method in order for that component to trigger events. You can create a new Events Manager instance for each component or you can set the same Events Manager to multiple components as the naming convention will avoid conflicts.

Instead of using lambda functions, you can use event listener classes instead. Event listeners also allow you to listen to multiple events. In this example, we will implement the *Phalcon\Db\Profiler* to detect the SQL statements that are taking longer to execute than expected:

```
<?php

use Phalcon\Db\Profiler;
use Phalcon\Events\Event;
use Phalcon\Logger;
use Phalcon\Logger\Adapter\File;

class MyDbListener
{
    protected $_profiler;

    protected $_logger;

    /**
     * Creates the profiler and starts the logging
     */
    public function __construct()
    {
        $this->_profiler = new Profiler();
        $this->_logger    = new Logger("../apps/logs/db.log");
    }

    /**
     * This is executed if the event triggered is 'beforeQuery'
     */
    public function beforeQuery(Event $event, $connection)
    {
        $this->_profiler->startProfile(
            $connection->getSQLStatement()
        );
    }

    /**
     * This is executed if the event triggered is 'afterQuery'
     */
    public function afterQuery(Event $event, $connection)
    {
        $this->_logger->log(
            $connection->getSQLStatement(),
            Logger::INFO
        );

        $this->_profiler->stopProfile();
    }

    public function getProfiler()
    {
        return $this->_profiler;
    }
}
```

Attaching an event listener to the events manager is as simple as:

```
<?php

// Create a database listener
$dbListener = new MyDbListener();

// Listen all the database events
$eventsManager->attach(
    "db",
    $dbListener
);
```

The resulting profile data can be obtained from the listener:

```
<?php

// Send a SQL command to the database server
$connection->execute(
    "SELECT * FROM products p WHERE p.status = 1"
);

foreach ($dbListener->getProfiler()->getProfiles() as $profile) {
    echo "SQL Statement: ", $profile->getSQLStatement(), "\n";
    echo "Start Time: ", $profile->getInitialTime(), "\n";
    echo "Final Time: ", $profile->getFinalTime(), "\n";
    echo "Total Elapsed Time: ", $profile->getTotalElapsedSeconds(), "\n";
}
```

Creating components that trigger Events

You can create components in your application that trigger events to an EventsManager. As a consequence, there may exist listeners that react to these events when generated. In the following example we’re creating a component called “MyComponent”. This component is EventsManager aware (it implements *Phalcon\Events\EventsAwareInterface*); when its `someTask()` method is executed it triggers two events to any listener in the EventsManager:

```
<?php

use Phalcon\Events\EventsAwareInterface;
use Phalcon\Events\Manager as EventsManager;

class MyComponent implements EventsAwareInterface
{
    protected $_eventsManager;

    public function setEventsManager(EventsManager $eventsManager)
    {
        $this->_eventsManager = $eventsManager;
    }

    public function getEventsManager()
    {
        return $this->_eventsManager;
    }
}
```



```

public function someTask()
{
    $this->_eventsManager->fire("my-component:beforeSomeTask", $this);

    // Do some task
    echo "Here, someTask\n";

    $this->_eventsManager->fire("my-component:afterSomeTask", $this);
}
}

```

Notice that in this example, we’re using the “my-component” event namespace. Now we need to create an event listener for this component:

```

<?php
use Phalcon\Events\Event;

class SomeListener
{
    public function beforeSomeTask(Event $event, $myComponent)
    {
        echo "Here, beforeSomeTask\n";
    }

    public function afterSomeTask(Event $event, $myComponent)
    {
        echo "Here, afterSomeTask\n";
    }
}

```

Now let’s make everything work together:

```

<?php
use Phalcon\Events\Manager as EventsManager;

// Create an Events Manager
$eventsManager = new EventsManager();

// Create the MyComponent instance
$myComponent = new MyComponent();

// Bind the eventsManager to the instance
$myComponent->setEventsManager($eventsManager);

// Attach the listener to the EventsManager
$eventsManager->attach(
    "my-component",
    new SomeListener()
);

// Execute methods in the component
$myComponent->someTask();

```

As `someTask()` is executed, the two methods in the listener will be executed, producing the following output:

```
Here, beforeSomeTask
Here, someTask
Here, afterSomeTask
```

Additional data may also be passed when triggering an event using the third parameter of `fire()`:

```
<?php
$eventsManager->fire("my-component:afterSomeTask", $this, $extraData);
```

In a listener the third parameter also receives this data:

```
<?php
use Phalcon\Events\Event;

// Receiving the data in the third parameter
$eventsManager->attach(
    "my-component",
    function (Event $event, $component, $data) {
        print_r($data);
    }
);

// Receiving the data from the event context
$eventsManager->attach(
    "my-component",
    function (Event $event, $component) {
        print_r($event->getData());
    }
);
```

Using Services From The DI

By extending *Phalcon\Mvc\User\Plugin*, you can access services from the DI, just like you would in a controller:

```
<?php
use Phalcon\Events\Event;
use Phalcon\Mvc\User\Plugin;

class SomeListener extends Plugin
{
    public function beforeSomeTask(Event $event, $myComponent)
    {
        echo "Here, beforeSomeTask\n";

        $this->logger->debug(
            "beforeSomeTask has been triggered";
        );
    }

    public function afterSomeTask(Event $event, $myComponent)
    {
        echo "Here, afterSomeTask\n";
    }
}
```

```

        $this->logger->debug(
            "afterSomeTask has been triggered";
        );
    }
}

```

Event Propagation/Cancellation

Many listeners may be added to the same event manager. This means that for the same type of event, many listeners can be notified. The listeners are notified in the order they were registered in the EventsManager. Some events are cancelable, indicating that these may be stopped preventing other listeners from being notified about the event:

```

<?php

use Phalcon\Events\Event;

$eventsManager->attach(
    "db",
    function (Event $event, $connection) {
        // We stop the event if it is cancelable
        if ($event->isCancelable()) {
            // Stop the event, so other listeners will not be notified about this
            $event->stop();
        }

        // ...
    }
);

```

By default, events are cancelable - even most of the events produced by the framework are cancelables. You can fire a not-cancelable event by passing `false` in the fourth parameter of `fire()`:

```

<?php

$eventsManager->fire("my-component:afterSomeTask", $this, $extraData, false);

```

Listener Priorities

When attaching listeners you can set a specific priority. With this feature you can attach listeners indicating the order in which they must be called:

```

<?php

$eventsManager->enablePriorities(true);

$eventsManager->attach("db", new DbListener(), 150); // More priority
$eventsManager->attach("db", new DbListener(), 100); // Normal priority
$eventsManager->attach("db", new DbListener(), 50);  // Less priority

```

Collecting Responses

The events manager can collect every response returned by every notified listener. This example explains how it works:

```
<?php

use Phalcon\Events\Manager as EventsManager;

$eventsManager = new EventsManager();

// Set up the events manager to collect responses
$eventsManager->collectResponses(true);

// Attach a listener
$eventsManager->attach(
    "custom:custom",
    function () {
        return "first response";
    }
);

// Attach a listener
$eventsManager->attach(
    "custom:custom",
    function () {
        return "second response";
    }
);

// Fire the event
$eventsManager->fire("custom:custom", null);

// Get all the collected responses
print_r($eventsManager->getResponses());
```

The above example produces:

```
Array ( [0] => first response [1] => second response )
```

Implementing your own EventsManager

The *Phalcon\Events\ManagerInterface* interface must be implemented to create your own EventsManager replacing the one provided by Phalcon.

2.3.25 Request Environment

Every HTTP request (usually originated by a browser) contains additional information regarding the request such as header data, files, variables, etc. A web based application needs to parse that information so as to provide the correct response back to the requester. *Phalcon\Http\Request* encapsulates the information of the request, allowing you to access it in an object-oriented way.

```
<?php

use Phalcon\Http\Request;

// Getting a request instance
$request = new Request();

// Check whether the request was made with method POST
```

```

if ($request->isPost()) {
    // Check whether the request was made with Ajax
    if ($request->isAjax()) {
        echo "Request was made using POST and AJAX";
    }
}

```

Getting Values

PHP automatically fills the superglobal arrays `$_GET` and `$_POST` depending on the type of the request. These arrays contain the values present in forms submitted or the parameters sent via the URL. The variables in the arrays are never sanitized and can contain illegal characters or even malicious code, which can lead to [SQL injection](#) or [Cross Site Scripting \(XSS\)](#) attacks.

Phalcon\Http\Request allows you to access the values stored in the `$_REQUEST`, `$_GET` and `$_POST` arrays and sanitize or filter them with the ‘filter’ service, (by default *Phalcon\Filter*). The following examples offer the same behavior:

```

<?php

use Phalcon\Filter;

$filter = new Filter();

// Manually applying the filter
$email = $filter->sanitize($_POST["user_email"], "email");

// Manually applying the filter to the value
$email = $filter->sanitize($request->getPost("user_email"), "email");

// Automatically applying the filter
$email = $request->getPost("user_email", "email");

// Setting a default value if the param is null
$email = $request->getPost("user_email", "email", "some@example.com");

// Setting a default value if the param is null without filtering
$email = $request->getPost("user_email", null, "some@example.com");

```

Accessing the Request from Controllers

The most common place to access the request environment is in an action of a controller. To access the *Phalcon\Http\Request* object from a controller you will need to use the `$this->request` public property of the controller:

```

<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {

    }
}

```

```
public function saveAction()
{
    // Check if request has made with POST
    if ($this->request->isPost()) {
        // Access POST data
        $customerName = $this->request->getPost("name");
        $customerBorn = $this->request->getPost("born");
    }
}
```

Uploading Files

Another common task is file uploading. *Phalcon\Http\Request* offers an object-oriented way to achieve this task:

```
<?php
use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function uploadAction()
    {
        // Check if the user has uploaded files
        if ($this->request->hasFiles()) {
            $files = $this->request->getUploadedFiles();

            // Print the real file names and sizes
            foreach ($files as $file) {
                // Print file details
                echo $file->getName(), " ", $file->getSize(), "\n";

                // Move the file into the application
                $file->moveTo(
                    "files/" . $file->getName()
                );
            }
        }
    }
}
```

Each object returned by `Phalcon\Http\Request::getUploadedFiles()` is an instance of the *Phalcon\Http\Request\File* class. Using the `$_FILES` superglobal array offers the same behavior. *Phalcon\Http\Request\File* encapsulates only the information related to each file uploaded with the request.

Working with Headers

As mentioned above, request headers contain useful information that allow us to send the proper response back to the user. The following examples show usages of that information:

```
<?php
// Get the Http-X-Requested-With header
$requestWith = $request->getHeader("HTTP_X_REQUESTED_WITH");
```

```

if ($requestedWith === "XMLHttpRequest") {
    echo "The request was made with Ajax";
}

// Same as above
if ($request->isAjax()) {
    echo "The request was made with Ajax";
}

// Check the request layer
if ($request->isSecure()) {
    echo "The request was made using a secure layer";
}

// Get the servers's IP address. ie. 192.168.0.100
$ipAddress = $request->getServerAddress();

// Get the client's IP address ie. 201.245.53.51
$ipAddress = $request->getClientAddress();

// Get the User Agent (HTTP_USER_AGENT)
$userAgent = $request->getUserAgent();

// Get the best acceptable content by the browser. ie text/xml
$contentType = $request->getAcceptableContent();

// Get the best charset accepted by the browser. ie. utf-8
$charset = $request->getBestCharset();

// Get the best language accepted configured in the browser. ie. en-us
$language = $request->getBestLanguage();

```

2.3.26 Returning Responses

Part of the HTTP cycle is returning responses to clients. *Phalcon\Http\Response* is the Phalcon component designed to achieve this task. HTTP responses are usually composed by headers and body. The following is an example of basic usage:

```

<?php

use Phalcon\Http\Response;

// Getting a response instance
$response = new Response();

// Set status code
$response->setStatusCode(404, "Not Found");

// Set the content of the response
$response->setContent("Sorry, the page doesn't exist");

// Send response to the client
$response->send();

```

If you are using the full MVC stack there is no need to create responses manually. However, if you need to return a

response directly from a controller's action follow this example:

```
<?php

use Phalcon\Http\Response;
use Phalcon\Mvc\Controller;

class FeedController extends Controller
{
    public function getAction()
    {
        // Getting a response instance
        $response = new Response();

        $feed = // ... Load here the feed

        // Set the content of the response
        $response->setContent(
            $feed->asString()
        );

        // Return the response
        return $response;
    }
}
```

Working with Headers

Headers are an important part of the HTTP response. It contains useful information about the response state like the HTTP status, type of response and much more.

You can set headers in the following way:

```
<?php

// Setting a header by its name
$response->setHeader("Content-Type", "application/pdf");
$response->setHeader("Content-Disposition", 'attachment; filename="downloaded.pdf"');

// Setting a raw header
$response->setRawHeader("HTTP/1.1 200 OK");
```

A *Phalcon\Http\Response\Headers* bag internally manages headers. This class retrieves the headers before sending it to client:

```
<?php

// Get the headers bag
$headers = $response->getHeaders();

// Get a header by its name
$contentType = $headers->get("Content-Type");
```

Making Redirections

With *Phalcon\Http\Response* you can also execute HTTP redirections:


```
<?php

// Redirect to the default URI
$response->redirect();

// Redirect to the local base URI
$response->redirect("posts/index");

// Redirect to an external URL
$response->redirect("http://en.wikipedia.org", true);

// Redirect specifying the HTTP status code
$response->redirect("http://www.example.com/new-location", true, 301);
```

All internal URIs are generated using the ‘url’ service (by default *Phalcon\Mvc\Url*). This example demonstrates how you can redirect using a route you have defined in your application:

```
<?php

// Redirect based on a named route
return $response->redirect(
    [
        "for"      => "index-lang",
        "lang"     => "jp",
        "controller" => "index",
    ]
);
```

Note that a redirection doesn’t disable the view component, so if there is a view associated with the current action it will be executed anyway. You can disable the view from a controller by executing `$this->view->disable()`;

HTTP Cache

One of the easiest ways to improve the performance in your applications and reduce the traffic is using HTTP Cache. Most modern browsers support HTTP caching and is one of the reasons why many websites are currently fast.

HTTP Cache can be altered in the following header values sent by the application when serving a page for the first time:

- *Expires*: With this header the application can set a date in the future or the past telling the browser when the page must expire.
- *Cache-Control*: This header allows to specify how much time a page should be considered fresh in the browser.
- *Last-Modified*: This header tells the browser which was the last time the site was updated avoiding page re-loads
- *ETag*: An etag is a unique identifier that must be created including the modification timestamp of the current page

Setting an Expiration Time

The expiration date is one of the easiest and most effective ways to cache a page in the client (browser). Starting from the current date we add the amount of time the page will be stored in the browser cache. Until this date expires no new content will be requested from the server:

```
<?php

$expiryDate = new DateTime();
$expiryDate->modify("+2 months");

$response->setExpires($expiryDate);
```

The Response component automatically shows the date in GMT timezone as expected in an Expires header.

If we set this value to a date in the past the browser will always refresh the requested page:

```
<?php

$expiryDate = new DateTime();
$expiryDate->modify("-10 minutes");

$response->setExpires($expiryDate);
```

Browsers rely on the client's clock to assess if this date has passed or not. The client clock can be modified to make pages expire and this may represent a limitation for this cache mechanism.

Cache-Control

This header provides a safer way to cache the pages served. We simply must specify a time in seconds telling the browser how long it must keep the page in its cache:

```
<?php

// Starting from now, cache the page for one day
$response->setHeader("Cache-Control", "max-age=86400");
```

The opposite effect (avoid page caching) is achieved in this way:

```
<?php

// Never cache the served page
$response->setHeader("Cache-Control", "private, max-age=0, must-revalidate");
```

E-Tag

An “entity-tag” or “E-tag” is a unique identifier that helps the browser realize if the page has changed or not between two requests. The identifier must be calculated taking into account that this must change if the previously served content has changed:

```
<?php

// Calculate the E-Tag based on the modification time of the latest news
$mostRecentDate = News::maximum(
    [
        "column" => "created_at"
    ]
);

$eTag = md5($mostRecentDate);
```

```
// Send an E-Tag header
$response->setHeader("E-Tag", $eTag);
```

2.3.27 Cookies Management

Cookies are a very useful way to store small pieces of data on the client's machine that can be retrieved even if the user closes his/her browser. *Phalcon\Http\Response\Cookies* acts as a global bag for cookies. Cookies are stored in this bag during the request execution and are sent automatically at the end of the request.

Basic Usage

You can set/get cookies by just accessing the 'cookies' service in any part of the application where services can be accessed:

```
<?php

use Phalcon\Mvc\Controller;

class SessionController extends Controller
{
    public function loginAction()
    {
        // Check if the cookie has previously set
        if ($this->cookies->has("remember-me")) {
            // Get the cookie
            $rememberMeCookie = $this->cookies->get("remember-me");

            // Get the cookie's value
            $value = $rememberMeCookie->getValue();
        }
    }

    public function startAction()
    {
        $this->cookies->set(
            "remember-me",
            "some value",
            time() + 15 * 86400
        );
    }

    public function logoutAction()
    {
        $rememberMeCookie = $this->cookies->get("remember-me");

        // Delete the cookie
        $rememberMeCookie->delete();
    }
}
```

Encryption/Decryption of Cookies

By default, cookies are automatically encrypted before being sent to the client and are decrypted when retrieved from the user. This protection prevents unauthorized users to see the cookies' contents in the client (browser). Despite this

protection, sensitive data should not be stored in cookies.

You can disable encryption in the following way:

```
<?php

use Phalcon\Http\Response\Cookies;

$di->set (
    "cookies",
    function () {
        $cookies = new Cookies();

        $cookies->useEncryption(false);

        return $cookies;
    }
);
```

If you wish to use encryption, a global key must be set in the 'crypt' service:

```
<?php

use Phalcon\Crypt;

$di->set (
    "crypt",
    function () {
        $crypt = new Crypt();

        $crypt->setKey('#1dj8$=dp?.ak//j1V$'); // Use your own key!

        return $crypt;
    }
);
```

Sending cookies data without encryption to clients including complex objects structures, resultsets, service information, etc. could expose internal application details that could be used by an attacker to attack the application. If you do not want to use encryption, we highly recommend you only send very basic cookie data like numbers or small string literals.

2.3.28 Generating URLs and Paths

Phalcon\Mvc\Url is the component responsible of generate URLs in a Phalcon application. It's capable of produce independent URLs based on routes.

Setting a base URI

Depending of which directory of your document root your application is installed, it may have a base URI or not.

For example, if your document root is /var/www/htdocs and your application is installed in /var/www/htdocs/invo then your baseUri will be /invo/. If you are using a VirtualHost or your application is installed on the document root, then your baseUri is /. Execute the following code to know the base URI detected by Phalcon:

```
<?php
```

```
use Phalcon\Mvc\Url;

$url = new Url();

echo $url->getBaseUri();
```

By default, Phalcon automatically may detect your baseUri, but if you want to increase the performance of your application is recommended setting up it manually:

```
<?php

use Phalcon\Mvc\Url;

$url = new Url();

// Setting a relative base URI
$url->setBaseUri("/invo/");

// Setting a full domain as base URI
$url->setBaseUri("//my.domain.com/");

// Setting a full domain as base URI
$url->setBaseUri("http://my.domain.com/my-app/");
```

Usually, this component must be registered in the Dependency Injector container, so you can set up it there:

```
<?php

use Phalcon\Mvc\Url;

$di->set(
    "url",
    function () {
        $url = new Url();

        $url->setBaseUri("/invo/");

        return $url;
    }
);
```

Generating URIs

If you are using the *Router* with its default behavior. Your application is able to match routes based on the following pattern: `/:controller/:action/:params`. Accordingly it is easy to create routes that satisfy that pattern (or any other pattern defined in the router) passing a string to the method “get”:

```
<?php echo $url->get("products/save"); ?>
```

Note that isn’t necessary to prepend the base URI. If you have named routes you can easily change it creating it dynamically. For Example if you have the following route:

```
<?php

$router->add(
    "/blog/{year}/{month}/{title}",
```

```
[
    "controller" => "posts",
    "action"      => "show",
]
)->setName("show-post");
```

A URL can be generated in the following way:

```
<?php

// This produces: /blog/2015/01/some-blog-post
$url->get(
    [
        "for"    => "show-post",
        "year"   => "2015",
        "month"  => "01",
        "title"  => "some-blog-post",
    ]
);
```

Producing URLs without mod_rewrite

You can use this component also to create URLs without mod_rewrite:

```
<?php

use Phalcon\Mvc\Url;

$url = new Url();

// Pass the URI in $_GET["_url"]
$url->setBaseUri("/invo/index.php?_url=/");

// This produce: /invo/index.php?_url=/products/save
echo $url->get("products/save");
```

You can also use `$_SERVER["REQUEST_URI"]`:

```
<?php

use Phalcon\Mvc\Url;

$url = new Url();

// Pass the URI in $_GET["_url"]
$url->setBaseUri("/invo/index.php?_url=/");

// Pass the URI using $_SERVER["REQUEST_URI"]
$url->setBaseUri("/invo/index.php/");
```

In this case, it's necessary to manually handle the required URI in the Router:

```
<?php

use Phalcon\Mvc Router;
```

```
$router = new Router();

// ... Define routes

$uri = str_replace($_SERVER["SCRIPT_NAME"], "", $_SERVER["REQUEST_URI"]);

$router->handle($uri);
```

The produced routes would look like:

```
<?php

// This produce: /invo/index.php/products/save
echo $url->get("products/save");
```

Producing URLs from Volt

The function “url” is available in volt to generate URLs using this component:

```
<a href="{{ url("posts/edit/1002") }}">Edit</a>
```

Generate static routes:

```
<link rel="stylesheet" href="{{ static_url("css/style.css") }}" type="text/css" />
```

Static vs. Dynamic URIs

This component allow you to set up a different base URI for static resources in the application:

```
<?php

use Phalcon\Mvc\Url;

$url = new Url();

// Dynamic URIs are
$url->setBaseUri("/");

// Static resources go through a CDN
$url->setStaticBaseUri("http://static.mywebsite.com/");
```

Phalcon\Tag will request both dynamical and static URIs using this component.

Implementing your own URL Generator

The *Phalcon\Mvc\UrlInterface* interface must be implemented to create your own URL generator replacing the one provided by Phalcon.

2.3.29 Flashing Messages

Flash messages are used to notify the user about the state of actions he/she made or simply show information to the users. These kinds of messages can be generated using this component.

Adapters

This component makes use of adapters to define the behavior of the messages after being passed to the Flasher:

Adapter	Description	API
Direct	Directly outputs the messages passed to the flasher	<i>Phalcon\Flash\Direct</i>
Session	Temporarily stores the messages in session, then messages can be printed in the next request	<i>Phalcon\Flash\Session</i>

Usage

Usually the Flash Messaging service is requested from the services container. If you're using *Phalcon\Di\FactoryDefault* then *Phalcon\Flash\Direct* is automatically registered as “flash” service and *Phalcon\Flash\Session* is automatically registered as “flashSession” service. You can also manually register it:

```
<?php

use Phalcon\Flash\Direct as FlashDirect;
use Phalcon\Flash\Session as FlashSession;

// Set up the flash service
$di->set(
    "flash",
    function () {
        return new FlashDirect();
    }
);

// Set up the flash session service
$di->set(
    "flashSession",
    function () {
        return new FlashSession();
    }
);
```

This way, you can use it in controllers or views:

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function indexAction()
    {

    }

    public function saveAction()
    {
        $this->flash->success("The post was correctly saved!");
    }
}
```

There are four built-in message types supported:


```
<?php

$this->flash->error("too bad! the form had errors");

$this->flash->success("yes!, everything went very smoothly");

$this->flash->notice("this a very important information");

$this->flash->warning("best check yo self, you're not looking too good.");
```

You can also add messages with your own types using the `message()` method:

```
<?php

$this->flash->message("debug", "this is debug message, you don't say");
```

Printing Messages

Messages sent to the flash service are automatically formatted with HTML:

```
<div class="errorMessage">too bad! the form had errors</div>

<div class="successMessage">yes!, everything went very smoothly</div>

<div class="noticeMessage">this a very important information</div>

<div class="warningMessage">best check yo self, you're not looking too good.</div>
```

As you can see, CSS classes are added automatically to the `<div>`'s. These classes allow you to define the graphical presentation of the messages in the browser. The CSS classes can be overridden, for example, if you're using Twitter Bootstrap, classes can be configured as:

```
<?php

use Phalcon\Flash\Direct as FlashDirect;

// Register the flash service with custom CSS classes
$di->set(
    "flash",
    function () {
        $flash = new FlashDirect(
            [
                "error"    => "alert alert-danger",
                "success"  => "alert alert-success",
                "notice"   => "alert alert-info",
                "warning"  => "alert alert-warning",
            ]
        );

        return $flash;
    }
);
```

Then the messages would be printed as follows:

```
<div class="alert alert-danger">too bad! the form had errors</div>

<div class="alert alert-success">yes!, everything went very smoothly</div>

<div class="alert alert-info">this a very important information</div>

<div class="alert alert-warning">best check yo self, you're not looking too good.</
↪div>
```

Implicit Flush vs. Session

Depending on the adapter used to send the messages, it could be producing output directly, or be temporarily storing the messages in session to be shown later. When should you use each? That usually depends on the type of redirection you do after sending the messages. For example, if you make a “forward” is not necessary to store the messages in session, but if you do a HTTP redirect then, they need to be stored in session:

```
<?php

use Phalcon\Mvc\Controller;

class ContactController extends Controller
{
    public function indexAction()
    {

    }

    public function saveAction()
    {
        // Store the post

        // Using direct flash
        $this->flash->success("Your information was stored correctly!");

        // Forward to the index action
        return $this->dispatcher->forward(
            [
                "action" => "index"
            ]
        );
    }
}
```

Or using a HTTP redirection:

```
<?php

use Phalcon\Mvc\Controller;

class ContactController extends Controller
{
    public function indexAction()
    {

    }
}
```

```

public function saveAction()
{
    // Store the post

    // Using session flash
    $this->flashSession->success("Your information was stored correctly!");

    // Make a full HTTP redirection
    return $this->response->redirect("contact/index");
}

```

In this case you need to manually print the messages in the corresponding view:

```

<!-- app/views/contact/index.phtml -->

<p><?php $this->flashSession->output() ?></p>

```

The attribute 'flashSession' is how the flash was previously set into the dependency injection container. You need to start the *session* first to successfully use the flashSession messenger.

Attention: cette traduction n'est pas parfaite, si des éléments vous paraissent faux ou mal expliqués, merci de modifier la documentation.

2.3.30 Stocker des données dans une session

Le composant de session fournit un ensemble de fonctions liés à la gestion des sessions.

Pourquoi utiliser ce composant plutôt que les sessions par défaut ?

- Vous pouvez facilement isoler des données de session à travers différentes applications du même domaine.
- Vous pouvez intercepter où les sessions sont définies/récupérées (get/set) dans votre application.
- Changer l'adaptateur de session en fonction de vos besoins.

Commencer la Session

Certaines applications sont très dépendantes des sessions, pratiquement chaque action qui est réalisée nécessite l'accès aux données de sessions. Il y a aussi d'autres applications qui nécessiteront moins les sessions. Grâce au conteneur de services, on peut s'assurer que les sessions sont accessibles seulement quand nécessaire :

```

<?php

use Phalcon\Session\Adapter\Files as Session;

// Start the session the first time when some component request the session service
$di->setShared(
    "session",
    function () {
        $session = new Session();

        $session->start();

        return $session;
    }
);

```

Stocker/Récupérer les données en session

A partir d'un contrôleur, d'une vue ou de n'importe quel autre composant qui hérite de *Phalcon\DI\Injectable* vous pourrez accéder aux services de session et stocker/récupérer des informations de cette manière :

```
<?php

use Phalcon\Mvc\Controller;

class UserController extends Controller
{
    public function indexAction()
    {
        // Set a session variable
        $this->session->set("user-name", "Michael");
    }

    public function welcomeAction()
    {
        // Check if the variable is defined
        if ($this->session->has("user-name")) {
            // Retrieve its value
            $name = $this->session->get("user-name");
        }
    }
}
```

Supprimer / Détruire des sessions

Il est aussi tout à fait possible de supprimer des variables spécifiques de session ou de supprimer la session entièrement :

```
<?php

use Phalcon\Mvc\Controller;

class UserController extends Controller
{
    public function removeAction()
    {
        // Remove a session variable
        $this->session->remove("user-name");
    }

    public function logoutAction()
    {
        // Destroy the whole session
        $this->session->destroy();
    }
}
```

Isoler les données de sessions entre les applications

Des fois un utilisateur peut utiliser la même application plusieurs fois sur le même serveur, dans la même session. Bien sûr, si on utilise des variables de session, nous voulons que chaque application ait ses propres données (même s'ils doivent utiliser les même noms de variable). Pour résoudre ce problème, vous pouvez ajouter un prefix pour chaque sessions de variable créé dans une certaine application :

```
<?php

use Phalcon\Session\Adapter\Files as Session;

// Isolating the session data
$di->set(
    "session",
    function () {
        // All variables created will prefixed with "my-app-1"
        $session = new Session(
            [
                "uniqueId" => "my-app-1",
            ]
        );

        $session->start();

        return $session;
    }
);
```

Adding a unique ID is not necessary.

Sac de Session

Phalcon\Session\Bag est un composant qui aide à séparer les données de sessions dans des “espaces de noms”. En travaillant de cette manière on peut facilement créer des groupes de sessions dans l’application. En plaçant les variables dans le “sac”, cela stocke automatiquement les données dans la session :

```
<?php

use Phalcon\Session\Bag as SessionBag;

$user = new SessionBag("user");

$user->setDI($di);

$user->name = "Kimbra Johnson";
$user->age  = 22;
```

Données persistantes dans les composants

Les contrôleurs, composants et classes qui héritent de *Phalcon\Di\Injectable* peuvent injecter un *Phalcon\Session\Bag*. Cette classe isole les variables pour chaque classes. Grace à cela, vous pouvez faire durer vos données entre les requêtes de chaque classes de manière indépendantes.

```
<?php
```

```
use Phalcon\Mvc\Controller;

class UserController extends Controller
{
    public function indexAction()
    {
        // Create a persistent variable "name"
        $this->persistent->name = "Laura";
    }

    public function welcomeAction()
    {
        if (isset($this->persistent->name)) {
            echo "Welcome, ", $this->persistent->name;
        }
    }
}
```

Dans un composant :

```
<?php

use Phalcon\Mvc\Controller;

class Security extends Component
{
    public function auth()
    {
        // Create a persistent variable "name"
        $this->persistent->name = "Laura";
    }

    public function getAuthName()
    {
        return $this->persistent->name;
    }
}
```

Les données ajoutés à la session (`$this->session`) sont disponibles à travers toute l'application, tandis qu'avec `$this->persistent`, on ne peut y accéder qu'à partir de la portée de la classe courante.

Implémenter son propre adaptateur

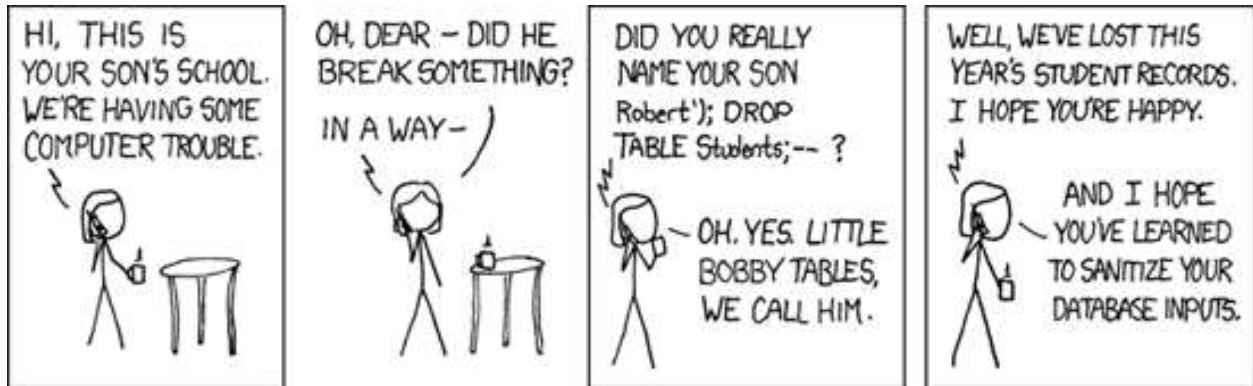
Phalcon\Session\AdapterInterface est une interface qui doit être implémentée pour créer ses propres adaptateurs de session ou hériter d'adaptateurs déjà existants.

Il y a plus d'adaptateur disponibles pour ce composant dans l'[Incubator Phalcon](#)

2.3.31 Filtering and Sanitizing

Sanitizing user input is a critical part of software development. Trusting or neglecting to sanitize user input could lead to unauthorized access to the content of your application, mainly user data, or even the server your application is hosted on.

Full image (from xkcd)



The *Phalcon\Filter* component provides a set of commonly used filters and data sanitizing helpers. It provides object-oriented wrappers around the PHP filter extension.

Types of Built-in Filters

The following are the built-in filters provided by this component:

Name	Description
string	Strip tags and encode HTML entities, including single and double quotes.
email	Remove all characters except letters, digits and !#\$%&*+/-=?^_`{ }~@.[].
int	Remove all characters except digits, plus and minus sign.
float	Remove all characters except digits, dot, plus and minus sign.
alphanumeric	Remove all characters except [a-zA-Z0-9]
striptags	Applies the <code>strip_tags</code> function
trim	Applies the <code>trim</code> function
lower	Applies the <code>strtolower</code> function
upper	Applies the <code>strtoupper</code> function

Sanitizing data

Sanitizing is the process which removes specific characters from a value, that are not required or desired by the user or application. By sanitizing input we ensure that application integrity will be intact.

```
<?php

use Phalcon\Filter;

$filter = new Filter();

// Returns "someone@example.com"
$filter->sanitize("some(one)@exa\mple.com", "email");

// Returns "hello"
$filter->sanitize("hello<<", "string");

// Returns "100019"
$filter->sanitize("!100a019", "int");

// Returns "100019.01"
$filter->sanitize("!100a019.01a", "float");
```

Sanitizing from Controllers

You can access a *Phalcon\Filter* object from your controllers when accessing GET or POST input data (through the request object). The first parameter is the name of the variable to be obtained; the second is the filter to be applied on it.

```
<?php

use Phalcon\Mvc\Controller;

class ProductsController extends Controller
{
    public function indexAction()
    {

    }

    public function saveAction()
    {
        // Sanitizing price from input
        $price = $this->request->getPost("price", "double");

        // Sanitizing email from input
        $email = $this->request->getPost("customerEmail", "email");
    }
}
```

Filtering Action Parameters

The next example shows you how to sanitize the action parameters within a controller action:

```
<?php

use Phalcon\Mvc\Controller;

class ProductsController extends Controller
{
    public function indexAction()
    {

    }

    public function showAction($productId)
    {
        $productId = $this->filter->sanitize($productId, "int");
    }
}
```

Filtering data

In addition to sanitizing, *Phalcon\Filter* also provides filtering by removing or modifying input data to the format we expect.

```
<?php
```



```

use Phalcon\Filter;

$filter = new Filter();

// Returns "Hello"
$filter->sanitize("<h1>Hello</h1>", "striptags");

// Returns "Hello"
$filter->sanitize(" Hello ", "trim");

```

Combining Filters

You can also run multiple filters on a string at the same time by passing an array of filter identifiers as the second parameter:

```

<?php

use Phalcon\Filter;

$filter = new Filter();

// Returns "Hello"
$filter->sanitize(
    "<h1> Hello </h1> ",
    [
        "striptags",
        "trim",
    ]
);

```

Creating your own Filters

You can add your own filters to *Phalcon\Filter*. The filter function could be an anonymous function:

```

<?php

use Phalcon\Filter;

$filter = new Filter();

// Using an anonymous function
$filter->add(
    "md5",
    function ($value) {
        return preg_replace("/[^0-9a-f]/", "", $value);
    }
);

// Sanitize with the "md5" filter
$filtered = $filter->sanitize($possibleMd5, "md5");

```

Or, if you prefer, you can implement the filter in a class:

```

<?php

```

```
use Phalcon\Filter;

class IPv4Filter
{
    public function filter($value)
    {
        return filter_var($value, FILTER_VALIDATE_IP, FILTER_FLAG_IPV4);
    }
}

$filter = new Filter();

// Using an object
$filter->add(
    "ipv4",
    new IPv4Filter()
);

// Sanitize with the "ipv4" filter
$filteredIp = $filter->sanitize("127.0.0.1", "ipv4");
```

Complex Sanitizing and Filtering

PHP itself provides an excellent filter extension you can use. Check out its documentation: [Data Filtering at PHP Documentation](#)

Implementing your own Filter

The *Phalcon\FilterInterface* interface must be implemented to create your own filtering service replacing the one provided by Phalcon.

2.3.32 Contextual Escaping

Websites and web applications are vulnerable to [XSS](#) attacks and although PHP provides escaping functionality, in some contexts it is not sufficient/appropriate. *Phalcon\Escaper* provides contextual escaping and is written in Zephir, providing the minimal overhead when escaping different kinds of texts.

We designed this component based on the [XSS \(Cross Site Scripting\) Prevention Cheat Sheet](#) created by the OWASP.

Additionally, this component relies on [mbstring](#) to support almost any charset.

To illustrate how this component works and why it is important, consider the following example:

```
<?php

use Phalcon\Escaper;

// Document title with malicious extra HTML tags
$maliciousTitle = "</title><script>alert(1)</script>";

// Malicious CSS class name
$className = ";`(";

// Malicious CSS font name
$fontName = "Verdana\"</style>";
```

```
// Malicious Javascript text
$javascriptText = "<script>Hello";

// Create an escaper
$e = new Escaper();

?>

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

    <title>
      <?php echo $e->escapeHtml($maliciousTitle); ?>
    </title>

    <style type="text/css">
      .<?php echo $e->escapeCss($className); ?> {
        font-family: "<?php echo $e->escapeCss($fontName); ?>";
        color: red;
      }
    </style>

  </head>

  <body>

    <div class='<?php echo $e->escapeHtmlAttr($className); ?>'>
      hello
    </div>

    <script>
      var some = '<?php echo $e->escapeJs($javascriptText); ?>';
    </script>

  </body>
</html>
```

Which produces the following:

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

    <title>
      &lt;/title&gt;&lt;script&gt;alert(1)&lt;/script&gt;
    </title>

    <style type="text/css">
      .\3c \2f style\3e {
        font-family: "Verdana\22 \3c \2f style\3e";
        color: red;
      }
    </style>

  </head>
```

```
<body>

    <div class='&#x3c; &#x2f; style&#x3e; '>
        hello
    </div>

    <script>
        var some = '\x27\x3b\x3c\x2fscript\x3eHello';
    </script>

</body>
</html>
```

Every text was escaped according to its context. Use the appropriate context is important to avoid XSS attacks.

Escaping HTML

The most common situation when inserting unsafe data is between HTML tags:

```
<div class="comments">
    <!-- Escape untrusted data here! -->
</div>
```

You can escape those data using the `escapeHtml` method:

```
<div class="comments">
    <?php echo $e->escapeHtml('></div><h1>myattack</h1>'); ?>
</div>
```

Which produces:

```
<div class="comments">
    &gt;&lt;&lt;/div&gt;&lt;h1&gt;myattack&lt;/h1&gt;
</div>
```

Escaping HTML Attributes

Escaping HTML attributes is different from escaping HTML content. The escaper works by changing every non-alphanumeric character to the form. This kind of escaping is intended to most simpler attributes excluding complex ones like 'href' or 'url':

```
<table width="Escape untrusted data here!">
    <tr>
        <td>
            Hello
        </td>
    </tr>
</table>
```

You can escape a HTML attribute by using the `escapeHtmlAttr` method:

```
<table width="<?php echo $e->escapeHtmlAttr('><h1>Hello</table>'); ?>">
    <tr>
        <td>
            Hello
        </td>
    </tr>
</table>
```

```

        </td>
    </tr>
</table>

```

Which produces:

```

<table width="&#x22;&#x3e;&#x3c;h1&#x3e;Hello&#x3c;&#x2f;table">
  <tr>
    <td>
      Hello
    </td>
  </tr>
</table>

```

Escaping URLs

Some HTML attributes like ‘href’ or ‘url’ need to be escaped differently:

```

<a href="Escape untrusted data here!">
  Some link
</a>

```

You can escape a HTML attribute by using the `escapeUrl` method:

```

<a href="<?php echo $e->escapeUrl('><script>alert(1)</script><a href="#"'); ?>">
  Some link
</a>

```

Which produces:

```

<a href="%22%3E%3Cscript%3Ealert%281%29%3C%2Fscript%3E%3Ca%20href%3D%22%23">
  Some link
</a>

```

Escaping CSS

CSS identifiers/values can be escaped too:

```

<a style="color: Escape untrusted data here">
  Some link
</a>

```

You can escape a CSS identifiers/value by using the `escapeCss` method:

```

<a style="color: <?php echo $e->escapeCss('><script>alert(1)</script><a href="#"'); ?>">
  Some link
</a>

```

Which produces:

```

<a style="color: \22 \3e \3c script\3e alert\28 1\29 \3c \2f script\3e \3c a\20
  href\3d \22 \23 ">
  Some link
</a>

```

Escaping JavaScript

Strings to be inserted into JavaScript code also must be properly escaped:

```
<script>
    document.title = 'Escape untrusted data here';
</script>
```

You can escape JavaScript code by using the `escapeJs` method:

```
<script>
    document.title = '<?php echo $e->escapeJs("&#039;; alert(100); var x='"); ?>';
</script>
```

```
<script>
    document.title = '\x27; alert(100); var x\x3d\x27';
</script>
```

2.3.33 Validation

Phalcon\Validation is an independent validation component that validates an arbitrary set of data. This component can be used to implement validation rules on data objects that do not belong to a model or collection.

The following example shows its basic usage:

```
<?php

use Phalcon\Validation;
use Phalcon\Validation\Validator\Email;
use Phalcon\Validation\Validator\PresenceOf;

$validation = new Validation();

$validation->add(
    "name",
    new PresenceOf(
        [
            "message" => "The name is required",
        ]
    )
);

$validation->add(
    "email",
    new PresenceOf(
        [
            "message" => "The e-mail is required",
        ]
    )
);

$validation->add(
    "email",
    new Email(
        [
            "message" => "The e-mail is not valid",
        ]
    )
);
```

```

    )
);

$messages = $validation->validate($_POST);

if (count($messages)) {
    foreach ($messages as $message) {
        echo $message, "<br>";
    }
}

```

The loosely-coupled design of this component allows you to create your own validators along with the ones provided by the framework.

Initializing Validation

Validation chains can be initialized in a direct manner by just adding validators to the *Phalcon\Validation* object. You can put your validations in a separate file for better re-use code and organization:

```

<?php

use Phalcon\Validation;
use Phalcon\Validation\Validator\Email;
use Phalcon\Validation\Validator\PresenceOf;

class MyValidation extends Validation
{
    public function initialize()
    {
        $this->add(
            "name",
            new PresenceOf(
                [
                    "message" => "The name is required",
                ]
            )
        );

        $this->add(
            "email",
            new PresenceOf(
                [
                    "message" => "The e-mail is required",
                ]
            )
        );

        $this->add(
            "email",
            new Email(
                [
                    "message" => "The e-mail is not valid",
                ]
            )
        );
    }
}

```

Then initialize and use your own validator:

```
<?php

$validation = new MyValidation();

$messages = $validation->validate($_POST);

if (count($messages)) {
    foreach ($messages as $message) {
        echo $message, "<br>";
    }
}
```

Validators

Phalcon exposes a set of built-in validators for this component:

The following example explains how to create additional validators for this component:

```
<?php

use Phalcon\Validation;
use Phalcon\Validation\Message;
use Phalcon\Validation\Validator;

class IpValidator extends Validator
{
    /**
     * Executes the validation
     *
     * @param Phalcon\Validation $validator
     * @param string $attribute
     * @return boolean
     */
    public function validate(Validation $validator, $attribute)
    {
        $value = $validator->getValue($attribute);

        if (!filter_var($value, FILTER_VALIDATE_IP, FILTER_FLAG_IPV4 | FILTER_FLAG_
→IPV6)) {
            $message = $this->getOption("message");

            if (!$message) {
                $message = "The IP is not valid";
            }

            $validator->appendMessage(
                new Message($message, $attribute, "Ip")
            );

            return false;
        }

        return true;
    }
}
```



```
}
}
```

It is important that validators return a valid boolean value indicating if the validation was successful or not.

Callback Validator

By using `Phalcon\Validation\Validator\Callback` you can execute custom function which must return boolean or new validator class which will be used to validate the same field. By returning `true` validation will be successful, returning `false` will mean validation failed. When executing this validator Phalcon will pass data depending what it is - if it's an entity then entity will be passed, otherwise data. There is example:

```
<?php

use \Phalcon\Validation;
use \Phalcon\Validation\Validator\Callback;
use \Phalcon\Validation\Validator\PresenceOf;

$validation = new Validation();
$validation->add(
    "amount",
    new Callback(
        [
            "callback" => function($data) {
                return $data["amount"] % 2 == 0;
            },
            "message" => "Only even number of products are accepted"
        ]
    )
);
$validation->add(
    "amount",
    new Callback(
        [
            "callback" => function($data) {
                if($data["amount"] % 2 == 0) {
                    return $data["amount"] != 2;
                }

                return true;
            },
            "message" => "You can't buy 2 products"
        ]
    )
);
$validation->add(
    "description",
    new Callback(
        [
            "callback" => function($data) {
                if($data["amount"] >= 10) {
                    return new PresenceOf(
                        [
                            "message" => "You must write why you need so big amount."
                        ]
                    );
                }
            }
        ]
    );
}
```

```
        return true;
    }
    ]
)
);

$messages = $validation->validate(["amount" => 1]); // will return message from first_
↳ validator
$messages = $validation->validate(["amount" => 2]); // will return message from_
↳ second validator
$messages = $validation->validate(["amount" => 10]); // will return message from_
↳ validator returned by third validator
```

Validation Messages

Phalcon\Validation has a messaging subsystem that provides a flexible way to output or store the validation messages generated during the validation processes.

Each message consists of an instance of the class *Phalcon\Validation\Message*. The set of messages generated can be retrieved with the `getMessages()` method. Each message provides extended information like the attribute that generated the message or the message type:

```
<?php

$messages = $validation->validate();

if (count($messages)) {
    foreach ($messages as $message) {
        echo "Message: ", $message->getMessage(), "\n";
        echo "Field: ", $message->getField(), "\n";
        echo "Type: ", $message->getType(), "\n";
    }
}
```

You can pass a ‘message’ parameter to change/translate the default message in each validator:

```
<?php

use Phalcon\Validation\Validator\Email;

$validation->add(
    "email",
    new Email(
        [
            "message" => "The e-mail is not valid",
        ]
    )
);
```

By default, the `getMessages()` method returns all the messages generated during validation. You can filter messages for a specific field using the `filter()` method:

```
<?php

$messages = $validation->validate();
```

```

if (count($messages)) {
    // Filter only the messages generated for the field 'name'
    $filteredMessages = $messages->filter("name");

    foreach ($filteredMessages as $message) {
        echo $message;
    }
}

```

Filtering of Data

Data can be filtered prior to the validation ensuring that malicious or incorrect data is not validated.

```

<?php

use Phalcon\Validation;

$validation = new Validation();

$validation->add(
    "name",
    new PresenceOf(
        [
            "message" => "The name is required",
        ]
    )
);

$validation->add(
    "email",
    new PresenceOf(
        [
            "message" => "The email is required",
        ]
    )
);

// Filter any extra space
$validation->setFilters("name", "trim");
$validation->setFilters("email", "trim");

```

Filtering and sanitizing is performed using the *filter* component. You can add more filters to this component or use the built-in ones.

Validation Events

When validations are organized in classes, you can implement the `beforeValidation()` and `afterValidation()` methods to perform additional checks, filters, clean-up, etc. If the `beforeValidation()` method returns false the validation is automatically cancelled:

```

<?php

use Phalcon\Validation;

```

```
class LoginValidation extends Validation
{
    public function initialize()
    {
        // ...
    }

    /**
     * Executed before validation
     *
     * @param array $data
     * @param object $entity
     * @param Phalcon\Validation\Message\Group $messages
     * @return bool
     */
    public function beforeValidation($data, $entity, $messages)
    {
        if ($this->request->getHttpHost() !== "admin.mydomain.com") {
            $messages->appendMessage(
                new Message("Only users can log on in the administration domain")
            );

            return false;
        }

        return true;
    }

    /**
     * Executed after validation
     *
     * @param array $data
     * @param object $entity
     * @param Phalcon\Validation\Message\Group $messages
     */
    public function afterValidation($data, $entity, $messages)
    {
        // ... Add additional messages or perform more validations
    }
}
```

Cancelling Validations

By default all validators assigned to a field are tested regardless if one of them have failed or not. You can change this behavior by telling the validation component which validator may stop the validation:

```
<?php

use Phalcon\Validation;
use Phalcon\Validation\Validator\Regex;
use Phalcon\Validation\Validator\PresenceOf;

$validation = new Validation();

$validation->add(
    "telephone",
```

```

    new PresenceOf(
        [
            "message"      => "The telephone is required",
            "cancelOnFail" => true,
        ]
    )
);

$validation->add(
    "telephone",
    new Regex(
        [
            "message" => "The telephone is required",
            "pattern" => "/\+44 [0-9]+/",
        ]
    )
);

$validation->add(
    "telephone",
    new StringLength(
        [
            "messageMinimum" => "The telephone is too short",
            "min"             => 2,
        ]
    )
);

```

The first validator has the option ‘cancelOnFail’ with a value of true, therefore if that validator fails the remaining validators in the chain are not executed.

If you are creating custom validators you can dynamically stop the validation chain by setting the ‘cancelOnFail’ option:

```

<?php

use Phalcon\Validation;
use Phalcon\Validation\Message;
use Phalcon\Validation\Validator;

class MyValidator extends Validator
{
    /**
     * Executes the validation
     *
     * @param Phalcon\Validation $validator
     * @param string $attribute
     * @return boolean
     */
    public function validate(Validation $validator, $attribute)
    {
        // If the attribute value is name we must stop the chain
        if ($attribute === "name") {
            $validator->setOption("cancelOnFail", true);
        }

        // ...
    }
}

```

```
}
```

Avoid validate empty values

You can pass the option ‘allowEmpty’ to all the built-in validators to avoid the validation to be performed if an empty value is passed:

```
<?php

use Phalcon\Validation;
use Phalcon\Validation\Validator\Regex;

$validation = new Validation();

$validation->add(
    "telephone",
    new Regex(
        [
            "message"    => "The telephone is required",
            "pattern"     => "/\+44 [0-9]+/",
            "allowEmpty" => true,
        ]
    )
);
```

Recursive Validation

You can also run Validation instances within another via the `afterValidation()` method. In this example, validating the `CompanyValidation` instance will also check the `PhoneValidation` instance:

```
<?php

use Phalcon\Validation;

class CompanyValidation extends Validation
{
    /**
     * @var PhoneValidation
     */
    protected $phoneValidation;

    public function initialize()
    {
        $this->phoneValidation = new PhoneValidation();
    }

    public function afterValidation($data, $entity, $messages)
    {
        $phoneValidationMessages = $this->phoneValidation->validate(
            $data["phone"]
        );
    }
}
```

```

        $messages->appendMessages (
            $phoneValidationMessages
        );
    }
}

```

2.3.34 Forms

Phalcon\Forms is a component that aids you in the creation and maintenance of forms in web applications.

The following example shows its basic usage:

```

<?php

use Phalcon\Forms\Form;
use Phalcon\Forms\Element\Text;
use Phalcon\Forms\Element\Select;

$form = new Form();

$form->add(
    new Text (
        "name"
    )
);

$form->add(
    new Text (
        "telephone"
    )
);

$form->add(
    new Select (
        "telephoneType",
        [
            "H" => "Home",
            "C" => "Cell",
        ]
    )
);

```

Forms can be rendered based on the form definition:

```

<h1>
    Contacts
</h1>

<form method="post">

    <p>
        <label>
            Name
        </label>

        <?php echo $form->render("name"); ?>
    
```

```
</p>

<p>
    <label>
        Telephone
    </label>

    <?php echo $form->render("telephone"); ?>
</p>

<p>
    <label>
        Type
    </label>

    <?php echo $form->render("telephoneType"); ?>
</p>

<p>
    <input type="submit" value="Save" />
</p>

</form>
```

Each element in the form can be rendered as required by the developer. Internally, *Phalcon\Tag* is used to produce the correct HTML for each element and you can pass additional HTML attributes as the second parameter of `render()`:

```
<p>
    <label>
        Name
    </label>

    <?php echo $form->render("name", ["maxlength" => 30, "placeholder" => "Type your_
↪name"]); ?>
</p>
```

HTML attributes also can be set in the element's definition:

```
<?php
$form->add(
    new Text(
        "name",
        [
            "maxlength" => 30,
            "placeholder" => "Type your name",
        ]
    )
);
```

Initializing forms

As seen before, forms can be initialized outside the form class by adding elements to it. You can re-use code or organize your form classes implementing the form in a separated file:


```

<?php

use Phalcon\Forms\Form;
use Phalcon\Forms\Element\Text;
use Phalcon\Forms\Element\Select;

class ContactForm extends Form
{
    public function initialize()
    {
        $this->add(
            new Text(
                "name"
            )
        );

        $this->add(
            new Text(
                "telephone"
            )
        );

        $this->add(
            new Select(
                "telephoneType",
                TelephoneTypes::find(),
                [
                    "using" => [
                        "id",
                        "name",
                    ]
                ]
            )
        );
    }
}

```

Phalcon\Forms\Form extends *Phalcon\DI\Injectable* so you have access to the application services if needed:

```

<?php

use Phalcon\Forms\Form;
use Phalcon\Forms\Element\Text;
use Phalcon\Forms\Element\Hidden;

class ContactForm extends Form
{
    /**
     * This method returns the default value for field 'csrf'
     */
    public function getCsrft()
    {
        return $this->security->getToken();
    }

    public function initialize()
    {
        // Set the same form as entity
    }
}

```

```
$this->setEntity($this);

// Add a text element to capture the 'email'
$this->add(
    new Text(
        "email"
    )
);

// Add a text element to put a hidden CSRF
$this->add(
    new Hidden(
        "csrf"
    )
);
}
```

The associated entity added to the form in the initialization and custom user options are passed to the form constructor:

```
<?php

use Phalcon\Forms\Form;
use Phalcon\Forms\Element\Text;
use Phalcon\Forms\Element\Hidden;

class UsersForm extends Form
{
    /**
     * Forms initializer
     *
     * @param Users $user
     * @param array $options
     */
    public function initialize(Users $user, array $options)
    {
        if ($options["edit"]) {
            $this->add(
                new Hidden(
                    "id"
                )
            );
        } else {
            $this->add(
                new Text(
                    "id"
                )
            );
        }

        $this->add(
            new Text(
                "name"
            )
        );
    }
}
```

In the form's instantiation you must use:

```
<?php

$form = new UsersForm(
    new Users(),
    [
        "edit" => true,
    ]
);
```

Validation

Phalcon forms are integrated with the *validation* component to offer instant validation. Built-in or custom validators could be set to each element:

```
<?php

use Phalcon\Forms\Element\Text;
use Phalcon\Validation\Validator\PresenceOf;
use Phalcon\Validation\Validator\StringLength;

$name = new Text(
    "name"
);

$name->addValidator(
    new PresenceOf(
        [
            "message" => "The name is required",
        ]
    )
);

$name->addValidator(
    new StringLength(
        [
            "min" => 10,
            "messageMinimum" => "The name is too short",
        ]
    )
);

$form->add($name);
```

Then you can validate the form according to the input entered by the user:

```
<?php

if (!$form->isValid($_POST)) {
    $messages = $form->getMessages();

    foreach ($messages as $message) {
        echo $message, "<br>";
    }
}
```

Validators are executed in the same order as they were registered.

By default messages generated by all the elements in the form are joined so they can be traversed using a single foreach, you can change this behavior to get the messages separated by the field:

```
<?php

foreach ($form->getMessages(false) as $attribute => $messages) {
    echo "Messages generated by ", $attribute, ":", "\n";

    foreach ($messages as $message) {
        echo $message, "<br>";
    }
}
```

Or get specific messages for an element:

```
<?php

$messages = $form->getMessagesFor("name");

foreach ($messages as $message) {
    echo $message, "<br>";
}
```

Filtering

A form is also able to filter data before it is validated. You can set filters in each element:

```
<?php

use Phalcon\Forms\Element\Text;

$name = new Text(
    "name"
);

// Set multiple filters
$name->setFilters(
    [
        "string",
        "trim",
    ]
);

$form->add($name);

$email = new Text(
    "email"
);

// Set one filter
$email->setFilters(
    "email"
);
```

```
$form->add($email);
```

Learn more about filtering in Phalcon by reading the *Filter documentation*.

Forms + Entities

An entity such as a model/collection/plain instance or just a plain PHP class can be linked to the form in order to set default values in the form's elements or assign the values from the form to the entity easily:

```
<?php

$robot = Robots::findFirst();

$form = new Form($robot);

$form->add(
    new Text(
        "name"
    )
);

$form->add(
    new Text(
        "year"
    )
);
```

Once the form is rendered if there is no default values assigned to the elements it will use the ones provided by the entity:

```
<?php echo $form->render("name"); ?>
```

You can validate the form and assign the values from the user input in the following way:

```
<?php

$form->bind($_POST, $robot);

// Check if the form is valid
if ($form->isValid()) {
    // Save the entity
    $robot->save();
}
```

Setting up a plain class as entity also is possible:

```
<?php

class Preferences
{
    public $timezone = "Europe/Amsterdam";

    public $receiveEmails = "No";
}
```

Using this class as entity, allows the form to take the default values from it:

```
<?php

$form = new Form(
    new Preferences()
);

$form->add(
    new Select(
        "timezone",
        [
            "America/New_York" => "New York",
            "Europe/Amsterdam" => "Amsterdam",
            "America/Sao_Paulo" => "Sao Paulo",
            "Asia/Tokyo"       => "Tokyo",
        ]
    )
);

$form->add(
    new Select(
        "receiveEmails",
        [
            "Yes" => "Yes, please!",
            "No"  => "No, thanks",
        ]
    )
);
```

Entities can implement getters, which have a higher precedence than public properties. These methods give you more freedom to produce values:

```
<?php

class Preferences
{
    public $timezone;

    public $receiveEmails;

    public function getTimezone()
    {
        return "Europe/Amsterdam";
    }

    public function getReceiveEmails()
    {
        return "No";
    }
}
```

Form Elements

Phalcon provides a set of built-in elements to use in your forms, all these elements are located in the *Phalcon\Forms\Element* namespace:

Name	Description
<i>Phalcon\Forms\Element\Text</i>	Generate INPUT[type=text] elements
<i>Phalcon\Forms\Element>Password</i>	Generate INPUT[type=password] elements
<i>Phalcon\Forms\Element\Select</i>	Generate SELECT tag (combo lists) elements based on choices
<i>Phalcon\Forms\Element\Check</i>	Generate INPUT[type=check] elements
<i>Phalcon\Forms\Element\TextArea</i>	Generate TEXTAREA elements
<i>Phalcon\Forms\Element\Hidden</i>	Generate INPUT[type=hidden] elements
<i>Phalcon\Forms\Element\File</i>	Generate INPUT[type=file] elements
<i>Phalcon\Forms\Element\Date</i>	Generate INPUT[type=date] elements
<i>Phalcon\Forms\Element\Numeric</i>	Generate INPUT[type=number] elements
<i>Phalcon\Forms\Element\Submit</i>	Generate INPUT[type=submit] elements

Event Callbacks

Whenever forms are implemented as classes, the callbacks: `beforeValidation()` and `afterValidation()` can be implemented in the form's class to perform pre-validations and post-validations:

```
<?php
use Phalcon\Forms\Form;

class ContactForm extends Form
{
    public function beforeValidation()
    {
    }
}
```

Rendering Forms

You can render the form with total flexibility, the following example shows how to render each element using a standard procedure:

```
<?php
<form method="post">
    <?php
        // Traverse the form
        foreach ($form as $element) {
            // Get any generated messages for the current element
            $messages = $form->getMessagesFor(
                $element->getName()
            );

            if (count($messages)) {
                // Print each element
                echo '<div class="messages">';
```

```
        foreach ($messages as $message) {
            echo $message;
        }

        echo "</div>";
    }

    echo "<p>";

    echo '<label for="' . $element->getName() . '">' . $element->getLabel() . "</
↪label>";

    echo $element;

    echo "</p>";
}

?>

<input type="submit" value="Send" />
</form>
```

Or reuse the logic in your form class:

```
<?php
use Phalcon\Forms\Form;

class ContactForm extends Form
{
    public function initialize()
    {
        // ...
    }

    public function renderDecorated($name)
    {
        $element = $this->get($name);

        // Get any generated messages for the current element
        $messages = $this->getMessagesFor(
            $element->getName()
        );

        if (count($messages)) {
            // Print each element
            echo '<div class="messages">';

            foreach ($messages as $message) {
                echo $this->flash->error($message);
            }

            echo "</div>";
        }

        echo "<p>";
    }
}
```



```

        echo '<label for="' . $element->getName() . '">' . $element->getLabel() . "</
↵label>";

        echo $element;

        echo "</p>";
    }
}

```

In the view:

```

<?php

echo $element->renderDecorated("name");

echo $element->renderDecorated("telephone");

```

Creating Form Elements

In addition to the form elements provided by Phalcon you can create your own custom elements:

```

<?php

use Phalcon\Forms\Element;

class MyElement extends Element
{
    public function render($attributes = null)
    {
        $html = // ... Produce some HTML

        return $html;
    }
}

```

Forms Manager

This component provides a forms manager that can be used by the developer to register forms and access them via the service locator:

```

<?php

use Phalcon\Forms\Manager as FormsManager;

$di["forms"] = function () {
    return new FormsManager();
};

```

Forms are added to the forms manager and referenced by a unique name:

```

<?php

$this->forms->set(
    "login",

```

```
new LoginForm()  
);
```

Using the unique name, forms can be accessed in any part of the application:

```
<?php  
  
$loginForm = $this->forms->get("login");  
  
echo $loginForm->render();
```

External Resources

- [Vökuró](#), is a sample application that uses the forms builder to create and manage forms, [\[Github\]](#)

2.3.35 Configuration de lecture

Phalcon\Config est un composant utilisé pour lire des fichiers de configuration sous différents formats (en utilisant des adaptateurs) et transformer le contenu en objet PHP pour pouvoir l'utiliser dans une application.

Les Array natifs

L'exemple suivant montre comment convertir les arrays natifs en objets *Phalcon\Config*. Cette option offre les meilleures performances vu qu'il n'y a pas de fichiers lu pendant la requête.

```
<?php  
  
use Phalcon\Config;  
  
$settings = [  
    "database" => [  
        "adapter" => "Mysql",  
        "host" => "localhost",  
        "username" => "scott",  
        "password" => "cheetah",  
        "dbname" => "test_db"  
    ],  
    "app" => [  
        "controllersDir" => "../app/controllers/",  
        "modelsDir" => "../app/models/",  
        "viewsDir" => "../app/views/"  
    ],  
    "mysetting" => "the-value"  
];  
  
$config = new Config($settings);  
  
echo $config->app->controllersDir, "\n";  
echo $config->database->username, "\n";  
echo $config->mysetting, "\n";
```

Si vous voulez mieux organiser votre projet vous pouvez sauvegarder l'array dans un autre fichier pour ensuite le lire.

```
<?php

use Phalcon\Config;

require "config/config.php";

$config = new Config($settings);
```

Adaptateur de fichier

Les adaptateurs disponibles sont :

File Type	Description
<i>Phalcon\Config\Adapter\Ini</i>	Utilise des fichiers INI pour stocker des paramètres. L'adaptateur utilise la fonction PHP <code>parse_ini_files</code>
<i>Phalcon\Config\Adapter\Json</i>	Uses JSON files to store settings.
<i>Phalcon\Config\Adapter\Php</i>	Uses PHP multidimensional arrays to store settings. This adapter offers the best performance.
<i>Phalcon\Config\Adapter\Yaml</i>	Uses YAML files to store settings.

Lire les fichiers INI

Les fichiers INI sont un moyen habituel pour stocker des paramètres. *Phalcon\Config* utilise la fonction `parse_ini_file` optimisé pour lire ces fichiers. Les sections sont parsées en sous-paramètres pour un accès simplifié.

```
[database]
adapter  = Mysql
host     = localhost
username = scott
password = cheetah
dbname   = test_db

[phalcon]
controllersDir = "../app/controllers/"
modelsDir      = "../app/models/"
viewsDir       = "../app/views/"

[models]
metadata.adapter = "Memory"
```

Vous pouvez lire le fichier comme cela :

```
<?php

use Phalcon\Config\Adapter\Ini as ConfigIni;

$config = new ConfigIni("path/config.ini");

echo $config->phalcon->controllersDir, "\n";
echo $config->database->username, "\n";
echo $config->models->metadata->adapter, "\n";
```

Configuration de fusion

Phalcon\Config permet de fusionner une configuration objet en un autre de manière récursif :

```
<?php

use Phalcon\Config;

$config = new Config(
    [
        "database" => [
            "host"    => "localhost",
            "dbname"  => "test_db",
        ],
        "debug" => 1,
    ]
);

$config2 = new Config(
    [
        "database" => [
            "dbname"  => "production_db",
            "username" => "scott",
            "password" => "secret",
        ],
        "logging" => 1,
    ]
);

$config->merge($config2);

print_r($config);
```

Le code fournit le résultat suivant :

```
Phalcon\Config Object
(
    [database] => Phalcon\Config Object
        (
            [host] => localhost
            [dbname] => production_db
            [username] => scott
            [password] => secret
        )
    [debug] => 1
    [logging] => 1
)
```

Il y a plus d'adaptateurs disponible pour ce composant dans l' [Incubateur Phalcon](#) There are more adapters available for this components in the [Incubateur Phalcon](#)

Injecting Configuration Dependency

You can inject configuration dependency to controller allowing us to use *Phalcon\Config* inside *Phalcon\Mvc\Controller*. To be able to do that, add following code inside your dependency injector script.

```
<?php

use Phalcon\Di\FactoryDefault;
use Phalcon\Config;

// Create a DI
$di = new FactoryDefault();

$di->set(
    "config",
    function () {
        $configData = require "config/config.php";

        return new Config($configData);
    }
);
```

Now in your controller you can access your configuration by using dependency injection feature using name *config* like following code:

```
<?php

use Phalcon\Mvc\Controller;

class MyController extends Controller
{
    private function getDatabaseName()
    {
        return $this->config->database->dbname;
    }
}
```

2.3.36 Pagination

The process of pagination takes place when we need to present big groups of arbitrary data gradually. `Phalcon\Paginator` offers a fast and convenient way to split these sets of data into browsable pages.

Data Adapters

This component makes use of adapters to encapsulate different sources of data:

Adapter	Description
<code>Phalcon\Paginator\Adapter\NativeArray</code>	Use a PHP array as source data
<code>Phalcon\Paginator\Adapter\ScrollableCursor</code>	Use a <code>Phalcon\Mvc\Model\Resultset</code> object as source data. Since PDO doesn't support scrollable cursors this adapter shouldn't be used to paginate a large number of records
<code>Phalcon\Paginator\Adapter\QueryBuilder</code>	Use a <code>Phalcon\Mvc\Model\Query\Builder</code> object as source data

Examples

In the example below, the paginator will use the result of a query from a model as its source data, and limit the displayed data to 10 records per page:

```
<?php

use Phalcon\Paginator\Adapter\Model as PaginatorModel;

// Current page to show
// In a controller/component this can be:
// $this->request->getQuery("page", "int"); // GET
// $this->request->getPost("page", "int"); // POST
$currentPage = (int) $_GET["page"];

// The data set to paginate
$robots = Robots::find();

// Create a Model paginator, show 10 rows by page starting from $currentPage
$paginator = new PaginatorModel(
    [
        "data" => $robots,
        "limit" => 10,
        "page" => $currentPage,
    ]
);

// Get the paginated results
$page = $paginator->getPaginate();
```

The `$currentPage` variable controls the page to be displayed. The `$paginator->getPaginate()` returns a `$page` object that contains the paginated data. It can be used for generating the pagination:

```
<table>
    <tr>
        <th>Id</th>
        <th>Name</th>
        <th>Type</th>
    </tr>
    <?php foreach ($page->items as $item) { ?>
    <tr>
        <td><?php echo $item->id; ?></td>
        <td><?php echo $item->name; ?></td>
        <td><?php echo $item->type; ?></td>
    </tr>
    <?php } ?>
</table>
```

The `$page` object also contains navigation data:

```
<a href="/robots/search">First</a>
<a href="/robots/search?page=<?= $page->before; ?>">Previous</a>
<a href="/robots/search?page=<?= $page->next; ?>">Next</a>
<a href="/robots/search?page=<?= $page->last; ?>">Last</a>

<?php echo "You are in page ", $page->current, " of ", $page->total_pages; ?>
```

Adapters Usage

An example of the source data that must be used for each adapter:

```

<?php

use Phalcon\Paginator\Adapter\Model as PaginatorModel;
use Phalcon\Paginator\Adapter\NativeArray as PaginatorArray;
use Phalcon\Paginator\Adapter\QueryBuilder as PaginatorQueryBuilder;

// Passing a resultset as data
$paginator = new PaginatorModel(
    [
        "data" => Products::find(),
        "limit" => 10,
        "page" => $currentPage,
    ]
);

// Passing an array as data
$paginator = new PaginatorArray(
    [
        "data" => [
            ["id" => 1, "name" => "Artichoke"],
            ["id" => 2, "name" => "Carrots"],
            ["id" => 3, "name" => "Beet"],
            ["id" => 4, "name" => "Lettuce"],
            ["id" => 5, "name" => ""],
        ],
        "limit" => 2,
        "page" => $currentPage,
    ]
);

// Passing a QueryBuilder as data

$builder = $this->modelsManager->createBuilder()
    ->columns("id, name")
    ->from("Robots")
    ->orderBy("name");

$paginator = new PaginatorQueryBuilder(
    [
        "builder" => $builder,
        "limit" => 20,
        "page" => 1,
    ]
);

```

Page Attributes

The `$page` object has the following attributes:

Attribute	Description
items	The set of records to be displayed at the current page
current	The current page
before	The previous page to the current one
next	The next page to the current one
last	The last page in the set of records
total_pages	The number of pages
total_items	The number of items in the source data

Implementing your own adapters

The *Phalcon\Paginator\AdapterInterface* interface must be implemented in order to create your own paginator adapters or extend the existing ones:

```
<?php

use Phalcon\Paginator\AdapterInterface as PaginatorInterface;

class MyPaginator implements PaginatorInterface
{
    /**
     * Adapter constructor
     *
     * @param array $config
     */
    public function __construct($config);

    /**
     * Set the current page number
     *
     * @param int $page
     */
    public function setCurrentPage($page);

    /**
     * Returns a slice of the resultset to show in the pagination
     *
     * @return stdClass
     */
    public function getPaginate();
}
```

2.3.37 Improving Performance with Cache

Phalcon provides the *Phalcon\Cache* class allowing faster access to frequently used or already processed data. *Phalcon\Cache* is written in C, achieving higher performance and reducing the overhead when getting items from the backends. This class uses an internal structure of frontend and backend components. Front-end components act as input sources or interfaces, while backend components offer storage options to the class.

When to implement cache?

Although this component is very fast, implementing it in cases that are not needed could lead to a loss of performance rather than gain. We recommend you check this cases before using a cache:

- You are making complex calculations that every time return the same result (changing infrequently)
- You are using a lot of helpers and the output generated is almost always the same
- You are accessing database data constantly and these data rarely change

NOTE Even after implementing the cache, you should check the hit ratio of your cache over a period of time. This can easily be done, especially in the case of Memcache or Apc, with the relevant tools that the backends provide.

Caching Behavior

The caching process is divided into 2 parts:

- **Frontend:** This part is responsible for checking if a key has expired and perform additional transformations to the data before storing and after retrieving them from the backend-
- **Backend:** This part is responsible for communicating, writing/reading the data required by the frontend.

Caching Output Fragments

An output fragment is a piece of HTML or text that is cached as is and returned as is. The output is automatically captured from the `ob_*` functions or the PHP output so that it can be saved in the cache. The following example demonstrates such usage. It receives the output generated by PHP and stores it into a file. The contents of the file are refreshed every 172800 seconds (2 days).

The implementation of this caching mechanism allows us to gain performance by not executing the helper `Phalcon\Tag::linkTo()` call whenever this piece of code is called.

```
<?php

use Phalcon\Tag;
use Phalcon\Cache\Backend\File as BackFile;
use Phalcon\Cache\Frontend\Output as FrontOutput;

// Create an Output frontend. Cache the files for 2 days
$frontCache = new FrontOutput(
    [
        "lifetime" => 172800,
    ]
);

// Create the component that will cache from the "Output" to a "File" backend
// Set the cache file directory - it's important to keep the "/" at the end of
// the value for the folder
$cache = new BackFile(
    $frontCache,
    [
        "cacheDir" => "../app/cache/",
    ]
);

// Get/Set the cache file to ../app/cache/my-cache.html
$content = $cache->start("my-cache.html");

// If $content is null then the content will be generated for the cache
if ($content === null) {
    // Print date and time
```

```
echo date("r");

// Generate a link to the sign-up action
echo Tag::linkTo(
    [
        "user/signup",
        "Sign Up",
        "class" => "signup-button",
    ]
);

// Store the output into the cache file
$cache->save();
} else {
    // Echo the cached output
    echo $content;
}
```

NOTE In the example above, our code remains the same, echoing output to the user as it has been doing before. Our cache component transparently captures that output and stores it in the cache file (when the cache is generated) or it sends it back to the user pre-compiled from a previous call, thus avoiding expensive operations.

Caching Arbitrary Data

Caching just data is equally important for your application. Caching can reduce database load by reusing commonly used (but not updated) data, thus speeding up your application.

File Backend Example

One of the caching adapters is 'File'. The only key area for this adapter is the location of where the cache files will be stored. This is controlled by the `cacheDir` option which *must* have a backslash at the end of it.

```
<?php

use Phalcon\Cache\Backend\File as BackFile;
use Phalcon\Cache\Frontend\Data as FrontData;

// Cache the files for 2 days using a Data frontend
$frontCache = new FrontData(
    [
        "lifetime" => 172800,
    ]
);

// Create the component that will cache "Data" to a "File" backend
// Set the cache file directory - important to keep the "/" at the end of
// the value for the folder
$cache = new BackFile(
    $frontCache,
    [
        "cacheDir" => "../app/cache/",
    ]
);

$cacheKey = "robots_order_id.cache";
```

```
// Try to get cached records
$robots = $cache->get($cacheKey);

if ($robots === null) {
    // $robots is null because of cache expiration or data does not exist
    // Make the database call and populate the variable
    $robots = Robots::find(
        [
            "order" => "id",
        ]
    );

    // Store it in the cache
    $cache->save($cacheKey, $robots);
}

// Use $robots :)
foreach ($robots as $robot) {
    echo $robot->name, "\n";
}
```

Memcached Backend Example

The above example changes slightly (especially in terms of configuration) when we are using a Memcached backend.

```
<?php

use Phalcon\Cache\Frontend\Data as FrontData;
use Phalcon\Cache\Backend\Libmemcached as BackMemCached;

// Cache data for one hour
$frontCache = new FrontData(
    [
        "lifetime" => 3600,
    ]
);

// Create the component that will cache "Data" to a "Memcached" backend
// Memcached connection settings
$cache = new BackMemCached(
    $frontCache,
    [
        "servers" => [
            [
                "host"    => "127.0.0.1",
                "port"    => "11211",
                "weight" => "1",
            ]
        ]
    ]
);

$cacheKey = "robots_order_id.cache";

// Try to get cached records
```

```
$robots = $cache->get($cacheKey);

if ($robots === null) {
    // $robots is null because of cache expiration or data does not exist
    // Make the database call and populate the variable
    $robots = Robots::find(
        [
            "order" => "id",
        ]
    );

    // Store it in the cache
    $cache->save($cacheKey, $robots);
}

// Use $robots :)
foreach ($robots as $robot) {
    echo $robot->name, "\n";
}
```

Querying the cache

The elements added to the cache are uniquely identified by a key. In the case of the File backend, the key is the actual filename. To retrieve data from the cache, we just have to call it using the unique key. If the key does not exist, the get method will return null.

```
<?php

// Retrieve products by key "myProducts"
$products = $cache->get("myProducts");
```

If you want to know which keys are stored in the cache you could call the queryKeys method:

```
<?php

// Query all keys used in the cache
$keys = $cache->queryKeys();

foreach ($keys as $key) {
    $data = $cache->get($key);

    echo "Key=", $key, " Data=", $data;
}

// Query keys in the cache that begins with "my-prefix"
$keys = $cache->queryKeys("my-prefix");
```

Deleting data from the cache

There are times where you will need to forcibly invalidate a cache entry (due to an update in the cached data). The only requirement is to know the key that the data have been stored with.

```
<?php

// Delete an item with a specific key
```

```

$cache->delete("someKey");

$keys = $cache->queryKeys();

// Delete all items from the cache
foreach ($keys as $key) {
    $cache->delete($key);
}

```

Checking cache existence

It is possible to check if a cache already exists with a given key:

```

<?php

if ($cache->exists("someKey")) {
    echo $cache->get("someKey");
} else {
    echo "Cache does not exists!";
}

```

Lifetime

A “lifetime” is a time in seconds that a cache could live without expire. By default, all the created caches use the lifetime set in the frontend creation. You can set a specific lifetime in the creation or retrieving of the data from the cache:

Setting the lifetime when retrieving:

```

<?php

$cacheKey = "my.cache";

// Setting the cache when getting a result
$robots = $cache->get($cacheKey, 3600);

if ($robots === null) {
    $robots = "some robots";

    // Store it in the cache
    $cache->save($cacheKey, $robots);
}

```

Setting the lifetime when saving:

```

<?php

$cacheKey = "my.cache";

$robots = $cache->get($cacheKey);

if ($robots === null) {
    $robots = "some robots";

    // Setting the cache when saving data
}

```

```
$cache->save($cacheKey, $robots, 3600);  
}
```

Multi-Level Cache

This feature of the cache component, allows the developer to implement a multi-level cache. This new feature is very useful because you can save the same data in several cache locations with different lifetimes, reading first from the one with the faster adapter and ending with the slowest one until the data expires:

```
<?php  
  
use Phalcon\Cache\Multiple;  
use Phalcon\Cache\Backend\Apc as ApcCache;  
use Phalcon\Cache\Backend\File as FileCache;  
use Phalcon\Cache\Frontend\Data as DataFrontend;  
use Phalcon\Cache\Backend\Memcache as MemcacheCache;  
  
$ultraFastFrontend = new DataFrontend(  
    [  
        "lifetime" => 3600,  
    ]  
);  
  
$fastFrontend = new DataFrontend(  
    [  
        "lifetime" => 86400,  
    ]  
);  
  
$slowFrontend = new DataFrontend(  
    [  
        "lifetime" => 604800,  
    ]  
);  
  
// Backends are registered from the fastest to the slower  
$cache = new Multiple(  
    [  
        new ApcCache(  
            $ultraFastFrontend,  
            [  
                "prefix" => "cache",  
            ]  
        ),  
        new MemcacheCache(  
            $fastFrontend,  
            [  
                "prefix" => "cache",  
                "host"    => "localhost",  
                "port"    => "11211",  
            ]  
        ),  
        new FileCache(  
            $slowFrontend,  
            [  
                "prefix" => "cache",
```

```

        "cacheDir" => "../app/cache/",
    ],
),
];

// Save, saves in every backend
$cache->save("my-key", $data);

```

Frontend Adapters

The available frontend adapters that are used as interfaces or input sources to the cache are:

Adapter	Description
<i>Phalcon\Cache\Frontend\Output</i>	Read input data from standard PHP output
<i>Phalcon\Cache\Frontend\Data</i>	It's used to cache any kind of PHP data (big arrays, objects, text, etc). Data is serialized before stored in the backend.
<i>Phalcon\Cache\Frontend\Binary</i>	It's used to cache binary data. The data is serialized using base64_encode before be stored in the backend.
<i>Phalcon\Cache\Frontend\Json</i>	Data is encoded in JSON before be stored in the backend. Decoded after be retrieved. This frontend is useful to share data with other languages or frameworks.
<i>Phalcon\Cache\Frontend\Igbinary</i>	It's used to cache any kind of PHP data (big arrays, objects, text, etc). Data is serialized using igbinary before be stored in the backend.
<i>Phalcon\Cache\Frontend\None</i>	It's used to cache any kind of PHP data without serializing them.

Implementing your own Frontend adapters

The *Phalcon\Cache\FrontendInterface* interface must be implemented in order to create your own frontend adapters or extend the existing ones.

Backend Adapters

The backend adapters available to store cache data are:

Adapter	Description	Info	Required Extensions
<i>Phalcon\Cache\Backend\File</i>	Stores data to local plain files		
<i>Phalcon\Cache\Backend\Memcache</i>	Stores data to a memcached server	Mem-cached	memcache
<i>Phalcon\Cache\Backend\Apc</i>	Stores data to the Alternative PHP Cache (APC)	APC	APC extension
<i>Phalcon\Cache\Backend\Mongo</i>	Stores data to Mongo Database	MongoDb	Mongo
<i>Phalcon\Cache\Backend\Xcache</i>	Stores data in XCache	XCache	xcache extension
<i>Phalcon\Cache\Backend\Redis</i>	Stores data in Redis	Redis	redis extension

Implementing your own Backend adapters

The *Phalcon\Cache\BackendInterface* interface must be implemented in order to create your own backend adapters or extend the existing ones.

File Backend Options

This backend will store cached content into files in the local server. The available options for this backend are:

Option	Description
prefix	A prefix that is automatically prepended to the cache keys
cacheDir	A writable directory on which cached files will be placed

Memcached Backend Options

This backend will store cached content on a memcached server. The available options for this backend are:

Option	Description
prefix	A prefix that is automatically prepended to the cache keys
host	memcached host
port	memcached port
persistent	create a persistent connection to memcached?

APC Backend Options

This backend will store cached content on Alternative PHP Cache (**APC**). The available options for this backend are:

Option	Description
prefix	A prefix that is automatically prepended to the cache keys

Mongo Backend Options

This backend will store cached content on a MongoDB server. The available options for this backend are:

Option	Description
prefix	A prefix that is automatically prepended to the cache keys
server	A MongoDB connection string
db	Mongo database name
collection	Mongo collection in the database

XCache Backend Options

This backend will store cached content on XCache (**XCache**). The available options for this backend are:

Option	Description
prefix	A prefix that is automatically prepended to the cache keys

Redis Backend Options

This backend will store cached content on a Redis server ([Redis](#)). The available options for this backend are:

Option	Description
prefix	A prefix that is automatically prepended to the cache keys
host	Redis host
port	Redis port
auth	Password to authenticate to a password-protected Redis server
persistent	Create a persistent connection to Redis
index	The index of the Redis database to use

There are more adapters available for this components in the [Phalcon Incubator](#)

2.3.38 Security

This component aids the developer in common security tasks such as password hashing and Cross-Site Request Forgery protection (CSRF).

Password Hashing

Storing passwords in plain text is a bad security practice. Anyone with access to the database will immediately have access to all user accounts thus being able to engage in unauthorized activities. To combat that, many applications use the familiar one way hashing methods “md5” and “sha1”. However, hardware evolves each day, and becomes faster, these algorithms are becoming vulnerable to brute force attacks. These attacks are also known as [rainbow tables](#).

To solve this problem we can use hash algorithms as [bcrypt](#). Why bcrypt? Thanks to its “Eksblowfish” key setup algorithm we can make the password encryption as “slow” as we want. Slow algorithms make the process to calculate the real password behind a hash extremely difficult if not impossible. This will protect your for a long time from a possible attack using rainbow tables.

This component gives you the ability to use this algorithm in a simple way:

```
<?php

use Phalcon\Mvc\Controller;

class UsersController extends Controller
{
    public function registerAction()
    {
        $user = new Users();

        $login    = $this->request->getPost("login");
        $password = $this->request->getPost("password");

        $user->login = $login;

        // Store the password hashed
        $user->password = $this->security->hash($password);

        $user->save();
    }
}
```

We saved the password hashed with a default work factor. A higher work factor will make the password less vulnerable as its encryption will be slow. We can check if the password is correct as follows:

```
<?php

use Phalcon\Mvc\Controller;

class SessionController extends Controller
{
    public function loginAction()
    {
        $login    = $this->request->getPost("login");
        $password = $this->request->getPost("password");

        $user = Users::findFirstByLogin($login);
        if ($user) {
            if ($this->security->checkHash($password, $user->password)) {
                // The password is valid
            }
            else {
                // To protect against timing attacks. Regardless of whether a user exists,
                ↪ or not, the script will take roughly the same amount as it will always be computing
                ↪ a hash.
                $this->security->hash(rand());
            }

            // The validation has failed
        }
    }
}
```

The salt is generated using pseudo-random bytes with the PHP's function `openssl_random_pseudo_bytes` so is required to have the `openssl` extension loaded.

Cross-Site Request Forgery (CSRF) protection

This is another common attack against web sites and applications. Forms designed to perform tasks such as user registration or adding comments are vulnerable to this attack.

The idea is to prevent the form values from being sent outside our application. To fix this, we generate a **random nonce** (token) in each form, add the token in the session and then validate the token once the form posts data back to our application by comparing the stored token in the session to the one submitted by the form:

```
<?php echo Tag::form('session/login') ?>

    <!-- Login and password inputs ... -->

    <input type="hidden" name="<?php echo $this->security->getTokenKey() ?>"
        value="<?php echo $this->security->getToken() ?>"/>

</form>
```

Then in the controller's action you can check if the CSRF token is valid:

```
<?php

use Phalcon\Mvc\Controller;
```

```

class SessionController extends Controller
{
    public function loginAction()
    {
        if ($this->request->isPost()) {
            if ($this->security->checkToken()) {
                // The token is OK
            }
        }
    }
}

```

Remember to add a session adapter to your Dependency Injector, otherwise the token check won't work:

```

<?php
$di->setShared(
    "session",
    function () {
        $session = new \Phalcon\Session\Adapter\Files();

        $session->start();

        return $session;
    }
);

```

Adding a [captcha](#) to the form is also recommended to completely avoid the risks of this attack.

Setting up the component

This component is automatically registered in the services container as 'security', you can re-register it to setup its options:

```

<?php
use Phalcon\Security;

$di->set(
    "security",
    function () {
        $security = new Security();

        // Set the password hashing factor to 12 rounds
        $security->setWorkFactor(12);

        return $security;
    },
    true
);

```

Random

The *Phalcon\Security\Random* class makes it really easy to generate lots of types of random data.

```
<?php

use Phalcon\Security\Random;

$random = new Random();

// ...
$bytes    = $random->bytes();

// Generate a random hex string of length $len.
$hex      = $random->hex($len);

// Generate a random base64 string of length $len.
$base64   = $random->base64($len);

// Generate a random URL-safe base64 string of length $len.
$base64Safe = $random->base64Safe($len);

// Generate a UUID (version 4). See https://en.wikipedia.org/wiki/Universally_unique_
↳ identifier
$uuid     = $random->uuid();

// Generate a random integer between 0 and $n.
$number   = $random->number($n);
```

External Resources

- [Vökuró](#), is a sample application that uses the Security component for avoid CSRF and password hashing, [\[Github\]](#)

2.3.39 Encryption/Decryption

Phalcon provides encryption facilities via the *Phalcon\Crypt* component. This class offers simple object-oriented wrappers to the [openssl](#) PHP's encryption library.

By default, this component provides secure encryption using AES-256-CFB.

You must use a key length corresponding to the current algorithm. For the algorithm used by default it is 32 bytes.

Basic Usage

This component is designed to provide a very simple usage:

```
<?php

use Phalcon\Crypt;

// Create an instance
$crypt = new Crypt();

$key   = "This is a secret key (32 bytes).";
$text  = "This is the text that you want to encrypt.";
```

```
$encrypted = $crypt->encrypt($text, $key);

echo $crypt->decrypt($encrypted, $key);
```

You can use the same instance to encrypt/decrypt several times:

```
<?php

use Phalcon\Crypt;

// Create an instance
$crypt = new Crypt();

$texts = [
    "my-key"    => "This is a secret text",
    "other-key" => "This is a very secret",
];

foreach ($texts as $key => $text) {
    // Perform the encryption
    $encrypted = $crypt->encrypt($text, $key);

    // Now decrypt
    echo $crypt->decrypt($encrypted, $key);
}
```

Encryption Options

The following options are available to change the encryption behavior:

Name	Description
Cipher	The cipher is one of the encryption algorithms supported by openssl. You can see a list here

Example:

```
<?php

use Phalcon\Crypt;

// Create an instance
$crypt = new Crypt();

// Use blowfish
$crypt->setCipher("bf-cbc");

$key = "le password";
$text = "This is a secret text";

echo $crypt->encrypt($text, $key);
```

Base64 Support

In order for encryption to be properly transmitted (emails) or displayed (browsers) [base64](#) encoding is usually applied to encrypted texts:

```
<?php

use Phalcon\Crypt;

// Create an instance
$crypt = new Crypt();

$key = "le password";
$text = "This is a secret text";

$encrypt = $crypt->encryptBase64($text, $key);

echo $crypt->decryptBase64($encrypt, $key);
```

Setting up an Encryption service

You can set up the encryption component in the services container in order to use it from any part of the application:

```
<?php

use Phalcon\Crypt;

$di->set(
    "crypt",
    function () {
        $crypt = new Crypt();

        // Set a global encryption key
        $crypt->setKey(
            "%31.1e$i86e$f!8jz"
        );

        return $crypt;
    },
    true
);
```

Then, for example, in a controller you can use it as follows:

```
<?php

use Phalcon\Mvc\Controller;

class SecretsController extends Controller
{
    public function saveAction()
    {
        $secret = new Secrets();

        $text = $this->request->getPost("text");

        $secret->content = $this->crypt->encrypt($text);

        if ($secret->save()) {
            $this->flash->success(
                "Secret was successfully created!"
            );
        }
    }
}
```

```

    );
}
}
}

```

2.3.40 Access Control Lists - Listes de Contrôle d'Access (ACL)

Phalcon\Acl fournit une gestion facile et légère des ACLs ainsi que les permissions qui lui sont rattachés. Les *Access Control Lists* (ACL) permettent à une application de contrôler l'accès à ses différentes zones ainsi que les objets concernés par les requêtes. Vous êtes encouragés de vous documenter sur la méthodologie ACL afin d'être familiarisé avec ces concepts.

En résumé, les ACLs possèdent des rôles et des ressources. Les ressources sont des objets qui doivent se conformer aux permissions qui leur sont attribuées par les ACLs. Les rôles sont des objets dont les requêtes d'accès aux ressources sont autorisées ou refusées par le mécanisme des ACLs.

Création d'une ACL

Ce composant est destiné initialement à fonctionner en mémoire. Ceci fournit une utilisation simple et rapide pour accéder aux différents aspects de la liste.

Le constructeur de *Phalcon\Acl* prend en premier paramètre un adaptateur pour récupérer l'information relative à la liste de contrôle. Un exemple utilisant l'adaptateur en mémoire se trouve ci-dessous:

```

<?php

use Phalcon\Acl\Adapter\Memory as AclList;

$acl = new AclList();

```

Par défaut *Phalcon\Acl* autorise les actions sur les ressources qui ne sont pas encore définies. Pour augmenter le niveau de sécurité de la liste d'accès nous pouvons définir un niveau "deny" comme niveau d'accès par défaut.

```

<?php

// Par défaut l'action est interdite
$acl->setDefaultAction(Phalcon\Acl::DENY);

```

Ajout de Rôles à l'ACL

Un rôle est un objet qui peut ou ne peut pas accéder à certaines ressource dans la liste d'accès. Par exemple nous associerons des rôles à des groupes de personnes dans une organisation. La classe *Phalcon\Acl\Role* peut servir à créer des rôles d'une façon structurée. Ajoutons maintenant quelques rôles à notre liste récemment créée:

```

<?php

use Phalcon\Acl\Role;

// Création de quelques rôles
// Le premier paramètre est le nom, le deuxième une description optionnelle.
$roleAdmins = new Role("Administrators", "Rôle super utilisateur");
$roleGuests = new Role("Guests");

```

```
// Ajoute le rôle "Guests" à l'ACL
$acl->addRole($roleGuests);

// Ajout le rôle "Designers" à l'ACL sans Phalcon\Acl\Role
$acl->addRole("Designers");
```

Comme vous pouvez le voir, les rôles sont directement définis sans utiliser une instance.

Ajout de ressources

Les ressources sont des objets dont l'accès est contrôlé. Dans les applications MVC il s'agit normalement des contrôleurs. Comme ce n'est pas obligatoire la classe *Phalcon\Acl\Resource* peut être utilisée pour définir des ressources. Il est important d'ajouter les actions ou les opérations associées à une ressource afin que l'ACL sache ce qu'il contrôle.

```
<?php

use Phalcon\Acl\Resource;

// Definition de la ressource "Customers"
$customersResource = new Resource("Customers");

// Ajoute la ressource "customers" a un couple d'opérations

$acl->addResource(
    $customersResource,
    "search"
);

$acl->addResource(
    $customersResource,
    [
        "create",
        "update",
    ]
);
```

Définition des Contrôles d'Accès

Maintenant que nous avons des rôles et des ressources, il est temps de définir notre ACL (par ex. quels rôles peut accéder à quelles ressources). Cette partie est très importante surtout qu'il faut prendre en compte l'accès par défaut qui est "allow" ou "deny".

```
<?php

// Définition des niveaux d'accès aux ressources

$acl->allow("Guests", "Customers", "search");

$acl->allow("Guests", "Customers", "create");

$acl->deny("Guests", "Customers", "update");
```

La méthode `allow()` indique qu'un rôle particulier peut accéder à une ressource en particulier. La méthode `deny()` fait l'opposé.

Interrogation de l'ACL

Une fois que la liste est complète, nous pouvons l'interroger pour vérifier si un rôle dispose d'une autorisation ou non.

```
<?php

// Vérifier quel rôle accède aux opérations

// Returns 0
$acl->isAllowed("Guests", "Customers", "edit");

// Returns 1
$acl->isAllowed("Guests", "Customers", "search");

// Returns 1
$acl->isAllowed("Guests", "Customers", "create");
```

Accès par fonction

Vous pouvez aussi ajouter en 4ème paramètre votre fonction personnalisée qui doit retourner un booléen. Celle-ci est invoquée dès que la méthode `isAllowed()` est appelée. Vous pouvez passer les paramètres en tant que tableau associatif à la méthode `isAllowed()` en tant que 4ème argument où la clé est le nom du paramètre dans votre fonction.

```
<?php
// Set access level for role into resources with custom function
$acl->allow(
    "Guests",
    "Customers",
    "search",
    function ($a) {
        return $a % 2 === 0;
    }
);

// Check whether role has access to the operation with custom function

// Returns true
$acl->isAllowed(
    "Guests",
    "Customers",
    "search",
    [
        "a" => 4,
    ]
);

// Returns false
$acl->isAllowed(
    "Guests",
    "Customers",
    "search",
    [
        "a" => 3,
    ]
);
```

Si vous ne fournissez pas de paramètres à la méthode `isAllowed()`, le comportement par défaut est `Acl::ALLOW`. Vous pouvez changer cela en utilisant la méthode `setNoArgumentsDefaultAction()`.

```
use Phalcon\Acl;

<?php
// Set access level for role into resources with custom function
$acl->allow(
    "Guests",
    "Customers",
    "search",
    function ($a) {
        return $a % 2 === 0;
    }
);

// Check whether role has access to the operation with custom function

// Returns true
$acl->isAllowed(
    "Guests",
    "Customers",
    "search"
);

// Change no arguments default action
$acl->setNoArgumentsDefaultAction(
    Acl::DENY
);

// Returns false
$acl->isAllowed(
    "Guests",
    "Customers",
    "search"
);
```

Des objets en tant que nom de rôle et de ressource

Vous pouvez transmettre des objets à `roleName` et `resourceName`. Vos classes doivent implémenter *Phalcon\Acl\RoleAware* pour `roleName` et *Phalcon\Acl\ResourceAware* pour `resourceName`.

Notre classe `UserRole`

```
<?php

use Phalcon\Acl\RoleAware;

// Create our class which will be used as roleName
class UserRole implements RoleAware
{
    protected $id;

    protected $roleName;

    public function __construct($id, $roleName)
    {
```

```

        $this->id      = $id;
        $this->roleName = $roleName;
    }

    public function getId()
    {
        return $this->id;
    }

    // Implemented function from RoleAware Interface
    public function getRoleName()
    {
        return $this->roleName;
    }
}

```

Et notre classe ModelResource

```

<?php

use Phalcon\Acl\ResourceAware;

// Create our class which will be used as resourceName
class ModelResource implements ResourceAware
{
    protected $id;

    protected $resourceName;

    protected $userId;

    public function __construct($id, $resourceName, $userId)
    {
        $this->id      = $id;
        $this->resourceName = $resourceName;
        $this->userId   = $userId;
    }

    public function getId()
    {
        return $this->id;
    }

    public function getUserId()
    {
        return $this->userId;
    }

    // Implemented function from ResourceAware Interface
    public function getResourceName()
    {
        return $this->resourceName;
    }
}

```

Ainsi vous pouvez les utiliser dans la méthode `isAllowed()`.

```
<?php

use UserRole;
use ModelResource;

// Set access level for role into resources
$acl->allow("Guests", "Customers", "search");
$acl->allow("Guests", "Customers", "create");
$acl->deny("Guests", "Customers", "update");

// Create our objects providing roleName and resourceName

$customer = new ModelResource(
    1,
    "Customers",
    2
);

$designer = new UserRole(
    1,
    "Designers"
);

$guest = new UserRole(
    2,
    "Guests"
);

$anotherGuest = new UserRole(
    3,
    "Guests"
);

// Check whether our user objects have access to the operation on model object

// Returns false
$acl->isAllowed(
    $designer,
    $customer,
    "search"
);

// Returns true
$acl->isAllowed(
    $guest,
    $customer,
    "search"
);

// Returns true
$acl->isAllowed(
    $anotherGuest,
    $customer,
    "search"
);
```

Vous pouvez également accéder à ces objets dans votre fonction personnalisée dans `allow()` ou `deny()`. Les paramètres sont automatiquement liés selon les types dans la fonction.

```

<?php

use UserRole;
use ModelResource;

// Set access level for role into resources with custom function
$acl->allow(
    "Guests",
    "Customers",
    "search",
    function (UserRole $user, ModelResource $model) { // User and Model classes are_
↪necessary
        return $user->getId == $model->getUserId();
    }
);

$acl->allow(
    "Guests",
    "Customers",
    "create"
);

$acl->deny(
    "Guests",
    "Customers",
    "update"
);

// Create our objects providing roleName and resourceName

$customer = new ModelResource(
    1,
    "Customers",
    2
);

$designer = new UserRole(
    1,
    "Designers"
);

$guest = new UserRole(
    2,
    "Guests"
);

$anotherGuest = new UserRole(
    3,
    "Guests"
);

// Check whether our user objects have access to the operation on model object

// Returns false
$acl->isAllowed(
    $designer,
    $customer,
    "search"

```

```
);

// Returns true
$acl->isAllowed(
    $quest,
    $customer,
    "search"
);

// Returns false
$acl->isAllowed(
    $anotherGuest,
    $customer,
    "search"
);
```

Vous pouvez toujours ajouter des paramètres personnalisés à la fonction et passer un tableau associatif à la méthode `isAllowed()`. L'ordre n'a aucune importance.

Héritage de rôles

Vous pouvez construire des structures de rôles complexes en exploitant l'héritage que fournit *Phalcon\Acl\Role*. Les rôles peuvent hériter d'autre rôles, autorisant ainsi l'accès à des surensembles ou des sous-ensembles de ressources. Pour profiter de l'héritage, vous devrez passer le rôle parent en second paramètre lors de l'appel de la méthode d'ajout du rôle dans la liste.

```
<?php

use Phalcon\Acl\Role;

// ...

// Création de quelques rôles

$roleAdmins = new Role("Administrators", "Super-User role");

$roleGuests = new Role("Guests");

// Ajout du rôle "Guests" à l'ACL
$acl->addRole($roleGuests);

// Ajout du rôle "Administrators" héritant des accès de "Guests"
$acl->addRole($roleAdmins, $roleGuests);
```

Sérialisation des listes ACL

Pour améliorer les performances, les instance de *Phalcon\Acl* peuvent être sérialisées et stockées dans l'APC, en session, dans des fichiers textes ou une table de base de données, et peuvent être chargées à volonté sans avoir à redéfinir toute la liste. Vous pouvez faire ça comme suit:

```
<?php

use Phalcon\Acl\Adapter\Memory as AclList;

// ...
```

```
// Vérifier que les données de l'ACL existent
if (!is_file("app/security/acl.data")) {
    $acl = new AclList();

    // ... Définition des rôles, ressources, accès, etc.

    // Stockage de la liste sérialisée dans un fichier
    file_put_contents(
        "app/security/acl.data",
        serialize($acl)
    );
} else {
    // Récupération des objets ACL depuis le fichier sérialisé
    $acl = unserialize(
        file_get_contents("app/security/acl.data")
    );
}

// Utilisation de la liste selon les besoins
if ($acl->isAllowed("Guests", "Customers", "edit")) {
    echo "Accès autorisé!";
} else {
    echo "Accès refusé :(";
}
```

Nous vous recommandons d'utiliser l'adaptateur mémoire pendant le développement et d'utiliser l'un des autres adaptateurs en production.

Événements ACL

Phalcon\Acl est capable d'émettre des événements à *EventsManager* s'il existe. Les événements sont déclenchés en utilisant le type "acl". Certains événements peuvent interrompre l'opération active lorsqu'ils retournent faux. Les événements suivants sont supportés:

Nom d'évt	Déclenché par	Opération stoppée ?
beforeCheckAccess	Déclenché avant le contrôle d'accès d'un rôle ou ressource	Oui
afterCheckAccess	Déclenché après le contrôle d'accès d'un rôle ou ressource	Non

L'exemple suivant montre comment attacher un écouteur à ce composant:

```
<?php

use Phalcon\Acl\Adapter\Memory as AclList;
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;

// ...

// Création d'un gestionnaire d'événements
$eventsManager = new EventsManager();

// Attach a listener for type "acl"
$eventsManager->attach(
    "acl:beforeCheckAccess",
    function (Event $event, $acl) {
        echo $acl->getActiveRole();
    }
);
```

```
        echo $acl->getActiveResource();

        echo $acl->getActiveAccess();
    }
);

$acl = new AclList();

// Setup the $acl
// ...

// Bind the eventsManager to the ACL component
$acl->setEventsManager($eventsManager);
```

Création de vos propre adaptateurs

Pour créer votre propre adaptateur vous devez implémenter l'interface *Phalcon\Acl\AdapterInterface* ou bien étendre un adaptateur existant.

2.3.41 Multi-lingual Support

The component *Phalcon\Translate* aids in creating multilingual applications. Applications using this component, display content in different languages, based on the user's chosen language supported by the application.

Adapters

This component makes use of adapters to read translation messages from different sources in a unified way.

Adapter	Description
<i>Phalcon\Translate\Adapter\NativeArray</i>	Uses PHP arrays to store the messages. This is the best option in terms of performance.

Component Usage

Translation strings are stored in files. The structure of these files could vary depending of the adapter used. Phalcon gives you the freedom to organize your translation strings. A simple structure could be:

```
app/messages/en.php
app/messages/es.php
app/messages/fr.php
app/messages/zh.php
```

Each file contains an array of the translations in a key/value manner. For each translation file, keys are unique. The same array is used in different files, where keys remain the same and values contain the translated strings depending on each language.

```
<?php

// app/messages/en.php
$messages = [
    "hi"      => "Hello",
```



```

    "bye"      => "Good Bye",
    "hi-name" => "Hello %name%",
    "song"     => "This song is %song%",
];

```

```

<?php

// app/messages/fr.php
$messages = [
    "hi"      => "Bonjour",
    "bye"     => "Au revoir",
    "hi-name" => "Bonjour %name%",
    "song"    => "La chanson est %song%",
];

```

Implementing the translation mechanism in your application is trivial but depends on how you wish to implement it. You can use an automatic detection of the language from the user's browser or you can provide a settings page where the user can select their language.

A simple way of detecting the user's language is to parse the `$_SERVER['HTTP_ACCEPT_LANGUAGE']` contents, or if you wish, access it directly by calling `$this->request->getBestLanguage()` from an action/controller:

```

<?php

use Phalcon\Mvc\Controller;
use Phalcon\Translate\Adapter\NativeArray;

class UserController extends Controller
{
    protected function getTranslation()
    {
        // Ask browser what is the best language
        $language = $this->request->getBestLanguage();

        $translationFile = "app/messages/" . $language . ".php";

        // Check if we have a translation file for that lang
        if (file_exists($translationFile)) {
            require $translationFile;
        } else {
            // Fallback to some default
            require "app/messages/en.php";
        }

        // Return a translation object
        return new NativeArray(
            [
                "content" => $messages,
            ]
        );
    }

    public function indexAction()
    {
        $this->view->name = "Mike";
        $this->view->t     = $this->getTranslation();
    }
}

```

The `_getTranslation()` method is available for all actions that require translations. The `$t` variable is passed to the views, and with it, we can translate strings in that layer:

```
<!-- welcome -->
<!-- String: hi => 'Hello' -->
<p><?php echo $t->_("hi"), " ", $name; ?></p>
```

The `_()` method is returning the translated string based on the index passed. Some strings need to incorporate placeholders for calculated data i.e. Hello `%name%`. These placeholders can be replaced with passed parameters in the `_()` method. The passed parameters are in the form of a key/value array, where the key matches the placeholder name and the value is the actual data to be replaced:

```
<!-- welcome -->
<!-- String: hi-name => 'Hello %name%' -->
<p><?php echo $t->_("hi-name", ["name" => $name]); ?></p>
```

Some applications implement multilingual on the URL such as <http://www.mozilla.org/es-ES/firefox/>. Phalcon can implement this by using a *Router*.

Implementing your own adapters

The *Phalcon\Translate\AdapterInterface* interface must be implemented in order to create your own translate adapters or extend the existing ones:

```
<?php

use Phalcon\Translate\AdapterInterface;

class MyTranslateAdapter implements AdapterInterface
{
    /**
     * Adapter constructor
     *
     * @param array $data
     */
    public function __construct($options);

    /**
     * Returns the translation string of the given key
     *
     * @param string $translateKey
     * @param array $placeholders
     * @return string
     */
    public function _($translateKey, $placeholders = null);

    /**
     * Returns the translation related to the given key
     *
     * @param string $index
     * @param array $placeholders
     * @return string
     */
    public function query($index, $placeholders = null);
```

```

/**
 * Check whether is defined a translation key in the internal array
 *
 * @param    string $index
 * @return   bool
 */
public function exists($index);
}

```

There are more adapters available for this components in the [Phalcon Incubator](#)

2.3.42 Class Autoloader

Phalcon\Loader allows you to load project classes automatically, based on some predefined rules. Since this component is written in C, it provides the lowest overhead in reading and interpreting external PHP files.

The behavior of this component is based on the PHP's capability of [autoloading classes](#). If a class that does not yet exist is used in any part of the code, a special handler will try to load it. *Phalcon\Loader* serves as the special handler for this operation. By loading classes on a need-to-load basis, the overall performance is increased since the only file reads that occur are for the files needed. This technique is called [lazy initialization](#).

With this component you can load files from other projects or vendors, this autoloader is [PSR-0](#) and [PSR-4](#) compliant.

Phalcon\Loader offers four options to autoload classes. You can use them one at a time or combine them.

Security Layer

Phalcon\Loader offers a security layer sanitizing by default class names avoiding possible inclusion of unauthorized files. Consider the following example:

```

<?php

// Basic autoloader
spl_autoload_register(
    function ($className) {
        $filepath = $className . ".php";

        if (file_exists($filepath)) {
            require $filepath;
        }
    }
);

```

The above auto-loader lacks any kind of security. If a function mistakenly launches the auto-loader and a malicious prepared string is used as parameter this would allow to execute any file accessible by the application:

```

<?php

// This variable is not filtered and comes from an insecure source
$className = "../processes/important-process";

// Check if the class exists triggering the auto-loader
if (class_exists($className)) {
    // ...
}

```

If ‘../processes/important-process.php’ is a valid file, an external user could execute the file without authorization.

To avoid these or most sophisticated attacks, *Phalcon\Loader* removes invalid characters from the class name, reducing the possibility of being attacked.

Registering Namespaces

If you’re organizing your code using namespaces, or using external libraries which do, the `registerNamespaces()` method provides the autoloading mechanism. It takes an associative array; the keys are namespace prefixes and their values are directories where the classes are located in. The namespace separator will be replaced by the directory separator when the loader tries to find the classes. Always remember to add a trailing slash at the end of the paths.

```
<?php

use Phalcon\Loader;

// Creates the autoloader
$loader = new Loader();

// Register some namespaces
$loader->registerNamespaces(
    [
        "Example\Base"    => "vendor/example/base/",
        "Example\Adapter" => "vendor/example/adapter/",
        "Example"          => "vendor/example/",
    ]
);

// Register autoloader
$loader->register();

// The required class will automatically include the
// file vendor/example/adapter/Some.php
$some = new \Example\Adapter\Some();
```

Registering Directories

The third option is to register directories, in which classes could be found. This option is not recommended in terms of performance, since Phalcon will need to perform a significant number of file stats on each folder, looking for the file with the same name as the class. It’s important to register the directories in relevance order. Remember always add a trailing slash at the end of the paths.

```
<?php

use Phalcon\Loader;

// Creates the autoloader
$loader = new Loader();

// Register some directories
$loader->registerDirs(
    [
        "library/MyComponent/",
        "library/OtherComponent/Other/",
        "vendor/example/adapters/",
    ]
);
```

```

        "vendor/example/",
    ]
);

// Register autoloader
$loader->register();

// The required class will automatically include the file from
// the first directory where it has been located
// i.e. library/OtherComponent/Other/Some.php
$some = new \Some();

```

Registering Classes

The last option is to register the class name and its path. This autoloader can be very useful when the folder convention of the project does not allow for easy retrieval of the file using the path and the class name. This is the fastest method of autoloading. However the more your application grows, the more classes/files need to be added to this autoloader, which will effectively make maintenance of the class list very cumbersome and it is not recommended.

```

<?php

use Phalcon\Loader;

// Creates the autoloader
$loader = new Loader();

// Register some classes
$loader->registerClasses([
    "Some"          => "library/OtherComponent/Other/Some.php",
    "Example\Base" => "vendor/example/adapters/Example/BaseClass.php",
]);

// Register autoloader
$loader->register();

// Requiring a class will automatically include the file it references
// in the associative array
// i.e. library/OtherComponent/Other/Some.php
$some = new \Some();

```

Registering Files

You can also registers files that are “non-classes” hence needing a “require”. This is very useful for including files that only have functions:

```

<?php

use Phalcon\Loader;

// Creates the autoloader
$loader = new Loader();

// Register some classes

```

```
$loader->registerFiles(
    [
        "functions.php",
        "arrayFunctions.php",
    ]
);

// Register autoloader
$loader->register();
```

These files are automatically loaded in the `register()` method.

Additional file extensions

Some autoloading strategies such as “prefixes”, “namespaces” or “directories” automatically append the “php” extension at the end of the checked file. If you are using additional extensions you could set it with the method “`setExtensions`”. Files are checked in the order as it were defined:

```
<?php

use Phalcon\Loader;

// Creates the autoloader
$loader = new Loader();

// Set file extensions to check
$loader->setExtensions(
    [
        "php",
        "inc",
        "phb",
    ]
);
```

Modifying current strategies

Additional auto-loading data can be added to existing values by passing “true” as the second parameter:

```
<?php

// Adding more directories
$loader->registerDirs(
    [
        "../app/library/",
        "../app/plugins/",
    ],
    true
);
```

Autoloading Events

In the following example, the `EventManager` is working with the class loader, allowing us to obtain debugging information regarding the flow of operation:

```

<?php

use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;
use Phalcon\Loader;

$eventsManager = new EventsManager();

$loader = new Loader();

$loader->registerNamespaces(
    [
        "Example\\Base"      => "vendor/example/base/",
        "Example\\Adapter"    => "vendor/example/adapter/",
        "Example"             => "vendor/example/",
    ]
);

// Listen all the loader events
$eventsManager->attach(
    "loader:beforeCheckPath",
    function (Event $event, Loader $loader) {
        echo $loader->getCheckedPath();
    }
);

$loader->setEventsManager($eventsManager);

$loader->register();

```

Some events when returning boolean false could stop the active operation. The following events are supported:

Nom d'évt	Triggered
beforeCheckClass	Triggered before starting the autoloading process
pathFound	Triggered when the loader locate a class
afterCheckClass	Triggered after finish the autoloading process. If this event is launched the autoloader didn't find the class file

Troubleshooting

Some things to keep in mind when using the universal autoloader:

- Auto-loading process is case-sensitive, the class will be loaded as it is written in the code
- Strategies based on namespaces/prefixes are faster than the directories strategy
- If a cache bytecode like [APC](#) is installed this will be used to retrieve the requested file (an implicit caching of the file is performed)

2.3.43 Logging

Phalcon\Logger is a component whose purpose is to provide logging services for applications. It offers logging to different backends using different adapters. It also offers transaction logging, configuration options, different formats and filters. You can use the *Phalcon\Logger* for every logging need your application has, from debugging processes to tracing application flow.

Adapters

This component makes use of adapters to store the logged messages. The use of adapters allows for a common logging interface which provides the ability to easily switch backends if necessary. The adapters supported are:

Adapter	Description
<i>Phalcon\Logger\Adapter\File</i>	Logs to a plain text file
<i>Phalcon\Logger\Adapter\Stream</i>	Logs to a PHP Streams
<i>Phalcon\Logger\Adapter\Syslog</i>	Logs to the system logger
<i>Phalcon\Logger\Adapter\FirePHP</i>	Logs to the FirePHP

Creating a Log

The example below shows how to create a log and add messages to it:

```
<?php

use Phalcon\Logger;
use Phalcon\Logger\Adapter\File as FileAdapter;

$logger = new FileAdapter("app/logs/test.log");

// These are the different log levels available:

$logger->critical(
    "This is a critical message"
);

$logger->emergency(
    "This is an emergency message"
);

$logger->debug(
    "This is a debug message"
);

$logger->error(
    "This is an error message"
);

$logger->info(
    "This is an info message"
);

$logger->notice(
    "This is a notice message"
);

$logger->warning(
    "This is a warning message"
);

$logger->alert(
    "This is an alert message"
);
```



```
// You can also use the log() method with a Logger constant:
$logger->log(
    "This is another error message",
    Logger::ERROR
);

// If no constant is given, DEBUG is assumed.
$logger->log(
    "This is a message"
);

// You can also pass context parameters like this
$logger->log(
    "This is a {message}",
    [
        'message' => 'parameter'
    ]
);
```

The log generated is below:

```
[Tue, 28 Jul 15 22:09:02 -0500][CRITICAL] This is a critical message
[Tue, 28 Jul 15 22:09:02 -0500][EMERGENCY] This is an emergency message
[Tue, 28 Jul 15 22:09:02 -0500][DEBUG] This is a debug message
[Tue, 28 Jul 15 22:09:02 -0500][ERROR] This is an error message
[Tue, 28 Jul 15 22:09:02 -0500][INFO] This is an info message
[Tue, 28 Jul 15 22:09:02 -0500][NOTICE] This is a notice message
[Tue, 28 Jul 15 22:09:02 -0500][WARNING] This is a warning message
[Tue, 28 Jul 15 22:09:02 -0500][ALERT] This is an alert message
[Tue, 28 Jul 15 22:09:02 -0500][ERROR] This is another error message
[Tue, 28 Jul 15 22:09:02 -0500][DEBUG] This is a message
[Tue, 28 Jul 15 22:09:02 -0500][DEBUG] This is a parameter
```

You can also set a log level using the `setLogLevel()` method. This method takes a Logger constant and will only save log messages that are as important or more important than the constant:

```
<?php

use Phalcon\Logger;
use Phalcon\Logger\Adapter\File as FileAdapter;

$logger = new FileAdapter("app/logs/test.log");

$logger->setLogLevel(
    Logger::CRITICAL
);
```

In the example above, only critical and emergency messages will get saved to the log. By default, everything is saved.

Transactions

Logging data to an adapter i.e. File (file system) is always an expensive operation in terms of performance. To combat that, you can take advantage of logging transactions. Transactions store log data temporarily in memory and later on write the data to the relevant adapter (File in this case) in a single atomic operation.

```
<?php

use Phalcon\Logger\Adapter\File as FileAdapter;

// Create the logger
$logger = new FileAdapter("app/logs/test.log");

// Start a transaction
$logger->begin();

// Add messages

$logger->alert(
    "This is an alert"
);

$logger->error(
    "This is another error"
);

// Commit messages to file
$logger->commit();
```

Logging to Multiple Handlers

Phalcon\Logger can send messages to multiple handlers with a just single call:

```
<?php

use Phalcon\Logger;
use Phalcon\Logger\Multiple as MultipleStream;
use Phalcon\Logger\Adapter\File as FileAdapter;
use Phalcon\Logger\Adapter\Stream as StreamAdapter;

$logger = new MultipleStream();

$logger->push(
    new FileAdapter("test.log")
);

$logger->push(
    new StreamAdapter("php://stdout")
);

$logger->log(
    "This is a message"
);

$logger->log(
    "This is an error",
    Logger::ERROR
);
```

```
$logger->error(
    "This is another error"
);
```

The messages are sent to the handlers in the order they were registered.

Message Formatting

This component makes use of ‘formatters’ to format messages before sending them to the backend. The formatters available are:

Adapter	Description
<i>Phalcon\Logger\Formatter\Line</i>	Formats the messages using a one-line string
<i>Phalcon\Logger\Formatter\Firephp</i>	Formats the messages so that they can be sent to FirePHP
<i>Phalcon\Logger\Formatter\Json</i>	Prepares a message to be encoded with JSON
<i>Phalcon\Logger\Formatter\Syslog</i>	Prepares a message to be sent to syslog

Line Formatter

Formats the messages using a one-line string. The default logging format is:

```
[%date%] [%type%] %message%
```

You can change the default format using `setFormat()`, this allows you to change the format of the logged messages by defining your own. The log format variables allowed are:

Variable	Description
<code>%message%</code>	The message itself expected to be logged
<code>%date%</code>	Date the message was added
<code>%type%</code>	Uppercase string with message type

The example below shows how to change the log format:

```
<?php

use Phalcon\Logger\Formatter\Line as LineFormatter;

$formatter = new LineFormatter("%date% - %message%");

// Changing the logger format
$logger->setFormatter($formatter);
```

Implementing your own formatters

The *Phalcon\Logger\FormatterInterface* interface must be implemented in order to create your own logger formatter or extend the existing ones.

Adapters

The following examples show the basic use of each adapter:

Stream Logger

The stream logger writes messages to a valid registered stream in PHP. A list of streams is available [here](#):

```
<?php

use Phalcon\Logger\Adapter\Stream as StreamAdapter;

// Opens a stream using zlib compression
$logger = new StreamAdapter("compress.zlib://week.log.gz");

// Writes the logs to stderr
$logger = new StreamAdapter("php://stderr");
```

File Logger

This logger uses plain files to log any kind of data. By default all logger files are opened using append mode which opens the files for writing only; placing the file pointer at the end of the file. If the file does not exist, an attempt will be made to create it. You can change this mode by passing additional options to the constructor:

```
<?php

use Phalcon\Logger\Adapter\File as FileAdapter;

// Create the file logger in 'w' mode
$logger = new FileAdapter(
    "app/logs/test.log",
    [
        "mode" => "w",
    ]
);
```

Syslog Logger

This logger sends messages to the system logger. The syslog behavior may vary from one operating system to another.

```
<?php

use Phalcon\Logger\Adapter\Syslog as SyslogAdapter;

// Basic Usage
$logger = new SyslogAdapter(null);

// Setting ident/mode/facility
$logger = new SyslogAdapter(
    "ident-name",
    [
        "option"    => LOG_NDELAY,
        "facility"  => LOG_MAIL,
    ]
);
```

FirePHP Logger

This logger sends messages in HTTP response headers that are displayed by [FirePHP](#), a [Firebug](#) extension for Firefox.

```
<?php

use Phalcon\Logger;
use Phalcon\Logger\Adapter\Firephp as Firephp;

$logger = new Firephp("");

$logger->log(
    "This is a message"
);

$logger->log(
    "This is an error",
    Logger::ERROR
);

$logger->error(
    "This is another error"
);
```

Implementing your own adapters

The *Phalcon\Logger\AdapterInterface* interface must be implemented in order to create your own logger adapters or extend the existing ones.

2.3.44 Annotations Parser

It is the first time that an annotations parser component is written in C for the PHP world. *Phalcon\Annotations* is a general purpose component that provides ease of parsing and caching annotations in PHP classes to be used in applications.

Annotations are read from docblocks in classes, methods and properties. An annotation can be placed at any position in the docblock:

```
<?php

/**
 * This is the class description
 *
 * @AmazingClass(true)
 */
class Example
{
    /**
     * This a property with a special feature
     *
     * @SpecialFeature
     */
    protected $someProperty;
```

```
/**
 * This is a method
 *
 * @SpecialFeature
 */
public function someMethod()
{
    // ...
}
```

An annotation has the following syntax:

```
/**
 * @Annotation-Name
 * @Annotation-Name(param1, param2, ...)
 */
```

Also, an annotation can be placed at any part of a docblock:

```
<?php

/**
 * This a property with a special feature
 *
 * @SpecialFeature
 *
 * More comments
 *
 * @AnotherSpecialFeature(true)
 */
```

The parser is highly flexible, the following docblock is valid:

```
<?php

/**
 * This a property with a special feature @SpecialFeature({
someParameter="the value", false

}) More comments @AnotherSpecialFeature(true) @MoreAnnotations
 */
```

However, to make the code more maintainable and understandable it is recommended to place annotations at the end of the docblock:

```
<?php

/**
 * This a property with a special feature
 * More comments
 *
 * @SpecialFeature({someParameter="the value", false})
 * @AnotherSpecialFeature(true)
 */
```

Reading Annotations

A reflector is implemented to easily get the annotations defined on a class using an object-oriented interface:

```
<?php

use Phalcon\Annotations\Adapter\Memory as MemoryAdapter;

$reader = new MemoryAdapter();

// Reflect the annotations in the class Example
$reflector = $reader->get("Example");

// Read the annotations in the class' docblock
$annotations = $reflector->getClassAnnotations();

// Traverse the annotations
foreach ($annotations as $annotation) {
    // Print the annotation name
    echo $annotation->getName(), PHP_EOL;

    // Print the number of arguments
    echo $annotation->numberArguments(), PHP_EOL;

    // Print the arguments
    print_r($annotation->getArguments());
}
```

The annotation reading process is very fast, however, for performance reasons it is recommended to store the parsed annotations using an adapter. Adapters cache the processed annotations avoiding the need of parse the annotations again and again.

Phalcon\Annotations\Adapter\Memory was used in the above example. This adapter only caches the annotations while the request is running and for this reason the adapter is more suitable for development. There are other adapters to swap out when the application is in production stage.

Types of Annotations

Annotations may have parameters or not. A parameter could be a simple literal (strings, number, boolean, null), an array, a hashed list or other annotation:

```
<?php

/**
 * Simple Annotation
 *
 * @SomeAnnotation
 */

/**
 * Annotation with parameters
 *
 * @SomeAnnotation("hello", "world", 1, 2, 3, false, true)
 */

/**
 * Annotation with named parameters
```

```
*
* @SomeAnnotation(first="hello", second="world", third=1)
* @SomeAnnotation(first: "hello", second: "world", third: 1)
*/

/**
 * Passing an array
 *
 * @SomeAnnotation([1, 2, 3, 4])
 * @SomeAnnotation({1, 2, 3, 4})
 */

/**
 * Passing a hash as parameter
 *
 * @SomeAnnotation({first=1, second=2, third=3})
 * @SomeAnnotation({'first'=1, 'second'=2, 'third'=3})
 * @SomeAnnotation({'first': 1, 'second': 2, 'third': 3})
 * @SomeAnnotation(['first': 1, 'second': 2, 'third': 3])
 */

/**
 * Nested arrays/hashes
 *
 * @SomeAnnotation({"name"="SomeName", "other"={
 *     "foo1": "bar1", "foo2": "bar2", {1, 2, 3},
 * }})
 */

/**
 * Nested Annotations
 *
 * @SomeAnnotation(first=@AnotherAnnotation(1, 2, 3))
 */
```

Practical Usage

Next we will explain some practical examples of annotations in PHP applications:

Cache Enabler with Annotations

Let's pretend we've created the following controller and you want to create a plugin that automatically starts the cache if the last action executed is marked as cacheable. First off all, we register a plugin in the Dispatcher service to be notified when a route is executed:

```
<?php

use Phalcon\Mvc\Dispatcher as MvcDispatcher;
use Phalcon\Events\Manager as EventsManager;

$di["dispatcher"] = function () {
    $eventsManager = new EventsManager();

    // Attach the plugin to 'dispatch' events
    $eventsManager->attach(
```



```

        "dispatch",
        new CacheEnablerPlugin()
    );

    $dispatcher = new MvcDispatcher();

    $dispatcher->setEventManager($eventsManager);

    return $dispatcher;
};

```

CacheEnablerPlugin is a plugin that intercepts every action executed in the dispatcher enabling the cache if needed:

```

<?php

use Phalcon\Events\Event;
use Phalcon\Mvc\Dispatcher;
use Phalcon\Mvc\User\Plugin;

/**
 * Enables the cache for a view if the latest
 * executed action has the annotation @Cache
 */
class CacheEnablerPlugin extends Plugin
{
    /**
     * This event is executed before every route is executed in the dispatcher
     */
    public function beforeExecuteRoute(Event $event, Dispatcher $dispatcher)
    {
        // Parse the annotations in the method currently executed
        $annotations = $this->annotations->getMethod(
            $dispatcher->getControllerClass(),
            $dispatcher->getActiveMethod()
        );

        // Check if the method has an annotation 'Cache'
        if ($annotations->has("Cache")) {
            // The method has the annotation 'Cache'
            $annotation = $annotations->get("Cache");

            // Get the lifetime
            $lifetime = $annotation->getNamedParameter("lifetime");

            $options = [
                "lifetime" => $lifetime,
            ];

            // Check if there is a user defined cache key
            if ($annotation->hasNamedParameter("key")) {
                $options["key"] = $annotation->getNamedParameter("key");
            }

            // Enable the cache for the current method
            $this->view->cache($options);
        }
    }
}

```

Now, we can use the annotation in a controller:

```
<?php

use Phalcon\Mvc\Controller;

class NewsController extends Controller
{
    public function indexAction()
    {

    }

    /**
     * This is a comment
     *
     * @Cache(lifetime=86400)
     */
    public function showAllAction()
    {
        $this->view->article = Articles::find();
    }

    /**
     * This is a comment
     *
     * @Cache(key="my-key", lifetime=86400)
     */
    public function showAction($slug)
    {
        $this->view->article = Articles::findFirstByTitle($slug);
    }
}
```

Private/Public areas with Annotations

You can use annotations to tell the ACL which controllers belong to the administrative areas:

```
<?php

use Phalcon\Acl;
use Phalcon\Acl\Role;
use Phalcon\Acl\Resource;
use Phalcon\Events\Event;
use Phalcon\Mvc\User\Plugin;
use Phalcon\Mvc\Dispatcher;
use Phalcon\Acl\Adapter\Memory as AclList;

/**
 * This is the security plugin which controls that users only have access to the
 * ↪ modules they're assigned to
 */
class SecurityAnnotationsPlugin extends Plugin
{
    /**
     * This action is executed before execute any action in the application
     *
     */
}
```

```

* @param Event $event
* @param Dispatcher $dispatcher
*/
public function beforeDispatch(Event $event, Dispatcher $dispatcher)
{
    // Possible controller class name
    $controllerName = $dispatcher->getControllerClass();

    // Possible method name
    $actionName = $dispatcher->getActiveMethod();

    // Get annotations in the controller class
    $annotations = $this->annotations->get($controllerName);

    // The controller is private?
    if ($annotations->getClassAnnotations()->has("Private")) {
        // Check if the session variable is active?
        if (!$this->session->get("auth")) {

            // The user is no logged redirect to login
            $dispatcher->forward(
                [
                    "controller" => "session",
                    "action"      => "login",
                ]
            );

            return false;
        }
    }

    // Continue normally
    return true;
}
}

```

Annotations Adapters

This component makes use of adapters to cache or no cache the parsed and processed annotations thus improving the performance or providing facilities to development/testing:

Class	Description
<i>Phalcon\Annotations\Adapter\Memory</i>	The annotations are cached only in memory. When the request ends the cache is cleaned reloading the annotations in each request. This adapter is suitable for a development stage
<i>Phalcon\Annotations\Adapter\PhpFile</i>	Parsed and processed annotations are stored permanently in PHP files improving performance. This adapter must be used together with a bytecode cache.
<i>Phalcon\Annotations\Adapter\Apc</i>	Parsed and processed annotations are stored permanently in the APC cache improving performance. This is the faster adapter
<i>Phalcon\Annotations\Adapter\Xcache</i>	Parsed and processed annotations are stored permanently in the XCache cache improving performance. This is a fast adapter too

Implementing your own adapters

The *Phalcon\Annotations\AdapterInterface* interface must be implemented in order to create your own annotations adapters or extend the existing ones.

External Resources

- [Tutorial: Creating a custom model's initializer with Annotations](#)

2.3.45 Applications en Ligne de Commande

Les applications CLI sont exécutées depuis la ligne de commande. Elles sont utiles pour créer de jobs planifiés, des scripts, des utilitaires et bien plus encore.

Structure

La structure minimale d'une application CLI doit ressembler à ceci:

- app/config/config.php
- app/tasks/MainTask.php
- app/cli.php ← fichier d'amorce principal

Création de l'Amorce

Comme dans les applications MVC classiques, le fichier d'amorce est utilisé pour amorcer l'application. Au lieu du traditionnel index.php des application web, nous utilisons un fichier cli.php comme point d'entrée de l'application.

Ci-dessous un exemple qui sera utilisé pour notre exemple.

```
<?php

use Phalcon\Di\FactoryDefault\Cli as CliDI;
use Phalcon\Cli\Console as ConsoleApp;
use Phalcon\Loader;

// Utilise le conteneur de services CLI par défaut
$di = new CliDI();

/**
 * Inscription d'un chargeur automatique et lui indique le chemin des tâches
 */
$loader = new Loader();

$loader->registerDirs(
    [
        __DIR__ . "/tasks",
    ]
);
```

```

$loader->register();

// Chargement de la configuration (si elle existe)
$configFile = __DIR__ . "/config/config.php";

if (is_readable($configFile)) {
    $config = include $configFile;

    $di->set("config", $config);
}

// Création de l'application console
$console = new ConsoleApp();

$console->setDI($di);

/**
 * Traitement des arguments
 */
$arguments = [];

foreach ($argv as $k => $arg) {
    if ($k === 1) {
        $arguments["task"] = $arg;
    } elseif ($k === 2) {
        $arguments["action"] = $arg;
    } elseif ($k >= 3) {
        $arguments["params"][] = $arg;
    }
}

try {
    // Gestion des arguments transmis
    $console->handle($arguments);
} catch (\Phalcon\Exception $e) {
    echo $e->getMessage();

    exit(255);
}

```

Cet extrait de code peut être exécuté ainsi:

```

$ php app/cli.php

Ceci est la tache 'default' et l'action 'default'

```

Tâches

Le fonctionnement des tâches est similaire à celui des contrôleurs. Chaque application nécessite au moins “MainTask” et “mainAction” et chaque tâche une “mainAction” qui sera exécutée si aucune action n’est indiquée explicitement.

Ci-dessous se trouve un exemple du fichier app/tasks/MainTask.php:

```
<?php

use Phalcon\Cli\Task;

class MainTask extends Task
{
    public function mainAction()
    {
        echo "Ceci est la tache 'default' et l'action 'default'" . PHP_EOL;
    }
}
```

Traitement des paramètre de l’action

Il est possible de transmettre des paramètres aux actions. Le code pour réaliser ceci existe déjà dans l’exemple d’amorce.

Si vous lancer l’application avec l’action et les paramètres suivants:

```
<?php

use Phalcon\Cli\Task;

class MainTask extends Task
{
    public function mainAction()
    {
        echo "Ceci est la tache 'default' et l'action 'default'" . PHP_EOL;
    }

    /**
     * @param array $params
     */
    public function testAction(array $params)
    {
        echo sprintf(
            "bonjour %s",
            $params[0]
        );

        echo PHP_EOL;

        echo sprintf(
            "cordialement, %s",
            $params[1]
        );

        echo PHP_EOL;
    }
}
```

On peut désormais lancer la commande suivante:

```
$ php app/cli.php main test monde univers

salut monde
cordialement, univers
```

Enchaînement de tâches

Il est également possible d'enchaîner les tâches si nécessaire. Pour réaliser ceci, vous devez ajouter la console elle-même au DI:

```
<?php

$di->setShared("console", $console);

try {
    // Gestion des arguments fournis
    $console->handle($arguments);
} catch (\Phalcon\Exception $e) {
    echo $e->getMessage();

    exit(255);
}
```

Ainsi vous pouvez utiliser la console à l'intérieur de n'importe quelle tâche. L'exemple ci-dessous est une version modifiée de MainTask.php:

```
<?php

use Phalcon\Cli\Task;

class MainTask extends Task
{
    public function mainAction()
    {
        echo "Ceci est la tache 'default' et l'action 'default'" . PHP_EOL;

        $this->console->handle(
            [
                "task"    => "main",
                "action" => "test",
            ]
        );
    }

    public function testAction()
    {
        echo "Je serais imprime aussi !" . PHP_EOL;
    }
}
```

Cependant, ce serait une meilleure idée que d'étendre *Phalcon\Cli\Task* et développer ce type de logique ici.

2.3.46 Images

Phalcon\Image is the component that allows you to manipulate image files. Multiple operations can be performed on the same image object.

This guide is not intended to be a complete documentation of available methods and their arguments. Please visit the [API](#) for a complete reference.

Adapters

This component makes use of adapters to encapsulate specific image manipulator programs. The following image manipulator programs are supported:

Class	Description
<i>Phalcon\Image\Adapter\Gd</i>	Requires the GD PHP extension.
<i>Phalcon\Image\Adapter\Imagick</i>	Requires the ImageMagick PHP extension.

Implementing your own adapters

The *Phalcon\Image\AdapterInterface* interface must be implemented in order to create your own image adapters or extend the existing ones.

Saving and rendering images

Before we begin with the various features of the image component, it's worth understanding how to save and render these images.

```
<?php
$image = new \Phalcon\Image\Adapter\Gd("image.jpg");
// ...

// Overwrite the original image
$image->save();
```

```
<?php
$image = new \Phalcon\Image\Adapter\Gd("image.jpg");
// ...

// Save to 'new-image.jpg'
$image->save("new-image.jpg");
```

You can also change the format of the image:

```
<?php
$image = new \Phalcon\Image\Adapter\Gd("image.jpg");
// ...
```



```
// Save as a PNG file
$image->save("image.png");
```

When saving as a JPEG, you can also specify the quality as the second parameter:

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");

// ...

// Save as a JPEG with 80% quality
$image->save("image.jpg", 80);
```

Resizing images

There are several modes of resizing:

- \Phalcon\Image::WIDTH
- \Phalcon\Image::HEIGHT
- \Phalcon\Image::NONE
- \Phalcon\Image::TENSILE
- \Phalcon\Image::AUTO
- \Phalcon\Image::INVERSE
- \Phalcon\Image::PRECISE

\Phalcon\Image::WIDTH

The height will automatically be generated to keep the proportions the same; if you specify a height, it will be ignored.

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");

$image->resize(
    300,
    null,
    \Phalcon\Image::WIDTH
);

$image->save("resized-image.jpg");
```

\Phalcon\Image::HEIGHT

The width will automatically be generated to keep the proportions the same; if you specify a width, it will be ignored.

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");
```

```
$image->resize(
    null,
    300,
    \Phalcon\Image::HEIGHT
);

$image->save("resized-image.jpg");
```

`\Phalcon\Image::NONE`

The `NONE` constant ignores the original image's ratio. Neither width and height are required. If a dimension is not specified, the original dimension will be used. If the new proportions differ from the original proportions, the image may be distorted and stretched.

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");

$image->resize(
    400,
    200,
    \Phalcon\Image::NONE
);

$image->save("resized-image.jpg");
```

`\Phalcon\Image::TENSILE`

Similar to the `NONE` constant, the `TENSILE` constant ignores the original image's ratio. Both width and height are required. If the new proportions differ from the original proportions, the image may be distorted and stretched.

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");

$image->resize(
    400,
    200,
    \Phalcon\Image::NONE
);

$image->save("resized-image.jpg");
```

Cropping images

For example, to get a 100px by 100px square from the centre of the image:

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");

$width  = 100;
```

```
$height = 100;
$offsetX = (($image->getWidth() - $width) / 2);
$offsetY = (($image->getHeight() - $height) / 2);

$image->crop($width, $height, $offsetX, $offsetY);

$image->save("cropped-image.jpg");
```

Rotating images

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");

// Rotate an image by 90 degrees clockwise
$image->rotate(90);

$image->save("rotated-image.jpg");
```

Flipping images

You can flip an image horizontally (using the `\Phalcon\Image::HORIZONTAL` constant) and vertically (using the `\Phalcon\Image::VERTICAL` constant):

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");

// Flip an image horizontally
$image->flip(
    \Phalcon\Image::HORIZONTAL
);

$image->save("flipped-image.jpg");
```

Sharpening images

The `sharpen()` method takes a single parameter - an integer between 0 (no effect) and 100 (very sharp):

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");

$image->sharpen(50);

$image->save("sharpened-image.jpg");
```

Adding watermarks to images

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");

$watermark = new \Phalcon\Image\Adapter\Gd("me.jpg");

// Put the watermark in the top left corner
$offsetX = 10;
$offsetY = 10;

$opacity = 70;

$image->watermark(
    $watermark,
    $offsetX,
    $offsetY,
    $opacity
);

$image->save("watermarked-image.jpg");
```

Of course, you can also manipulate the watermarked image before applying it to the main image:

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");

$watermark = new \Phalcon\Image\Adapter\Gd("me.jpg");

$watermark->resize(100, 100);
$watermark->rotate(90);
$watermark->sharpen(5);

// Put the watermark in the bottom right corner with a 10px margin
$offsetX = ($image->getWidth() - $watermark->getWidth() - 10);
$offsetY = ($image->getHeight() - $watermark->getHeight() - 10);

$opacity = 70;

$image->watermark(
    $watermark,
    $offsetX,
    $offsetY,
    $opacity
);

$image->save("watermarked-image.jpg");
```

Blurring images

The `blur()` method takes a single parameter - an integer between 0 (no effect) and 100 (very blurry):

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");
```

```
$image->blur(50);

$image->save("blurred-image.jpg");
```

Pixelating images

The `pixelate()` method takes a single parameter - the higher the integer, the more pixelated the image becomes:

```
<?php

$image = new \Phalcon\Image\Adapter\Gd("image.jpg");

$image->pixelate(10);

$image->save("pixelated-image.jpg");
```

2.3.47 File d'attente

Les activités comme la manipulation de vidéos, le recadrage d'image ou l'envoi de mails, ne sont pas adaptés pour être réalisées en ligne ou bien en temps réel à cause du temps de chargement des pages qui peut avoir un impact sérieux sur l'expérience utilisateur.

La meilleure des solutions serait de mettre en place des tâches de fond. Les applications ajouteraient les tâches à une file et ces tâches seraient traitées indépendamment.

Bien que vous puissiez trouver des extensions PHP plus sophistiquées pour la mise en file d'attente comme [RabbitMQ](#); Phalcon fournit un client pour [Beanstalk](#), un backend de mise en file d'attente de jobs inspiré de [Memcache](#). Il est simple, léger, et parfaitement spécialisé dans la mise en file d'attente.

Attention: Certains retour de méthodes de la queue nécessite la présence du module Yaml. Veuillez vous référer à <http://php.net/manual/book.yaml.php> pour plus d'information. Pour PHP < 7, Yaml 1.3.0 est acceptable. Pour PHP >= 7 vous devez utiliser Yaml >= 2.0.0.

Pousser les tâches dans la queue

Après vous être connecté à Beanstalk vous pouvez insérer autant de jobs que nécessaire. Vous pouvez définir la structure du message en fonction des besoins de l'application:

```
<?php

use Phalcon\Queue\Beanstalk;

// Connexion à la queue
$queue = new Beanstalk(
    [
        "host" => "192.168.0.21",
        "port" => "11300",
    ]
);

// Ajoute le job dans la queue
$queue->put(
```

```
[
    "processVideo" => 4871,
]
```

```
);
```

Les options de connexion disponibles sont:

Option	Description	Default
host	IP du serveur beanstalk	127.0.0.1
port	Port de connexion	11300

Dans l'exemple précédent nous stockions un message qui permettait à une tâche de fond de traiter une vidéo. Le message est enfilé immédiatement dans la queue et a une durée de vie limitée.

Pour les options supplémentaire comme le temps d'exécution, la priorité ou le délai sont passés en second paramètre:

```
<?php

// Enfile le job dans la queue avec des options
$queue->put (
    [
        "processVideo" => 4871,
    ],
    [
        "priority" => 250,
        "delay"    => 10,
        "ttr"      => 3600,
    ]
);
```

Les options suivantes sont possibles:

Op-tion	Description
priority	It's an integer $< 2^{*}32$. Jobs with smaller priority values will be scheduled before jobs with larger priorities. The most urgent priority is 0; the least urgent priority is 4,294,967,295.
delay	It's an integer number of seconds to wait before putting the job in the ready queue. The job will be in the "delayed" state during this time.
ttr	Time to run – is an integer number of seconds to allow a worker to run this job. This time is counted from the moment a worker reserves this job.

Chaque job enfilé dans la queue retourne un identifiant de job qui permet de suivre l'état du job:

```
<?php

$jobId = $queue->put (
    [
        "processVideo" => 4871,
    ]
);
```

Récupération de messages

Une fois le job placé dans la queue, ces messages sont consommés par un agent en arrière plan qui devrait avoir le temps de réaliser la tâche:

```
<?php

while (($job = $queue->peekReady()) !== false) {
    $message = $job->getBody();

    var_dump($message);

    $job->delete();
}
```

Les jobs doivent être défilés de la queue pour éviter d’être traités deux fois. Si plusieurs agents en tâche de fond sont mis en œuvre, il faut réserver les jobs pour éviter que les autres agents ne les traitent aussi.

```
<?php

while (($job = $queue->reserve()) !== false) {
    $message = $job->getBody();

    var_dump($message);

    $job->delete();
}
```

Notre client exploite un jeu élémentaire de fonctionnalités fournis par Beanstalkd mais suffisamment pour vous permettre de construire des applications qui mettent en œuvre des queues.

2.3.48 Database Abstraction Layer

Phalcon\Db is the component behind *Phalcon\Mvc\Model* that powers the model layer in the framework. It consists of an independent high-level abstraction layer for database systems completely written in C.

This component allows for a lower level database manipulation than using traditional models.

This guide is not intended to be a complete documentation of available methods and their arguments. Please visit the [API](#) for a complete reference.

Database Adapters

This component makes use of adapters to encapsulate specific database system details. Phalcon uses [PDO](#) to connect to databases. The following database engines are supported:

Class	Description
<i>Phalcon\Db\Adapter\Pdo\Mysql</i>	Is the world’s most used relational database management system (RDBMS) that runs as a server providing multi-user access to a number of databases
<i>Phalcon\Db\Adapter\Pdo\Postgresql</i>	PostgreSQL is a powerful, open source relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness.
<i>Phalcon\Db\Adapter\Pdo\Sqlite</i>	SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine

Implementing your own adapters

The *Phalcon\Db\AdapterInterface* interface must be implemented in order to create your own database adapters or extend the existing ones.

Database Dialects

Phalcon encapsulates the specific details of each database engine in dialects. Those provide common functions and SQL generator to the adapters.

Class	Description
<i>Phalcon\Db\Dialect\Mysql</i>	SQL specific dialect for MySQL database system
<i>Phalcon\Db\Dialect\Postgresql</i>	SQL specific dialect for PostgreSQL database system
<i>Phalcon\Db\Dialect\Sqlite</i>	SQL specific dialect for SQLite database system

Implementing your own dialects

The *Phalcon\Db\DialectInterface* interface must be implemented in order to create your own database dialects or extend the existing ones.

Connecting to Databases

To create a connection it's necessary instantiate the adapter class. It only requires an array with the connection parameters. The example below shows how to create a connection passing both required and optional parameters:

```
<?php

// Required
$config = [
    "host"      => "127.0.0.1",
    "username"  => "mike",
    "password"  => "sigma",
    "dbname"    => "test_db",
];

// Optional
$config["persistent"] = false;

// Create a connection
$connection = new \Phalcon\Db\Adapter\Pdo\Mysql($config);
```

```
<?php

// Required
$config = [
    "host"      => "localhost",
    "username"  => "postgres",
    "password"  => "secret1",
    "dbname"    => "template",
];

// Optional
$config["schema"] = "public";

// Create a connection
$connection = new \Phalcon\Db\Adapter\Pdo\Postgresql($config);
```

```
<?php

// Required
```



```
$config = [
    "dbname" => "/path/to/database.db",
];

// Create a connection
$connection = new \Phalcon\Db\Adapter\Pdo\Sqlite($config);
```

Setting up additional PDO options

You can set PDO options at connection time by passing the parameters ‘options’:

```
<?php

// Create a connection with PDO options
$connection = new \Phalcon\Db\Adapter\Pdo\Mysql(
    [
        "host"      => "localhost",
        "username"  => "root",
        "password"  => "sigma",
        "dbname"    => "test_db",
        "options"   => [
            PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES 'UTF8'",
            PDO::ATTR_CASE                => PDO::CASE_LOWER,
        ]
    ]
);
```

Finding Rows

Phalcon\Db provides several methods to query rows from tables. The specific SQL syntax of the target database engine is required in this case:

```
<?php

$sql = "SELECT id, name FROM robots ORDER BY name";

// Send a SQL statement to the database system
$result = $connection->query($sql);

// Print each robot name
while ($robot = $result->fetch()) {
    echo $robot["name"];
}

// Get all rows in an array
$robots = $connection->fetchAll($sql);
foreach ($robots as $robot) {
    echo $robot["name"];
}

// Get only the first row
$robot = $connection->fetchOne($sql);
```

By default these calls create arrays with both associative and numeric indexes. You can change this behavior by using *Phalcon\Db\Result::setFetchMode()*. This method receives a constant, defining which kind of index is

required.

Constant	Description
Phalcon\Db::FETCH_NUM	Return an array with numeric indexes
Phalcon\Db::FETCH_ASSOC	Return an array with associative indexes
Phalcon\Db::FETCH_BOTH	Return an array with both associative and numeric indexes
Phalcon\Db::FETCH_OBJ	Return an object instead of an array

```
<?php

$sql = "SELECT id, name FROM robots ORDER BY name";
$result = $connection->query($sql);

$result->setFetchMode(Phalcon\Db::FETCH_NUM);
while ($robot = $result->fetch()) {
    echo $robot[0];
}
```

The `Phalcon\Db::query()` returns an instance of *Phalcon\Db\Result\Pdo*. These objects encapsulate all the functionality related to the returned resultset i.e. traversing, seeking specific records, count etc.

```
<?php

$sql = "SELECT id, name FROM robots";
$result = $connection->query($sql);

// Traverse the resultset
while ($robot = $result->fetch()) {
    echo $robot["name"];
}

// Seek to the third row
$result->seek(2);
$robot = $result->fetch();

// Count the resultset
echo $result->numRows();
```

Binding Parameters

Bound parameters is also supported in *Phalcon\Db*. Although there is a minimal performance impact by using bound parameters, you are encouraged to use this methodology so as to eliminate the possibility of your code being subject to SQL injection attacks. Both string and positional placeholders are supported. Binding parameters can simply be achieved as follows:

```
<?php

// Binding with numeric placeholders
$sql = "SELECT * FROM robots WHERE name = ? ORDER BY name";
$result = $connection->query(
    $sql,
    [
        "Wall-E",
    ]
);

// Binding with named placeholders
```

```
$sql      = "INSERT INTO `robots` (name`, year) VALUES (:name, :year)";
$success = $connection->query(
    $sql,
    [
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);
```

When using numeric placeholders, you will need to define them as integers i.e. 1 or 2. In this case “1” or “2” are considered strings and not numbers, so the placeholder could not be successfully replaced. With any adapter data are automatically escaped using [PDO Quote](#).

This function takes into account the connection charset, so its recommended to define the correct charset in the connection parameters or in your database server configuration, as a wrong charset will produce undesired effects when storing or retrieving data.

Also, you can pass your parameters directly to the execute/query methods. In this case bound parameters are directly passed to PDO:

```
<?php

// Binding with PDO placeholders
$sql      = "SELECT * FROM robots WHERE name = ? ORDER BY name";
$result   = $connection->query(
    $sql,
    [
        1 => "Wall-E",
    ]
);
```

Inserting/Updating/Deleting Rows

To insert, update or delete rows, you can use raw SQL or use the preset functions provided by the class:

```
<?php

// Inserting data with a raw SQL statement
$sql      = "INSERT INTO `robots` (`name`, `year`) VALUES ('Astro Boy', 1952)";
$success = $connection->execute($sql);

// With placeholders
$sql      = "INSERT INTO `robots` (`name`, `year`) VALUES (?, ?)";
$success = $connection->execute(
    $sql,
    [
        "Astro Boy",
        1952,
    ]
);

// Generating dynamically the necessary SQL
$success = $connection->insert(
    "robots",
    [
        "Astro Boy",
        1952,
```

```
    ],
    [
        "name",
        "year",
    ],
],
);

// Generating dynamically the necessary SQL (another syntax)
$success = $connection->insertAsDict(
    "robots",
    [
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);

// Updating data with a raw SQL statement
$sql      = "UPDATE `robots` SET `name` = 'Astro boy' WHERE `id` = 101";
$success = $connection->execute($sql);

// With placeholders
$sql      = "UPDATE `robots` SET `name` = ? WHERE `id` = ?";
$success = $connection->execute(
    $sql,
    [
        "Astro Boy",
        101,
    ]
);

// Generating dynamically the necessary SQL
$success = $connection->update(
    "robots",
    [
        "name",
    ],
    [
        "New Astro Boy",
    ],
    "id = 101" // Warning! In this case values are not escaped
);

// Generating dynamically the necessary SQL (another syntax)
$success = $connection->updateAsDict(
    "robots",
    [
        "name" => "New Astro Boy",
    ],
    "id = 101" // Warning! In this case values are not escaped
);

// With escaping conditions
$success = $connection->update(
    "robots",
    [
        "name",
    ],
    [

```

```

        "New Astro Boy",
    ],
    [
        "conditions" => "id = ?",
        "bind"        => [101],
        "bindTypes"   => [PDO::PARAM_INT], // Optional parameter
    ]
);
$success = $connection->updateAsDict(
    "robots",
    [
        "name" => "New Astro Boy",
    ],
    [
        "conditions" => "id = ?",
        "bind"        => [101],
        "bindTypes"   => [PDO::PARAM_INT], // Optional parameter
    ]
);

// Deleting data with a raw SQL statement
$sql      = "DELETE `robots` WHERE `id` = 101";
$success = $connection->execute($sql);

// With placeholders
$sql      = "DELETE `robots` WHERE `id` = ?";
$success = $connection->execute($sql, [101]);

// Generating dynamically the necessary SQL
$success = $connection->delete(
    "robots",
    "id = ?",
    [
        101,
    ]
);

```

Transactions and Nested Transactions

Working with transactions is supported as it is with PDO. Perform data manipulation inside transactions often increase the performance on most database systems:

```

<?php
try {
    // Start a transaction
    $connection->begin();

    // Execute some SQL statements
    $connection->execute("DELETE `robots` WHERE `id` = 101");
    $connection->execute("DELETE `robots` WHERE `id` = 102");
    $connection->execute("DELETE `robots` WHERE `id` = 103");

    // Commit if everything goes well
    $connection->commit();
} catch (Exception $e) {

```

```
// An exception has occurred rollback the transaction
$connection->rollback();
}
```

In addition to standard transactions, *Phalcon\Db* provides built-in support for [nested transactions](#) (if the database system used supports them). When you call `begin()` for a second time a nested transaction is created:

```
<?php

try {
    // Start a transaction
    $connection->begin();

    // Execute some SQL statements
    $connection->execute("DELETE `robots` WHERE `id` = 101");

    try {
        // Start a nested transaction
        $connection->begin();

        // Execute these SQL statements into the nested transaction
        $connection->execute("DELETE `robots` WHERE `id` = 102");
        $connection->execute("DELETE `robots` WHERE `id` = 103");

        // Create a save point
        $connection->commit();
    } catch (Exception $e) {
        // An error has occurred, release the nested transaction
        $connection->rollback();
    }

    // Continue, executing more SQL statements
    $connection->execute("DELETE `robots` WHERE `id` = 104");

    // Commit if everything goes well
    $connection->commit();
} catch (Exception $e) {
    // An exception has occurred rollback the transaction
    $connection->rollback();
}
```

Database Events

Phalcon\Db is able to send events to a *EventsManager* if it's present. Some events when returning boolean false could stop the active operation. The following events are supported:

Nom d'évt	Triggered	Opération stoppée ?
afterConnect	After a successfully connection to a database system	Non
beforeQuery	Before send a SQL statement to the database system	Oui
afterQuery	After send a SQL statement to database system	Non
beforeDisconnect	Before close a temporal database connection	Non
beginTransaction	Before a transaction is going to be started	Non
rollbackTransaction	Before a transaction is rolledback	Non
commitTransaction	Before a transaction is committed	Non

Bind an *EventsManager* to a connection is simple, *Phalcon\Db* will trigger the events with the type "db":

```

<?php

use Phalcon\Events\Manager as EventsManager;
use Phalcon\Db\Adapter\Pdo\Mysql as Connection;

$eventsManager = new EventsManager();

// Listen all the database events
$eventsManager->attach('db', $dbListener);

$connection = new Connection(
    [
        "host"      => "localhost",
        "username"  => "root",
        "password"  => "secret",
        "dbname"    => "invo",
    ]
);

// Assign the eventsManager to the db adapter instance
$connection->setEventsManager($eventsManager);

```

Stop SQL operations are very useful if for example you want to implement some last-resource SQL injector checker:

```

<?php

use Phalcon\Events\Event;

$eventsManager->attach(
    "db:beforeQuery",
    function (Event $event, $connection) {
        $sql = $connection->getSQLStatement();

        // Check for malicious words in SQL statements
        if (preg_match("/DROP|ALTER/i", $sql)) {
            // DROP/ALTER operations aren't allowed in the application,
            // this must be a SQL injection!
            return false;
        }

        // It's OK
        return true;
    }
);

```

Profiling SQL Statements

Phalcon\Db includes a profiling component called *Phalcon\Db\Profiler*, that is used to analyze the performance of database operations so as to diagnose performance problems and discover bottlenecks.

Database profiling is really easy With *Phalcon\Db\Profiler*:

```

<?php

use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;
use Phalcon\Db\Profiler as DbProfiler;

```

```
$eventsManager = new EventsManager();

$profiler = new DbProfiler();

// Listen all the database events
$eventsManager->attach(
    "db",
    function (Event $event, $connection) use ($profiler) {
        if ($event->getType() === "beforeQuery") {
            $sql = $connection->getSQLStatement();

            // Start a profile with the active connection
            $profiler->startProfile($sql);
        }

        if ($event->getType() === "afterQuery") {
            // Stop the active profile
            $profiler->stopProfile();
        }
    }
);

// Assign the events manager to the connection
$connection->setEventsManager($eventsManager);

$sql = "SELECT buyer_name, quantity, product_name "
    . "FROM buyers "
    . "LEFT JOIN products ON buyers.pid = products.id";

// Execute a SQL statement
$connection->query($sql);

// Get the last profile in the profiler
$profile = $profiler->getLastProfile();

echo "SQL Statement: ", $profile->getSQLStatement(), "\n";
echo "Start Time: ", $profile->getInitialTime(), "\n";
echo "Final Time: ", $profile->getFinalTime(), "\n";
echo "Total Elapsed Time: ", $profile->getTotalElapsedSeconds(), "\n";
```

You can also create your own profile class based on *Phalcon\Db\Profiler* to record real time statistics of the statements sent to the database system:

```
<?php

use Phalcon\Events\Manager as EventsManager;
use Phalcon\Db\Profiler as Profiler;
use Phalcon\Db\Profiler\Item as Item;

class DbProfiler extends Profiler
{
    /**
     * Executed before the SQL statement will sent to the db server
     */
    public function beforeStartProfile(Item $profile)
    {
        echo $profile->getSQLStatement();
    }
}
```



```

    }

    /**
     * Executed after the SQL statement was sent to the db server
     */
    public function afterEndProfile(Item $profile)
    {
        echo $profile->getTotalElapsedSeconds();
    }
}

// Create an Events Manager
$eventsManager = new EventsManager();

// Create a listener
$dbProfiler = new DbProfiler();

// Attach the listener listening for all database events
$eventsManager->attach("db", $dbProfiler);

```

Logging SQL Statements

Using high-level abstraction components such as *Phalcon\Db* to access a database, it is difficult to understand which statements are sent to the database system. *Phalcon\Logger* interacts with *Phalcon\Db*, providing logging capabilities on the database abstraction layer.

```

<?php

use Phalcon\Logger;
use Phalcon\Events\Event;
use Phalcon\Events\Manager as EventsManager;
use Phalcon\Logger\Adapter\File as FileLogger;

$eventsManager = new EventsManager();

$logger = new FileLogger("app/logs/db.log");

$eventsManager->attach(
    "db:beforeQuery",
    function (Event $event, $connection) use ($logger) {
        $sql = $connection->getSQLStatement();

        $logger->log($sql, Logger::INFO);
    }
);

// Assign the eventsManager to the db adapter instance
$connection->setEventsManager($eventsManager);

// Execute some SQL statement
$connection->insert(
    "products",
    [
        "Hot pepper",
        3.50,
    ],

```

```
[
    "name",
    "price",
]
);
```

As above, the file `app/logs/db.log` will contain something like this:

```
[Sun, 29 Apr 12 22:35:26 -0500][DEBUG][Resource Id #77] INSERT INTO products
(name, price) VALUES ('Hot pepper', 3.50)
```

Implementing your own Logger

You can implement your own logger class for database queries, by creating a class that implements a single method called “log”. The method needs to accept a string as the first argument. You can then pass your logging object to `Phalcon\Db::setLogger()`, and from then on any SQL statement executed will call that method to log the results.

Describing Tables/Views

Phalcon\Db also provides methods to retrieve detailed information about tables and views:

```
<?php

// Get tables on the test_db database
$tables = $connection->listTables("test_db");

// Is there a table 'robots' in the database?
$exists = $connection->tableExists("robots");

// Get name, data types and special features of 'robots' fields
$fields = $connection->describeColumns("robots");
foreach ($fields as $field) {
    echo "Column Type: ", $field["Type"];
}

// Get indexes on the 'robots' table
$indexes = $connection->describeIndexes("robots");
foreach ($indexes as $index) {
    print_r(
        $index->getColumns()
    );
}

// Get foreign keys on the 'robots' table
$references = $connection->describeReferences("robots");
foreach ($references as $reference) {
    // Print referenced columns
    print_r(
        $reference->getReferencedColumns()
    );
}
```

A table description is very similar to the MySQL describe command, it contains the following information:

Index	Description
Field	Field's name
Type	Column Type
Key	Is the column part of the primary key or an index?
Null	Does the column allow null values?

Methods to get information about views are also implemented for every supported database system:

```
<?php

// Get views on the test_db database
$tables = $connection->listViews("test_db");

// Is there a view 'robots' in the database?
$exists = $connection->viewExists("robots");
```

Creating/Altering/Dropping Tables

Different database systems (MySQL, Postgresql etc.) offer the ability to create, alter or drop tables with the use of commands such as CREATE, ALTER or DROP. The SQL syntax differs based on which database system is used. *Phalcon\Db* offers a unified interface to alter tables, without the need to differentiate the SQL syntax based on the target storage system.

Creating Tables

The following example shows how to create a table:

```
<?php

use \Phalcon\Db\Column as Column;

$connection->createTable(
    "robots",
    null,
    [
        "columns" => [
            new Column(
                "id",
                [
                    "type"      => Column::TYPE_INTEGER,
                    "size"      => 10,
                    "notNull"   => true,
                    "autoIncrement" => true,
                    "primary"   => true,
                ]
            ),
            new Column(
                "name",
                [
                    "type"      => Column::TYPE_VARCHAR,
                    "size"      => 70,
                    "notNull"   => true,
                ]
            ),
            new Column(
```

```

        "year",
        [
            "type"      => Column::TYPE_INTEGER,
            "size"      => 11,
            "notNull"   => true,
        ]
    ),
]
]
);

```

`Phalcon\Db::createTable()` accepts an associative array describing the table. Columns are defined with the class *Phalcon\Db\Column*. The table below shows the options available to define a column:

Option	Description	Optional
"type"	Column type. Must be a <i>Phalcon\Db\Column</i> constant (see below for a list)	Non
"primary"	True if the column is part of the table's primary key	Oui
"size"	Some type of columns like VARCHAR or INTEGER may have a specific size	Oui
"scale"	DECIMAL or NUMBER columns may have a scale to specify how many decimals should be stored	Oui
"un-signed"	INTEGER columns may be signed or unsigned. This option does not apply to other types of columns	Oui
"notNull"	Column can store null values?	Oui
"default"	Default value (when used with "notNull" => true).	Oui
"autoIncrement"	With this attribute column will filled automatically with an auto-increment integer. Only one column in the table can have this attribute.	Oui
"bind"	One of the BIND_TYPE_* constants telling how the column must be binded before save it	Oui
"first"	Column must be placed at first position in the column order	Oui
"after"	Column must be placed after indicated column	Oui

Phalcon\Db supports the following database column types:

- `Phalcon\Db\Column::TYPE_INTEGER`
- `Phalcon\Db\Column::TYPE_DATE`
- `Phalcon\Db\Column::TYPE_VARCHAR`
- `Phalcon\Db\Column::TYPE_DECIMAL`
- `Phalcon\Db\Column::TYPE_DATETIME`
- `Phalcon\Db\Column::TYPE_CHAR`
- `Phalcon\Db\Column::TYPE_TEXT`

The associative array passed in `Phalcon\Db::createTable()` can have the possible keys:

Index	Description	Optional
"columns"	An array with a set of table columns defined with <i>Phalcon\Db\Column</i>	Non
"indexes"	An array with a set of table indexes defined with <i>Phalcon\Db/Index</i>	Oui
"references"	An array with a set of table references (foreign keys) defined with <i>Phalcon\Db/Reference</i>	Oui
"options"	An array with a set of table creation options. These options often relate to the database system in which the migration was generated.	Oui

Altering Tables

As your application grows, you might need to alter your database, as part of a refactoring or adding new features. Not all database systems allow to modify existing columns or add columns between two existing ones. *Phalcon\Db* is limited by these constraints.

```
<?php

use Phalcon\Db\Column as Column;

// Adding a new column
$connection->addColumn(
    "robots",
    null,
    new Column(
        "robot_type",
        [
            "type"    => Column::TYPE_VARCHAR,
            "size"    => 32,
            "notNull" => true,
            "after"   => "name",
        ]
    )
);

// Modifying an existing column
$connection->modifyColumn(
    "robots",
    null,
    new Column(
        "name",
        [
            "type"    => Column::TYPE_VARCHAR,
            "size"    => 40,
            "notNull" => true,
        ]
    )
);

// Deleting the column "name"
$connection->dropColumn(
    "robots",
    null,
    "name"
);
```

Dropping Tables

Examples on dropping tables:

```
<?php

// Drop table robot from active database
$connection->dropTable("robots");

// Drop table robot from database "machines"
```

```
$connection->dropTable("robots", "machines");
```

2.3.49 Internationalization

Phalcon is written in C as an extension for PHP. There is a [PECL](#) extension that offers internationalization functions to PHP applications called [intl](#). Starting from PHP 5.4/5.5 this extension is bundled with PHP. Its documentation can be found in the pages of the official [PHP manual](#).

Phalcon does not offer this functionality, since creating such a component would be replicating existing code.

In the examples below, we will show you how to implement the [intl](#) extension's functionality into Phalcon powered applications.

This guide is not intended to be a complete documentation of the [intl](#) extension. Please visit its the [documentation](#) of the extension for a reference.

Find out best available Locale

There are several ways to find out the best available locale using [intl](#). One of them is to check the HTTP “Accept-Language” header:

```
<?php

$locale = Locale::acceptFromHttp($_SERVER["HTTP_ACCEPT_LANGUAGE"]);

// Locale could be something like "en_GB" or "en"
echo $locale;
```

Below method returns a locale identified. It is used to get language, culture, or regionally-specific behavior from the Locale API.

Examples of identifiers include:

- en-US (English, United States)
- ru-RU (Russian, Russia)
- zh-Hant-TW (Chinese, Traditional Script, Taiwan)
- fr-CA, fr-FR (French for Canada and France respectively)

Formatting messages based on Locale

Part of creating a localized application is to produce concatenated, language-neutral messages. The [MessageFormatter](#) allows for the production of those messages.

Printing numbers formatted based on some locale:

```
<?php

// Prints € 4 560
$formatter = new MessageFormatter("fr_FR", "€ {0, number, integer}");
echo $formatter->format([4560]);

// Prints USD$ 4,560.5
$formatter = new MessageFormatter("en_US", "USD$ {0, number}");
echo $formatter->format([4560.50]);
```

```
// Prints ARS$ 1.250,25
$formatter = new MessageFormatter("es_AR", "ARS$ {0, number}");
echo $formatter->format([1250.25]);
```

Message formatting using time and date patterns:

```
<?php

// Setting parameters
$time    = time();
$values  = [7, $time, $time];

// Prints "At 3:50:31 PM on Apr 19, 2015, there was a disturbance on planet 7."
$pattern = "At {1, time} on {1, date}, there was a disturbance on planet {0, number}
↪.";
$formatter = new MessageFormatter("en_US", $pattern);
echo $formatter->format($values);

// Prints "À 15:53:01 le 19 avr. 2015, il y avait une perturbation sur la planète 7."
$pattern  = "À {1, time} le {1, date}, il y avait une perturbation sur la planète {0,
↪ number}.";
$formatter = new MessageFormatter("fr_FR", $pattern);
echo $formatter->format($values);
```

Locale-Sensitive comparison

The `Collator` class provides string comparison capability with support for appropriate locale-sensitive sort orderings. Check the examples below on the usage of this class:

```
<?php

// Create a collator using Spanish locale
$collator = new Collator("es");

// Returns that the strings are equal, in spite of the emphasis on the "o"
$collator->setStrength(Collator::PRIMARY);
var_dump($collator->compare("una canción", "una cancion"));

// Returns that the strings are not equal
$collator->setStrength(Collator::DEFAULT_VALUE);
var_dump($collator->compare("una canción", "una cancion"));
```

Transliteration

Transliterators provides transliteration of strings:

```
<?php

$id = "Any-Latin; NFC; [:Nonspacing Mark:] Remove; NFC; [:Punctuation:] Remove; ↪
↪ Lower()";
$transliterator = Transliterator::create($id);

$string = "garçon-étudiant-où-L'école";
echo $transliterator->transliterate($string); // garconetudiantoulecole
```

2.3.50 Database Migrations

Migrations are a convenient way for you to alter your database in a structured and organized manner.

Important: Migrations are available in *Phalcon Developer Tools*. You need at least Phalcon Framework version 0.5.0 to use developer tools. Also, it is recommended to have PHP 5.4 or greater installed.

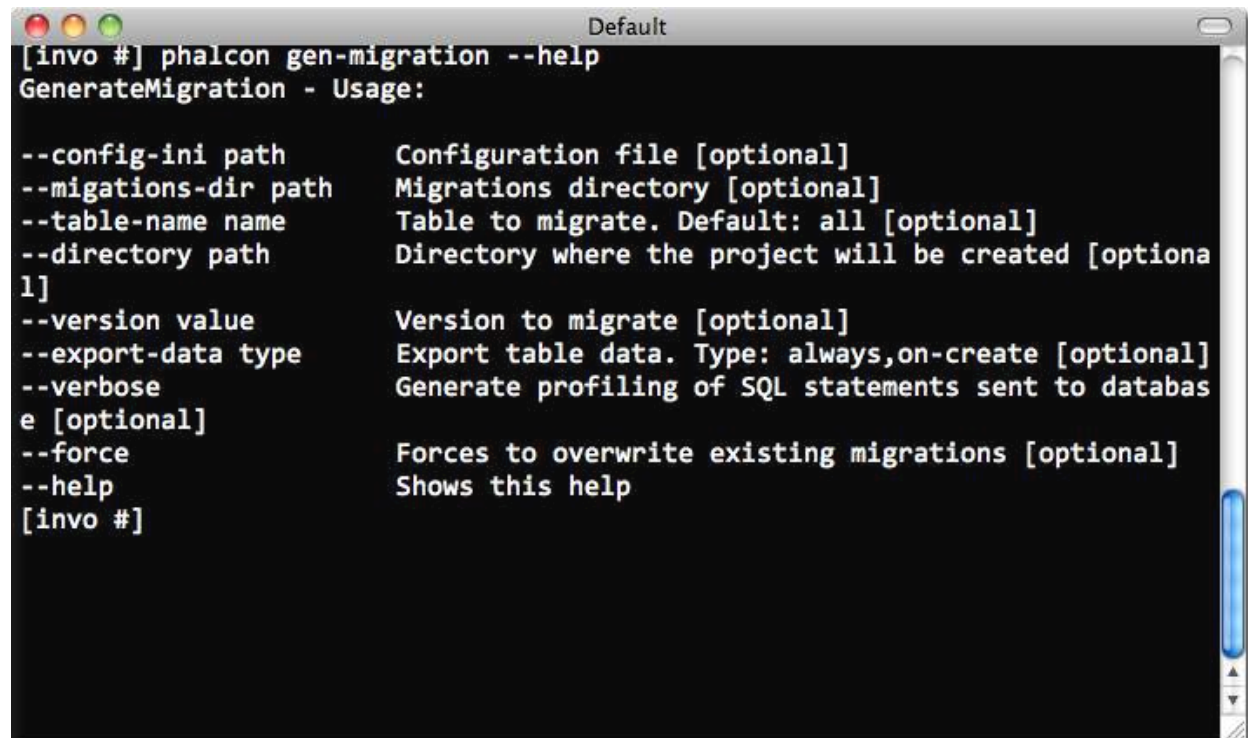
Often in development we need to update changes in production environments. Some of these changes could be database modifications like new fields, new tables, removing indexes, etc.

When a migration is generated a set of classes are created to describe how your database is structured at that particular moment. These classes can be used to synchronize the schema structure on remote databases setting your database ready to work with the new changes that your application implements. Migrations describe these transformations using plain PHP.

Schema Dumping

The *Phalcon Developer Tools* provides scripts to manage migrations (generation, running and rollback).

The available options for generating migrations are:

A screenshot of a terminal window titled "Default". The prompt is "[invo #]". The command entered is "phalcon gen-migration --help". The output shows the usage for "GenerateMigration" and lists several options: --config-ini path, --migrations-dir path, --table-name name, --directory path, --version value, --export-data type, --verbose, --force, and --help. Each option is followed by a description of its function.

```
[invo #] phalcon gen-migration --help
GenerateMigration - Usage:

--config-ini path      Configuration file [optional]
--migrations-dir path  Migrations directory [optional]
--table-name name      Table to migrate. Default: all [optional]
--directory path       Directory where the project will be created [optional]
--version value        Version to migrate [optional]
--export-data type     Export table data. Type: always,on-create [optional]
--verbose              Generate profiling of SQL statements sent to database [optional]
--force                Forces to overwrite existing migrations [optional]
--help                Shows this help
[invo #]
```

Running this script without any parameters will simply dump every object (tables and views) from your database into migration classes.

Each migration has a version identifier associated with it. The version number allows us to identify if the migration is newer or older than the current 'version' of our database. Versions will also inform Phalcon of the running order when executing a migration.

When a migration is generated, instructions are displayed on the console to describe the different steps of the migration and the execution time of those statements. At the end, a migration version is generated.


```

T, TABLES.ENGINE, TABLES.TABLE_COLLATION FROM INFORMATION_SCHEMA
.TABLES WHERE TABLES.TABLE_SCHEMA = "invo" AND TABLES.TABLE_NAME = "test" => 1335920161.9368 (0.00032401084899902)
1335920161.9369: DESCRIBE `invo`.`users` => 1335920161.938 (0.0010280609130859)
1335920161.9383: SHOW INDEXES FROM `invo`.`users` => 1335920161.9389 (0.00062680244445801)
1335920161.939: SELECT TABLE_NAME, COLUMN_NAME, CONSTRAINT_NAME, REFERENCED_TABLE_SCHEMA, REFERENCED_TABLE_NAME, REFERENCED_COLUMN_NAME FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE WHERE REFERENCED_TABLE_NAME IS NOT NULL AND CONSTRAINT_SCHEMA = "invo" AND TABLE_NAME = "users" => 1335920161.9403 (0.0012459754943848)
1335920161.9404: SELECT TABLES.TABLE_TYPE, TABLES.AUTO_INCREMENT, TABLES.ENGINE, TABLES.TABLE_COLLATION FROM INFORMATION_SCHEMA.TABLES WHERE TABLES.TABLE_SCHEMA = "invo" AND TABLES.TABLE_NAME = "users" => 1335920161.9407 (0.00038504600524902)
Version 1.0.0 was successfully generated
[invo #]

```

By default *Phalcon Developer Tools* uses the *app/migrations* directory to dump the migration files. You can change the location by setting one of the parameters on the generation script. Each table in the database has its respective class generated in a separated file under a directory referring its version:

Migration Class Anatomy

Each file contains a unique class that extends the `Phalcon\Mvc\Model\Migration` class. These classes normally have two methods: `up()` and `down()`. `up()` performs the migration, while `down()` rolls it back.

`up()` also contains the *magic* method `morphTable()`. The magic comes when it recognizes the changes needed to synchronize the actual table in the database to the description given.

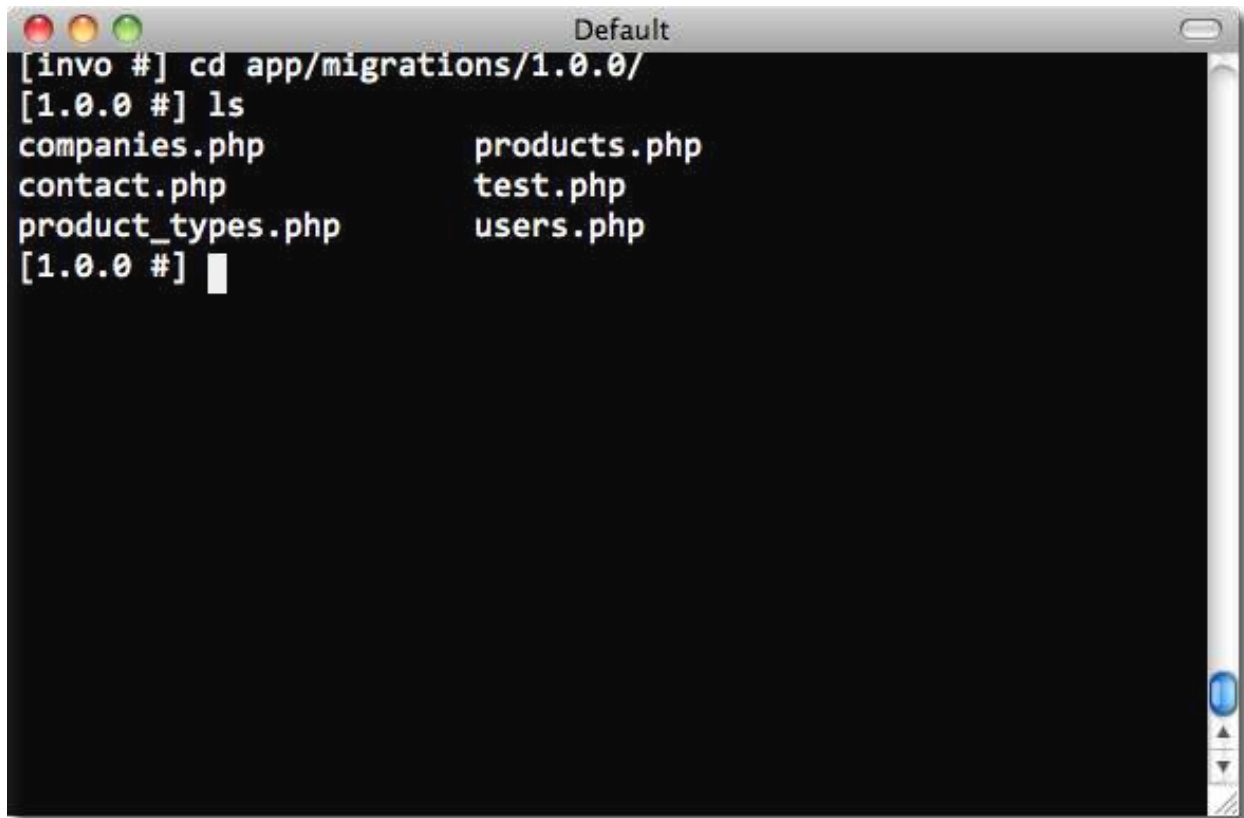
```

<?php

use Phalcon\Db\Column as Column;
use Phalcon\Db\Index as Index;
use Phalcon\Db\Reference as Reference;
use Phalcon\Mvc\Model\Migration;

class ProductsMigration_100 extends Migration
{
    public function up()
    {
        $this->morphTable(
            "products",
            [
                "columns" => [

```

A terminal window titled "Default" with a dark background and light-colored text. It shows a sequence of commands and their output. The first command is `[invo #] cd app/migrations/1.0.0/`. The second command is `[1.0.0 #] ls`, which outputs a list of files: `companies.php`, `contact.php`, `product_types.php`, `products.php`, `test.php`, and `users.php`. The prompt `[1.0.0 #]` is followed by a cursor.

```
[invo #] cd app/migrations/1.0.0/
[1.0.0 #] ls
companies.php      products.php
contact.php        test.php
product_types.php  users.php
[1.0.0 #]
```

```
new Column(
    "id",
    [
        "type"           => Column::TYPE_INTEGER,
        "size"           => 10,
        "unsigned"        => true,
        "notNull"         => true,
        "autoIncrement"   => true,
        "first"           => true,
    ]
),
new Column(
    "product_types_id",
    [
        "type"           => Column::TYPE_INTEGER,
        "size"           => 10,
        "unsigned"        => true,
        "notNull"         => true,
        "after"           => "id",
    ]
),
new Column(
    "name",
    [
        "type"           => Column::TYPE_VARCHAR,
        "size"           => 70,
        "notNull"         => true,
        "after"           => "product_types_id",
```

```

        ]
    ),
    new Column(
        "price",
        [
            "type"      => Column::TYPE_DECIMAL,
            "size"       => 16,
            "scale"      => 2,
            "notNull"    => true,
            "after"      => "name",
        ]
    ),
],
"indexes" => [
    new Index(
        "PRIMARY",
        [
            "id",
        ]
    ),
    new Index(
        "product_types_id",
        [
            "product_types_id",
        ],
    ),
],
"references" => [
    new Reference(
        "products_ibfk_1",
        [
            "referencedSchema" => "invo",
            "referencedTable"  => "product_types",
            "columns"           => ["product_types_id"],
            "referencedColumns" => ["id"],
        ]
    ),
],
"options" => [
    "TABLE_TYPE"      => "BASE TABLE",
    "ENGINE"           => "InnoDB",
    "TABLE_COLLATION" => "utf8_general_ci",
],
];
}
}

```

The class is called “ProductsMigration_100”. Suffix 100 refers to the version 1.0.0. `morphTable()` receives an associative array with 4 possible sections:

Index	Description	Optional
“columns”	An array with a set of table columns	Non
“indexes”	An array with a set of table indexes.	Oui
“references”	An array with a set of table references (foreign keys).	Oui
“options”	An array with a set of table creation options. These options are often related to the database system in which the migration was generated.	Oui

Defining Columns

Phalcon\Db\Column is used to define table columns. It encapsulates a wide variety of column related features. Its constructor receives as first parameter the column name and an array describing the column. The following options are available when describing columns:

Option	Description	Optional
“type”	Column type. Must be a <i>Phalcon\Db\Column</i> constant (see below)	Non
“size”	Some type of columns like VARCHAR or INTEGER may have a specific size	Oui
“scale”	DECIMAL or NUMBER columns may have a scale to specify how much decimals it must store	Oui
“unsigned”	INTEGER columns may be signed or unsigned. This option does not apply to other types of columns	Oui
“notNull”	Column can store null values?	Oui
“default”	Defines a default value for a column (can only be an actual value, not a function such as <i>NOW()</i>)	Oui
“autoIncrement”	With this attribute column will filled automatically with an auto-increment integer. Only one column in the table can have this attribute.	Oui
“first”	Column must be placed at first position in the column order	Oui
“after”	Column must be placed after indicated column	Oui

Database migrations support the following database column types:

- `Phalcon\Db\Column::TYPE_INTEGER`
- `Phalcon\Db\Column::TYPE_VARCHAR`
- `Phalcon\Db\Column::TYPE_CHAR`
- `Phalcon\Db\Column::TYPE_DATE`
- `Phalcon\Db\Column::TYPE_DATETIME`
- `Phalcon\Db\Column::TYPE_TIMESTAMP`
- `Phalcon\Db\Column::TYPE_DECIMAL`
- `Phalcon\Db\Column::TYPE_TEXT`
- `Phalcon\Db\Column::TYPE_BOOLEAN`
- `Phalcon\Db\Column::TYPE_FLOAT`
- `Phalcon\Db\Column::TYPE_DOUBLE`
- `Phalcon\Db\Column::TYPE_TINYBLOB`
- `Phalcon\Db\Column::TYPE_BLOB`

- `Phalcon\Db\Column::TYPE_MEDIUMBLOB`
- `Phalcon\Db\Column::TYPE_LONGBLOB`
- `Phalcon\Db\Column::TYPE_JSON`
- `Phalcon\Db\Column::TYPE_JSONB`
- `Phalcon\Db\Column::TYPE_BIGINTEGER`

Defining Indexes

Phalcon\Db\Index defines table indexes. An index only requires that you define a name for it and a list of its columns. Note that if any index has the name PRIMARY, Phalcon will create a primary key index for that table.

Defining References

Phalcon\Db\Reference defines table references (also called foreign keys). The following options can be used to define a reference:

Index	Description	Optional	Implemented in
“referencedTable”	It’s auto-descriptive. It refers to the name of the referenced table.	Non	All
“columns”	An array with the name of the columns at the table that have the reference	Non	All
“referenced-Columns”	An array with the name of the columns at the referenced table	Non	All
“referenced-Schema”	The referenced table maybe is on another schema or database. This option allows you to define that.	Oui	All
“onDelete”	If the foreign record is removed, perform this action on the local record(s).	Oui	MySQL PostgreSQL
“onUpdate”	If the foreign record is updated, perform this action on the local record(s).	Oui	MySQL PostgreSQL

Writing Migrations

Migrations aren’t only designed to “morph” table. A migration is just a regular PHP class so you’re not limited to these functions. For example after adding a column you could write code to set the value of that column for existing records. For more details and examples of individual methods, check the [database component](#).

```
<?php

use Phalcon\Mvc\Model\Migration;

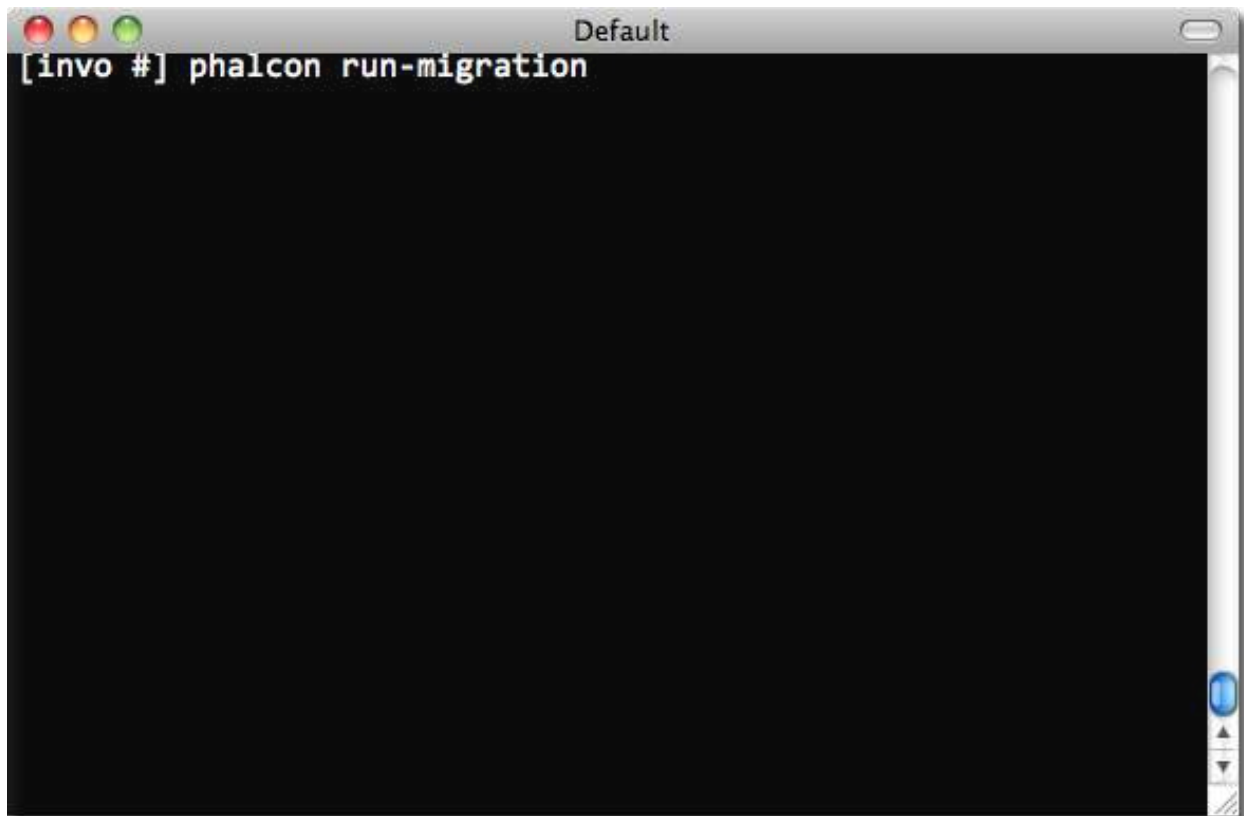
class ProductsMigration_100 extends Migration
{
    public function up()
    {
        // ...

        self::$_connection->insert(
            "products",
            [
                "Malabar spinach",
```

```
        14.50,  
    ],  
    [  
        "name",  
        "price",  
    ]  
    );  
}  
}
```

Running Migrations

Once the generated migrations are uploaded on the target server, you can easily run them as shown in the following example:



Depending on how outdated is the database with respect to migrations, Phalcon may run multiple migration versions in the same migration process. If you specify a target version, Phalcon will run the required migrations until it reaches the specified version.

2.3.51 Debugging Applications

PHP offers tools to debug applications with notices, warnings, errors and exceptions. The `Exception` class offers information such as the file, line, message, numeric code, backtrace etc. on where an error occurred. OOP frameworks like Phalcon mainly use this class to encapsulate this functionality and provide information back to the developer or user.

```

657)
1335932029.8233: SHOW INDEXES FROM `invo`.`products` => 13359
32029.8237 (0.00034308433532715)
1335932029.8246: SELECT COUNT(*) FROM `INFORMATION_SCHEMA`.`TA
BLES` WHERE `TABLE_NAME`='test' AND `TABLE_SCHEMA`='invo' =>
1335932029.8249 (0.00030779838562012)
1335932029.825: DESCRIBE `invo`.`test` => 1335932029.8259 (0.
00094413757324219)
1335932029.826: SHOW INDEXES FROM `invo`.`test` => 1335932029
.8269 (0.00085902214050293)
1335932029.8275: SELECT COUNT(*) FROM `INFORMATION_SCHEMA`.`TA
BLES` WHERE `TABLE_NAME`='users' AND `TABLE_SCHEMA`='invo' =
> 1335932029.8278 (0.00026392936706543)
1335932029.8279: DESCRIBE `invo`.`users` => 1335932029.8287 (
0.00088977813720703)
1335932029.8289: SHOW INDEXES FROM `invo`.`users` => 13359320
29.8299 (0.00097990036010742)
Version 1.0.0 was successfully migrated
[invo #]

```

MacGDBp @ 9000

eval \$i Attached

Variable	Value	Type	#	File	Line	Function
\$this	(\$this => (_depend	ProductsControll...	0	...oductsController.php	17	ProductsControll...
\$_COOKIE	(\$_COOKIE => (_u	array	1	...vo/public/index.php	0	Phalcon\Mvc\Ap...
\$_ENV	(\$_ENV => (DYLD_	array	2	...vo/public/index.php	121	{main}
\$_FILES		array				
\$_GET	(\$_GET => (_url =	array				
\$_POST		array				
\$_REQUEST	(\$_REQUEST => (_u	array				
\$_SERVER	(\$_SERVER => (RED	array				
\$_SESSION	(\$_SESSION => (Sec	array				
\$GLOBALS	(\$GLOBALS => (GL	array				

```

10 $this->view->setTemplateAfter('main');
11 Tag::setTitle('Manage your product types');
12 parent::initialize();
13 }
14
15 public function indexAction()
16 {
17     $this->persistent->searchParams = null;
18     $this->view->setVar("productTypes", ProductTypes::find());
19 }
20
21 public function searchAction()
22 {
23     $numberPage = 1;
24     if ($this->request->isPost()) {

```

Break

Despite being written in C, Phalcon executes methods in the PHP userland, providing the debug capability that any other application or framework written in PHP has.

Catching Exceptions

Throughout the tutorials and examples of the Phalcon documentation, there is a common element that is catching exceptions. This is a try/catch block:

```
<?php

try {

    // ... Some Phalcon/PHP code

} catch (\Exception $e) {

}
```

Any exception thrown within the block is captured in the variable `$e`. A *Phalcon\Exception* extends the PHP *Exception* class and is used to understand whether the exception came from Phalcon or PHP itself.

All exceptions generated by PHP are based on the *Exception* class, and have at least the following elements:

```
<?php

class Exception
{

    /* Properties */
    protected string $message;
    protected int $code;
    protected string $file;
    protected int $line;

    /* Methods */
    public __construct ([ string $message = "" [, int $code = 0 [, Exception
↪$previous = NULL ]]] )
    final public string getMessage ( void )
    final public Exception getPrevious ( void )
    final public mixed getCode ( void )
    final public string getFile ( void )
    final public int getLine ( void )
    final public array getTrace ( void )
    final public string getTraceAsString ( void )
    public string __toString ( void )
    final private void __clone ( void )
}
```

Retrieving information from *Phalcon\Exception* is the same as PHP's *Exception* class:

```
<?php

try {

    // ... App code ...

} catch (\Exception $e) {
    echo get_class($e), ": ", $e->getMessage(), "\n";
}
```



```

echo " File=", $e->getFile(), "\n";
echo " Line=", $e->getLine(), "\n";
echo $e->getTraceAsString();
}

```

It's therefore easy to find which file and line of the application's code generated the exception, as well as the components involved in generating the exception:

```

PDOException: SQLSTATE[28000] [1045] Access denied for user 'root'@'localhost'
  (using password: NO)
File=/Applications/MAMP/htdocs/invo/public/index.php
Line=74
#0 [internal function]: PDO->__construct('mysql:host=loca...', 'root', '', Array)
#1 [internal function]: Phalcon\Db\Adapter\Pdo->connect(Array)
#2 /Applications/MAMP/htdocs/invo/public/index.php(74):
  Phalcon\Db\Adapter\Pdo->__construct(Array)
#3 [internal function]: {closure}()
#4 [internal function]: call_user_func_array(Object(Closure), Array)
#5 [internal function]: Phalcon\Di->_factory(Object(Closure), Array)
#6 [internal function]: Phalcon\Di->get('db', Array)
#7 [internal function]: Phalcon\Di->getShared('db')
#8 [internal function]: Phalcon\Mvc\Model->getConnection()
#9 [internal function]: Phalcon\Mvc\Model::_getOrCreateResultset('Users', Array, true)
#10 /Applications/MAMP/htdocs/invo/app/controllers/SessionController.php(83):
  Phalcon\Mvc\Model::findFirst('email='demo@pha...')
#11 [internal function]: SessionController->startAction()
#12 [internal function]: call_user_func_array(Array, Array)
#13 [internal function]: Phalcon\Mvc\Dispatcher->dispatch()
#14 /Applications/MAMP/htdocs/invo/public/index.php(114): Phalcon\Mvc\Application->
  ->handle()
#15 {main}

```

As you can see from the above output the Phalcon's classes and methods are displayed just like any other component, and even showing the parameters that were invoked in every call. The method `Exception::getTrace` provides additional information if needed.

Debug component

Phalcon provides a debug component that allows the developer to easily find errors produced in an application created with the framework.

The following screencast explains how it works:

To enable it, add the following to your bootstrap:

```

<?php

$debug = new \Phalcon\Debug();
$debug->listen();

```

Any Try/Catch blocks must be removed or disabled to make this component work properly.

Reflection and Introspection

Any instance of a Phalcon class offers exactly the same behavior than a PHP normal one. It's possible to use the [Reflection API](#) or simply print any object to show how is its internal state:

```
<?php

$router = new Phalcon\Mvc\Router();
print_r($router);
```

It's easy to know the internal state of any object. The above example prints the following:

```
Phalcon\Mvc\Router Object
(
    [_dependencyInjector:protected] =>
    [_module:protected] =>
    [_controller:protected] =>
    [_action:protected] =>
    [_params:protected] => Array
        (
        )
    [_routes:protected] => Array
        (
            [0] => Phalcon\Mvc\Router\Route Object
                (
                    [_pattern:protected] => #^/([a-zA-Z0-9\_]+)/{0,1}$#
                    [_compiledPattern:protected] => #^/([a-zA-Z0-9\_]+)/{0,1}$#
                    [_paths:protected] => Array
                        (
                            [controller] => 1
                        )
                    [_methods:protected] =>
                    [_id:protected] => 0
                    [_name:protected] =>

                )
            [1] => Phalcon\Mvc\Router\Route Object
                (
                    [_pattern:protected] => #^/([a-zA-Z0-9\_]+)/([a-zA-Z0-9\_]+)/(.*)*
↪ $#
                    [_compiledPattern:protected] => #^/([a-zA-Z0-9\_]+)/([a-zA-Z0-9\_
↪ ]+)(/.*)*$#
                    [_paths:protected] => Array
                        (
                            [controller] => 1
                            [action] => 2
                            [params] => 3
                        )
                    [_methods:protected] =>
                    [_id:protected] => 1
                    [_name:protected] =>

                )
        )
    [_matchedRoute:protected] =>
    [_matches:protected] =>
    [_wasMatched:protected] =>
    [_defaultModule:protected] =>
    [_defaultController:protected] =>
    [_defaultAction:protected] =>
    [_defaultParams:protected] => Array
        (
        )
)
```

```
)
```

Using XDebug

XDebug is an amazing tool that complements the debugging of PHP applications. It is also a C extension for PHP, and you can use it together with Phalcon without additional configuration or side effects.

The following screencast shows a Xdebug session with Phalcon:

Once you have xdebug installed, you can use its API to get a more detailed information about exceptions and messages.

We highly recommend use at least XDebug 2.2.3 for a better compatibility with Phalcon

The following example implements `xdebug_print_function_stack` to stop the execution and generate a backtrace:

```
<?php
use Phalcon\Mvc\Controller;

class SignupController extends Controller
{
    public function indexAction()
    {

    }

    public function registerAction()
    {
        // Request variables from HTML form
        $name = $this->request->getPost("name", "string");
        $email = $this->request->getPost("email", "email");

        // Stop execution and show a backtrace
        return xdebug_print_function_stack("stop here!");

        $user = new Users();
        $user->name = $name;
        $user->email = $email;

        // Store and check for errors
        $user->save();
    }
}
```

In this instance, Xdebug will also show us the variables in the local scope, and a backtrace as well:

```
Xdebug: stop here! in /Applications/MAMP/htdocs/tutorial/app/controllers/
↳ SignupController.php
   on line 19

Call Stack:
 0.0383    654600    1. {main}() /Applications/MAMP/htdocs/tutorial/public/index.
↳ php:0
 0.0392    663864    2. Phalcon\Mvc\Application->handle()
   /Applications/MAMP/htdocs/tutorial/public/index.php:37
 0.0418    738848    3. SignupController->registerAction()
   /Applications/MAMP/htdocs/tutorial/public/index.php:0
```

```
0.0419      740144      4. xdebug_print_function_stack()
               /Applications/MAMP/htdocs/tutorial/app/controllers/SignupController.php:19
```

Xdebug provides several ways to get debug and trace information regarding the execution of your application using Phalcon. You can check the [XDebug documentation](#) for more information.

2.3.52 Unit testing

Writing proper tests can assist in writing better software. If you set up proper test cases you can eliminate most functional bugs and better maintain your software.

Integrating PHPunit with phalcon

If you don't already have phpunit installed, you can do it by using the following composer command:

```
composer require phpunit/phpunit:^5.0
```

or by manually adding it to composer.json:

```
{
    "require-dev": {
        "phpunit/phpunit": "^5.0"
    }
}
```

Once phpunit is installed create a directory called 'tests' in your root directory:

```
app/
public/
tests/
```

Next, we need a 'helper' file to bootstrap the application for unit testing.

The PHPunit helper file

A helper file is required to bootstrap the application for running the tests. We have prepared a sample file. Put the file in your tests/ directory as TestHelper.php.

```
<?php

use Phalcon\Di;
use Phalcon\Di\FactoryDefault;
use Phalcon\Loader;

ini_set("display_errors", 1);
error_reporting(E_ALL);

define("ROOT_PATH", __DIR__);

set_include_path(
    ROOT_PATH . PATH_SEPARATOR . get_include_path()
);

// Required for phalcon/incubator
```

```

include __DIR__ . "../vendor/autoload.php";

// Use the application autoloader to autoload the classes
// Autoload the dependencies found in composer
$loader = new Loader();

$loader->registerDirs(
    [
        ROOT_PATH,
    ]
);

$loader->register();

$di = new FactoryDefault();

Di::reset();

// Add any needed services to the DI here

Di::setDefault($di);

```

Should you need to test any components from your own library, add them to the autoloader or use the autoloader from your main application.

To help you build the unit tests, we made a few abstract classes you can use to bootstrap the unit tests themselves. These files exist in the Phalcon incubator @ <https://github.com/phalcon/incubator>.

You can use the incubator library by adding it as a dependency:

```
composer require phalcon/incubator
```

or by manually adding it to composer.json:

```

{
    "require": {
        "phalcon/incubator": "^3.0"
    }
}

```

You can also clone the repository using the repo link above.

PHPunit.xml file

Now, create a phpunit file:

```

<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="./TestHelper.php"
    backupGlobals="false"
    backupStaticAttributes="false"
    verbose="true"
    colors="false"
    convertErrorsToExceptions="true"
    convertNoticesToExceptions="true"
    convertWarningsToExceptions="true"
    processIsolation="false"
    stopOnFailure="false"

```

```
        syntaxCheck="true">
        <testsuite name="Phalcon - Testsuite">
            <directory>./</directory>
        </testsuite>
    </phpunit>
```

Modify the phpunit.xml to fit your needs and save it in tests/.

This will run any tests under the tests/ directory.

Sample unit test

To run any unit tests you need to define them. The autoloader will make sure the proper files are loaded so all you need to do is create the files and phpunit will run the tests for you.

This example does not contain a config file, most test cases however, do need one. You can add it to the DI to get the UnitTestCase file.

First create a base unit test called UnitTestCase.php in your /tests directory:

```
<?php

use Phalcon\Di;
use Phalcon\Test\UnitTestCase as PhalconTestCase;

abstract class UnitTestCase extends PhalconTestCase
{
    /**
     * @var bool
     */
    private $_loaded = false;

    public function setUp()
    {
        parent::setUp();

        // Load any additional services that might be required during testing
        $di = Di::getDefault();

        // Get any DI components here. If you have a config, be sure to pass it to
        ↪ the parent

        $this->setDi($di);

        $this->_loaded = true;
    }

    /**
     * Check if the test case is setup properly
     *
     * @throws \PHPUnit_Framework_IncompleteTestError;
     */
    public function __destruct()
    {
        if (!$this->_loaded) {
            throw new \PHPUnit_Framework_IncompleteTestError(
```

```

        "Please run parent::setUp() ."
    );
}
}
}

```

It's always a good idea to separate your Unit tests in namespaces. For this test we will create the namespace 'Test'. So create a file called testsTestUnitTest.php:

```

<?php

namespace Test;

/**
 * Class UnitTest
 */
class UnitTest extends \UnitTestCase
{
    public function testTestCase()
    {
        $this->assertEquals(
            "works",
            "works",
            "This is OK"
        );

        $this->assertEquals(
            "works",
            "works1",
            "This will fail"
        );
    }
}

```

Now when you execute 'phpunit' in your command-line from the tests directory you will get the following output:

```

$ phpunit
PHPUnit 3.7.23 by Sebastian Bergmann.

Configuration read from /private/var/www/tests/phpunit.xml

Time: 3 ms, Memory: 3.25Mb

There was 1 failure:

1) Test\UnitTest::testTestCase
This will fail
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'works'
+'works1'

/private/var/www/tests/Test/UnitTest.php:25

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.

```

Now you can start building your unit tests. You can view a good guide here (we also recommend reading the PHPunit documentation if you're not familiar with PHPUnit):

<http://blog.stevensanderson.com/2009/08/24/writing-great-unit-tests-best-and-worst-practises/>

2.4 In Depth Explanations / Further Reading

Attention: Cette documentation est incomplète.

2.4.1 Améliorer les performances : C'est quoi la suite ?

Avoir des applications plus rapides nécessite l'amélioration de différents composants : Le serveur, le client, le réseau la base de données, le serveur web, les sources statiques, etc. Dans ce chapitre nous avons sélectionné des scénarios où l'on pourra améliorer les performances et où nous verrons comment voir ce qui ralentit notre application.

Profilage du Server

Chaque application est différente, le profilage (l'analyse) constante est importante pour comprendre où les performances peuvent être améliorés. Le profilage nous permet d'avoir une idée réelle de ce qui est lent et de ce qui ne l'est pas. Chaque analyse varie en fonction des requêtes, donc il est important de faire assez de mesures pour obtenir des conclusions efficace.

Profilage avec XDebug

Xdebug nous fournit un moyen simple d'analyser des applications PHP, il suffit d'installer l'extension et d'autoriser le profilage dans php.ini :

```
xdebug.profiler_enable = On
```

En utilisant un outils comme [Webgrind](#) on peut voir quelles fonctions/méthodes sont lentes par rapport aux autres :

Profilage avec Xhprof

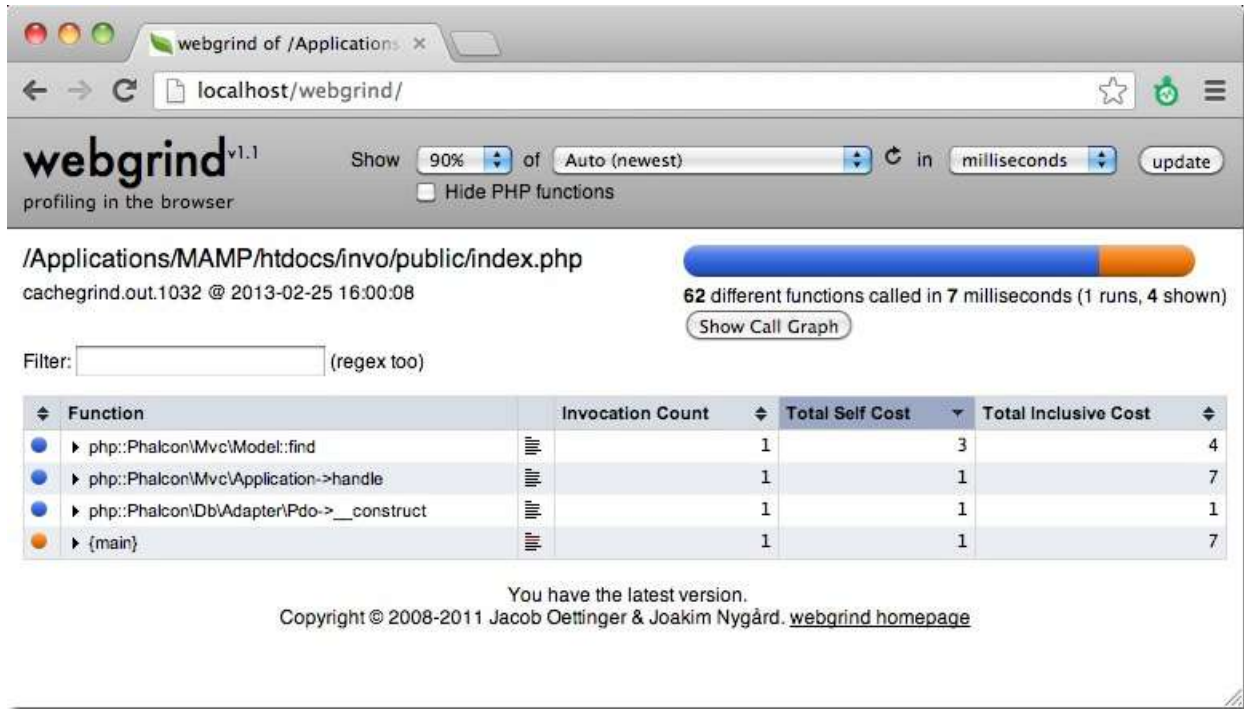
Xhprof est une autre extension intéressante pour l'analyse d'applications PHP. Ajoutez le code suivant au début du fichier index.php (le fichier bootstrap qui se trouve normalement dans public/) :

```
<?php
xhprof_enable(XHPROF_FLAGS_CPU + XHPROF_FLAGS_MEMORY);
```

Puis à la fin de ce même fichier, ajoutez ceci :

```
<?php
$хhprof_data = xhprof_disable('/tmp');
$XHPROF_ROOT = "/var/www/xhprof/";
include_once $XHPROF_ROOT . "/xhprof_lib/utils/xhprof_lib.php";
include_once $XHPROF_ROOT . "/xhprof_lib/utils/xhprof_runs.php";

$хhprof_runs = new XHProfRuns_Default();
```

```
$run_id = $xhprof_runs->save_run($xhprof_data, "xhprof_testing");

echo "http://localhost/xhprof/xhprof_html/index.php?run={$run_id}&source=xhprof_
    <testing\n";
```

Xhprof fournit un aperçu en HTML pour analyser les données récupérés :

Profilage des requête SQL

La plupart des bases de données fournissent des outils pour identifier les requêtes lourdes. Détecter et corriger ces requêtes est très important pour améliorer les performances du côté serveur. Dans le cas de MySQL, vous pouvez utiliser les “slow queries logs” (logs de requêtes lentes) pour savoir quelles requêtes prennent plus de temps que prévu :

```
log-slow-queries = /var/log/slow-queries.log
long_query_time = 1.5
```

Profilage côté Client

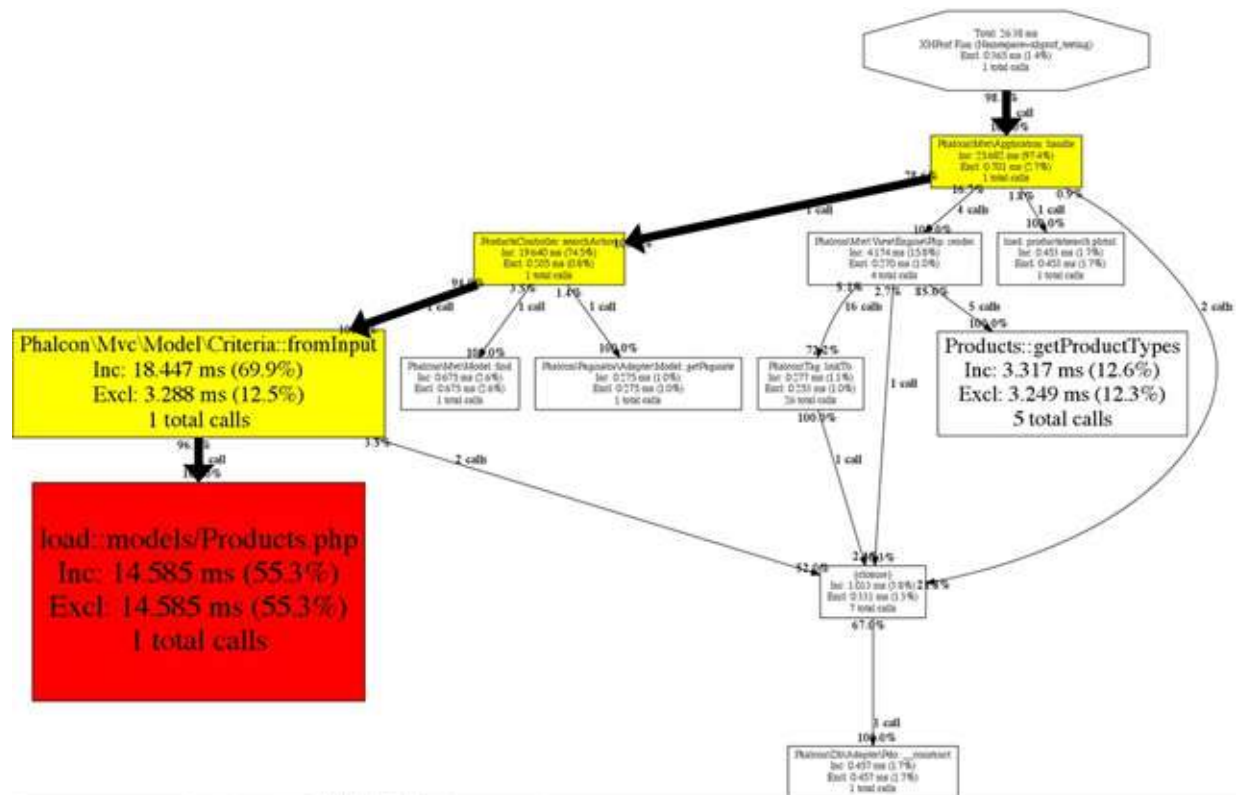
Des fois, on a besoin d’améliorer le chargement des éléments statiques comme des images, du javascript et du CSS pour améliorer les performances. Les outils suivants sont très utiles pour détecter les goulots d’étranglement du côté client :

Profilage avec Chrome/Firefox

La plupart des navigateurs modernes ont des outils pour profiler le chargement des pages. Dans chrome vous pouvez utiliser l’inspecteur d’élément pour savoir ce qui prends du temps à charger sur une page.

Displaying top 100 functions: Sorted by Excl. Wall Time (microsec) [\[display all\]](#)

Function Name	Calls	Calls%	Incl. Wall Time (microsec)	IWall%	Excl. Wall Time (microsec)	EWall%	Incl. CPU (microsecs)	ICpu%	Excl. CPU (microsec)	ECPU%	Incl. MemUse (bytes)
load::models/Products.php	1	0.8%	14,585	55.3%	14,585	55.3%	346	5.4%	346	5.4%	43,528
Phalcon\Mvc\Model\Criteria::fromInput	1	0.8%	18,447	69.9%	3,288	12.5%	1,515	23.7%	831	13.0%	87,544
Products::getProductTypes	5	3.8%	3,317	12.6%	3,249	12.3%	1,029	16.1%	958	15.0%	57,992
Phalcon\Mvc\Application::handle	1	0.8%	25,682	97.4%	701	2.7%	5,981	93.4%	683	10.7%	355,296
Phalcon\Mvc\Model::find	1	0.8%	675	2.6%	675	2.6%	457	7.1%	457	7.1%	16,288
Phalcon\Db\Adapter\Pdo::construct	1	0.8%	457	1.7%	457	1.7%	217	3.4%	217	3.4%	8,664
load::products/search.phhtml	1	0.8%	453	1.7%	453	1.7%	288	4.5%	288	4.5%	26,144
main()	1	0.8%	26,380	100.0%	365	1.4%	6,401	100.0%	128	2.0%	402,568
{closure}	7	5.4%	1,013	3.8%	331	1.3%	777	12.1%	308	4.8%	99,544
Phalcon\Paginator\Adapter\Model::getPaginate	1	0.8%	275	1.0%	275	1.0%	276	4.3%	276	4.3%	24,928
Phalcon\Mvc\View\Engine\Php::render	4	3.1%	4,174	15.8%	270	1.0%	1,864	29.1%	231	3.6%	99,664
Phalcon\Tag::linkTo	26	20.0%	277	1.1%	253	1.0%	292	4.6%	268	4.2%	9,024
ProductsController::searchAction	1	0.8%	19,640	74.5%	205	0.8%	2,479	38.7%	184	2.9%	142,392
Phalcon\Loader::autoLoad	5	3.8%	174	0.7%	128	0.5%	122	1.9%	89	1.4%	14,032
Phalcon\Config\Adapter\Ini::construct	1	0.8%	122	0.5%	122	0.5%	123	1.9%	123	1.9%	4,480
Phalcon\Session\Adapter::start	1	0.8%	115	0.4%	115	0.4%	117	1.8%	117	1.8%	34,328
Phalcon\DI::set	8	6.2%	80	0.3%	80	0.3%	65	1.0%	65	1.0%	2,592
Elements::getMenu	1	0.8%	107	0.4%	59	0.2%	108	1.7%	52	0.8%	5,632
Security::beforeDispatch	1	0.8%	243	0.9%	52	0.2%	244	3.8%	46	0.7%	47,208
Phalcon\Loader::construct	1	0.8%	51	0.2%	51	0.2%	15	0.2%	15	0.2%	1,160
Elements::getTabs	1	0.8%	96	0.4%	51	0.2%	97	1.5%	45	0.7%	4,504
Phalcon\DI\FactoryDefault::construct	1	0.8%	47	0.2%	47	0.2%	49	0.8%	49	0.8%	17,736



Firebug fournit les mêmes fonctionnalités sous firefox :

Yahoo! YSlow

YSlow analyse les pages web et suggère des moyens d'améliorer les performances en fonction d'un ensemble de [règles pour des pages de hautes performances](#)

Profilage avec Speed Tracer

[Speed Tracer](#) is a tool to help you identify and fix performance problems in your web applications. It visualizes metrics that are taken from low level instrumentation points inside of the browser and analyzes them as your application runs. Speed Tracer is available as a Chrome extension and works on all platforms where extensions are currently supported (Windows and Linux).

Cet outil est très pratique parce qu'il permet d'avoir un vrai temps de chargement nécessaire pour l'affichage de la page complet (y compris le passage des éléments HTML, Javascript et CSS).

Utiliser une version récente de PHP

PHP est plus rapide chaque jour, en utilisant la dernière version, vous pourrez améliorer les performances de votre application et aussi de PHP.

Utiliser un cache PHP Bytecode

[APC](#), comme beaucoup d'autre cache bytecode, aide une application à réduire le temps de chargement des lectures, il segmente et parse les fichiers PHP pour chaque requêtes. Une fois l'extension installé, utilisez la ligne suivante pour le mettre en place :

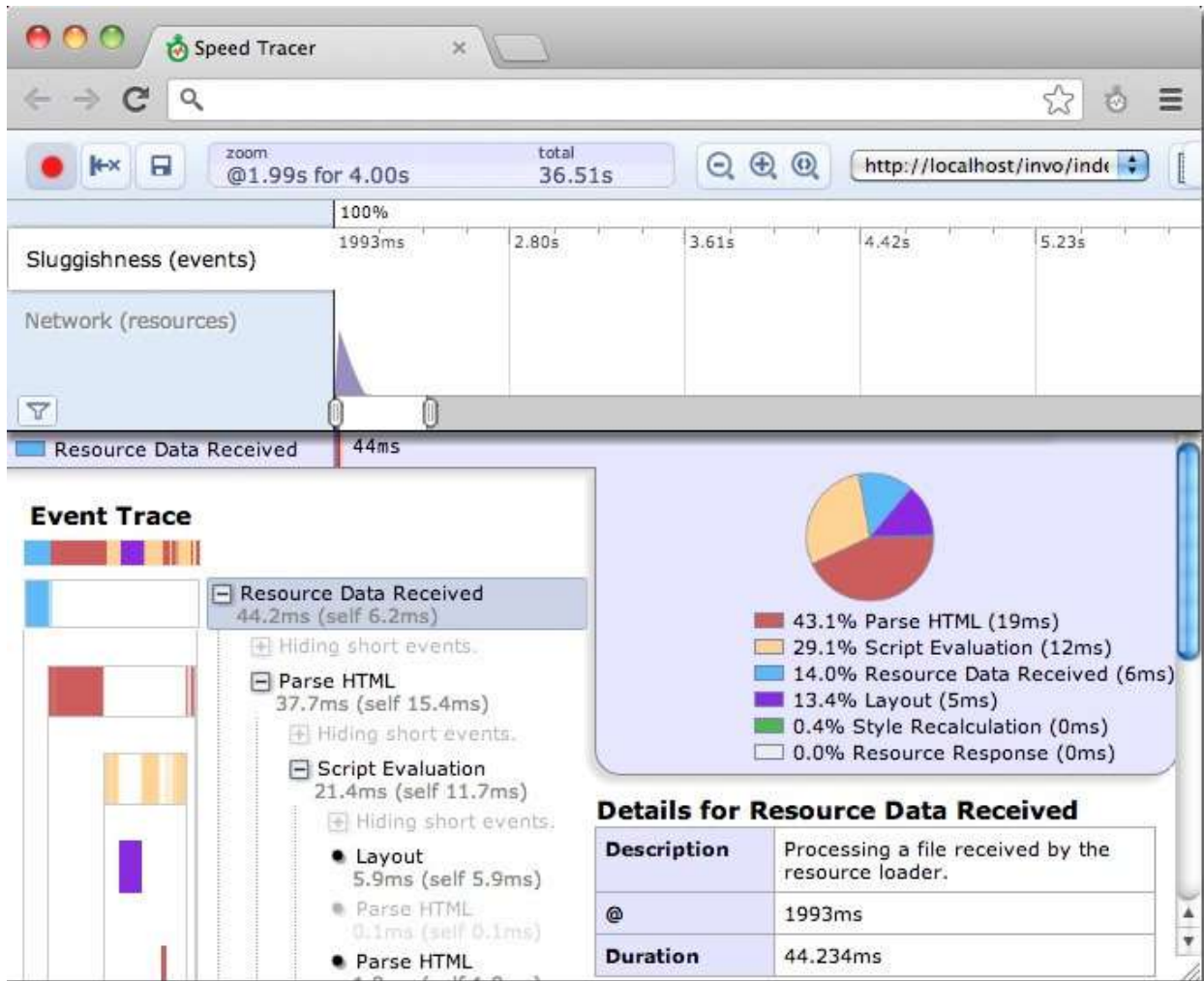
```
apc.enabled = On
```

PHP 5.5 inclus un cache bytecode intégré appelé ZendOptimizer+, cette extension est aussi disponible pour PHP 5.3 et 5.4.

Mettez le travail lent en tâche de fond

Traiter une vidéo, envoyer des emails, compresser un fichier ou une image sont des tâches lentes qui doivent être mises en tâche de fond. Voici une variété d'outils qui fournissent un système de mise en queue (effectuer les tâches les unes après les autres) ou un système de messages programme à programme qui fonctionne bien avec PHP :

- [Beanstalkd](#)
- [Redis](#)
- [RabbitMQ](#)
- [Resque](#)
- [Gearman](#)
- [ZeroMQ](#)



Google Page Speed

`mod_pagespeed` accélère votre site et réduit le temps de chargement des pages. Ce module apache open-source (aussi disponible pour nginx sous le nom `ngx_pagespeed`) met en place les meilleures pratique d'optimisation sur votre serveur, automatique. Il associe aussi les fichiers CSS, javascript et les images sans que vous n'ayez besoin de modifier le contenu de votre site.

2.4.2 La dépendance d'injection expliquée

L'exemple qui suit est un peu long, mais il tente d'expliquer pourquoi Phalcon utilise la localisation de service et l'injection de dépendance. Pour commencer, imaginons que nous avons développé un composant appelé `SomeComponent`. La tâche qu'il réalise n'a pas d'importance maintenant. Notre composant détient des dépendances dont la connexion à la base de données.

Dans ce premier exemple, la connexion est réalisée à l'intérieur du composant. Cette approche n'est pas pratique; nous ne pouvons pas changer les paramètres de la connexion ou le type de SGBD parce que le composant n'est créé que pour fonctionner comme ça.

```
<?php

class SomeComponent
{
    /**
     * L'instanciation de la connexion est codée en dur dans le
     * composant, ce qui fait qu'il est compliqué de le remplacer
     * extérieurement ou de modifier son comportement
     */
    public function someDbTask()
    {
        $connection = new Connection(
            [
                "host"      => "localhost",
                "username" => "root",
                "password" => "secret",
                "dbname"    => "invo",
            ]
        );

        // ...
    }
}

$some = new SomeComponent();

$some->someDbTask();
```

Pour résoudre ceci, nous avons créé un accesseur qui injecte une dépendance externe avant de l'utiliser. Pour l'instant, ceci semble être une bonne solution:

```
<?php

class SomeComponent
{
    protected $_connection;

    /**
     * Attribution d'une connexion externe
```



```

    */
    public function setConnection($connection)
    {
        $this->_connection = $connection;
    }

    public function someDbTask()
    {
        $connection = $this->_connection;

        // ...
    }
}

$some = new SomeComponent();

// Création de la connexion
$connection = new Connection(
    [
        "host"      => "localhost",
        "username"  => "root",
        "password"  => "secret",
        "dbname"    => "invo",
    ]
);

// Injection de la connexion dans le composant
$some->setConnection($connection);

$some->someDbTask();

```

Bon, considérons maintenant que nous utilisons ce composant dans différentes parties de l'application et que nous ayons besoin de créer la connexion plusieurs fois avant de la transmettre au composant.

Nous pouvons résoudre ceci en créant une sorte de registre global d'où nous obtenons l'instance de la connexion pour ne pas avoir à la recréer encore et encore:

```

<?php

class Registry
{
    /**
     * Retourne la connexion
     */
    public static function getConnection()
    {
        return new Connection(
            [
                "host"      => "localhost",
                "username"  => "root",
                "password"  => "secret",
                "dbname"    => "invo",
            ]
        );
    }
}

class SomeComponent

```

```
{  
    protected $_connection;  
  
    /**  
     * Attribution d'une connexion externe  
     */  
    public function setConnection($connection)  
    {  
        $this->_connection = $connection;  
    }  
  
    public function someDbTask()  
    {  
        $connection = $this->_connection;  
  
        // ...  
    }  
}  
  
$some = new SomeComponent();  
  
// Pass the connection defined in the registry  
$some->setConnection(Registry::getConnection());  
  
$some->someDbTask();
```

Maintenant, imaginons que nous devons réaliser deux méthodes dans ce composant, La première doit toujours créer une nouvelle connexion et la seconde doit utiliser une connexion partagée:

```
<?php  
  
class Registry  
{  
    protected static $_connection;  
  
    /**  
     * Création d'une connexion  
     */  
    protected static function _createConnection()  
    {  
        return new Connection(  
            [  
                "host"      => "localhost",  
                "username" => "root",  
                "password" => "secret",  
                "dbname"   => "invo",  
            ]  
        );  
    }  
  
    /**  
     * Création unique d'une connexion et la retourne  
     */  
    public static function getSharedConnection()  
    {  
        if (self::$_connection === null) {  
            self::$_connection = self::_createConnection();  
        }  
    }  
}
```

```

        return self::$_connection;
    }

    /**
     * Retourne toujours une nouvelle connexion
     */
    public static function getNewConnection()
    {
        return self::_createConnection();
    }
}

class SomeComponent
{
    protected $_connection;

    /**
     * Attribution d'une connexion externe
     */
    public function setConnection($connection)
    {
        $this->_connection = $connection;
    }

    /**
     * Cette méthode utilise toujours la connexion partagée
     */
    public function someDbTask()
    {
        $connection = $this->_connection;

        // ...
    }

    /**
     * Cette méthode utilise toujours une nouvelle connexion
     */
    public function someOtherDbTask($connection)
    {
    }
}

$some = new SomeComponent();

// Injection de la connexion partagée
$some->setConnection(
    Registry::getSharedConnection()
);

$some->someDbTask();

// Ici, nous passons toujours une nouvelle connexion en paramètre
$some->someOtherDbTask(
    Registry::getNewConnection()
);

```

Jusque là, nous avons vu comment l'injection de dépendance résout notre problème. Transmettre des dépendances en argument au lieu de les créer en interne dans le code rend notre application plus maintenable et découplée. Cependant, sur le long terme, cette forme de dépendance possède quelques inconvénients.

Par exemple, si le composant contient plusieurs dépendances, nous devons créer plusieurs mutateurs pour transmettre les dépendances ou créer un constructeur avec plusieurs arguments, créant ainsi systématiquement des dépendances avant d'utiliser le composant, rendant ainsi le code moins maintenable que nous ne le voudrions:

```
<?php

// Création de la dépendance ou récupération du registre
$connection = new Connection();
$session    = new Session();
$fileSystem = new FileSystem();
$filter     = new Filter();
$selector   = new Selector();

// Passage de paramètres au constructeur
$some = new SomeComponent($connection, $session, $fileSystem, $filter, $selector);

// ... ou avec des mutateurs
$some->setConnection($connection);
$some->setSession($session);
$some->setFileSystem($fileSystem);
$some->setFilter($filter);
$some->setSelector($selector);
```

Supposez que nous devons créer cet objet dans différentes parties de notre application. Si, dans le futur, nous n'avions plus besoin de ces dépendances, nous devrions naviguer au sein du code pour enlever le paramètre des constructeurs ou des accesseurs. Pour résoudre ceci, nous revenons au registre global pour créer le composant. Toutefois, on ajoute une nouvelle couche d'abstraction avant de créer l'objet:

```
<?php

class SomeComponent
{
    // ...

    /**
     * Définition d'une méthode de fabrication pour instancier SomeComponent
     * et lui injecter ses dépendances
     */
    public static function factory()
    {
        $connection = new Connection();
        $session    = new Session();
        $fileSystem = new FileSystem();
        $filter     = new Filter();
        $selector   = new Selector();

        return new self($connection, $session, $fileSystem, $filter, $selector);
    }
}
```

Maintenant, nous nous retrouvons à notre point de départ en ayant une fois de plus recréé les dépendances à l'intérieur du composant ! Nous devons trouver une solution pour éviter de reproduire ces mauvaises pratiques.

Une façon pratique et élégante de résoudre ces problèmes est d'exploiter un conteneur pour dépendances. Ces conteneur agissent comme le registre global que nous avons vus au préalable. L'utilisation d'un conteneur de dépendances

comme passerelle pour obtenir les dépendances nous permet de réduire la complexité de notre composant:

```
<?php

use Phalcon\Di;
use Phalcon\DiInterface;

class SomeComponent
{
    protected $_di;

    public function __construct(DiInterface $di)
    {
        $this->_di = $di;
    }

    public function someDbTask()
    {
        // Récupération du service de connexion
        // Retourne toujours une nouvelle connexion
        $connection = $this->_di->get("db");
    }

    public function someOtherDbTask()
    {
        // Récupération d'un service de connexion partagé
        // Retourne toujours la même connexion
        $connection = $this->_di->getShared("db");

        // Cette méthode nécessite également un filtre d'entrée
        $filter = $this->_di->get("filter");
    }
}

$di = new Di();

// Inscription d'un service "db" dans le conteneur
$di->set(
    "db",
    function () {
        return new Connection(
            [
                "host"      => "localhost",
                "username" => "root",
                "password" => "secret",
                "dbname"    => "invo",
            ]
        );
    }
);

// Inscription d'un service "filter" dans le conteneur
$di->set(
    "filter",
    function () {
        return new Filter();
    }
);
```

```
// Inscription d'un service "session" dans le conteneur
$di->set (
    "session",
    function () {
        return new Session();
    }
);

// Transmission du conteneur en un seul paramètre
$some = new SomeComponent($di);

$some->someDbTask();
```

Le composant peut maintenant accéder au service dont il n’a besoin que lorsque c’est nécessaire et s’il n’est pas requis il ne sera pas initialisé épargnant ainsi des ressources. Le composant est désormais fortement découplé. Par exemple nous pouvons remplacer la façon dont la connexion est créée, son comportement ou tout autre aspect n’affectera pas le composant.

2.4.3 Understanding How Phalcon Applications Work

If you’ve been following the [tutorial](#) or have generated the code using *Phalcon Devtools*, you may recognize the following bootstrap file:

```
<?php

use Phalcon\Mvc\Application;

// Register autoloaders
// ...

// Register services
// ...

// Handle the request
$application = new Application($di);

try {
    $response = $application->handle();

    $response->send();
} catch (\Exception $e) {
    echo "Exception: ", $e->getMessage();
}
```

The core of all the work of the controller occurs when `handle()` is invoked:

```
<?php

$response = $application->handle();
```

Manual bootstrapping

If you do not wish to use *Phalcon\Mvc\Application*, the code above can be changed as follows:

```

<?php

// Get the 'router' service
$router = $di["router"];

$router->handle();

$view = $di["view"];

$dispatcher = $di["dispatcher"];

// Pass the processed router parameters to the dispatcher
$dispatcher->setControllerName(
    $router->getControllerName()
);

$dispatcher->setActionName(
    $router->getActionName()
);

$dispatcher->setParams(
    $router->getParams()
);

// Start the view
$view->start();

// Dispatch the request
$dispatcher->dispatch();

// Render the related views
$view->render(
    $dispatcher->getControllerName(),
    $dispatcher->getActionName(),
    $dispatcher->getParams()
);

// Finish the view
$view->finish();

$response = $di["response"];

// Pass the output of the view to the response
$response->setContent(
    $view->getContent()
);

// Send the response
$response->send();

```

The following replacement of *Phalcon\Mvc\Application* lacks of a view component making it suitable for Rest APIs:

```

<?php

use Phalcon\Http\ResponseInterface;

// Get the 'router' service

```

```
$router = $di["router"];

$router->handle();

$dispatcher = $di["dispatcher"];

// Pass the processed router parameters to the dispatcher

$dispatcher->setControllerName(
    $router->getControllerName()
);

$dispatcher->setActionName(
    $router->getActionName()
);

$dispatcher->setParams(
    $router->getParams()
);

// Dispatch the request
$dispatcher->dispatch();

// Get the returned value by the last executed action
$response = $dispatcher->getReturnedValue();

// Check if the action returned is a 'response' object
if ($response instanceof ResponseInterface) {
    // Send the response
    $response->send();
}
```

Yet another alternative that catch exceptions produced in the dispatcher forwarding to other actions consequently:

```
<?php

use Phalcon\Http\ResponseInterface;

// Get the 'router' service
$router = $di["router"];

$router->handle();

$dispatcher = $di["dispatcher"];

// Pass the processed router parameters to the dispatcher

$dispatcher->setControllerName(
    $router->getControllerName()
);

$dispatcher->setActionName(
    $router->getActionName()
);

$dispatcher->setParams(
    $router->getParams()
);
```



```

try {
    // Dispatch the request
    $dispatcher->dispatch();
} catch (Exception $e) {
    // An exception has occurred, dispatch some controller/action aimed for that

    // Pass the processed router parameters to the dispatcher
    $dispatcher->setControllerName("errors");
    $dispatcher->setActionName("action503");

    // Dispatch the request
    $dispatcher->dispatch();
}

// Get the returned value by the last executed action
$response = $dispatcher->getReturnedValue();

// Check if the action returned is a 'response' object
if ($response instanceof ResponseInterface) {
    // Send the response
    $response->send();
}

```

Although the above implementations are a lot more verbose than the code needed while using *Phalcon\MvcApplication*, it offers an alternative in bootstrapping your application. Depending on your needs, you might want to have full control of what should be instantiated or not, or replace certain components with those of your own to extend the default functionality.

2.5 API

2.5.1 API Indice

Abstract class **Phalcon\Acl**

Constants

integer **ALLOW**

integer **DENY**

Abstract class **Phalcon\Acl\Adapter**

implements *Phalcon\Acl\AdapterInterface*, *Phalcon\Events\EventsAwareInterface*

Adapter for Phalcon\Acl adapters

Methods

public **getActiveRole** ()

Role which the list is checking if it's allowed to certain resource/access

public **getActiveResource** ()

Resource which the list is checking if some role can access it

public **getActiveAccess** ()

Active access which the list is checking if some role can access it

public **setEventsManager** (*Phalcon\Events\ManagerInterface* \$eventsManager)

Sets the events manager

public **getEventsManager** ()

Returns the internal event manager

public **setDefaultAction** (*mixed* \$defaultAccess)

Sets the default access level (Phalcon\Acl::ALLOW or Phalcon\Acl::DENY)

public **getDefaultAction** ()

Returns the default ACL access level

abstract public **setNoArgumentsDefaultAction** (*mixed* \$defaultAccess) inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **getNoArgumentsDefaultAction** () inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **addRole** (*mixed* \$role, [*mixed* \$accessInherits]) inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **addInherit** (*mixed* \$roleName, *mixed* \$roleToInherit) inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **isRole** (*mixed* \$roleName) inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **isResource** (*mixed* \$resourceName) inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **addResource** (*mixed* \$resourceObject, *mixed* \$accessList) inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **addResourceAccess** (*mixed* \$resourceName, *mixed* \$accessList) inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **dropResourceAccess** (*mixed* \$resourceName, *mixed* \$accessList) inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **allow** (*mixed* \$roleName, *mixed* \$resourceName, *mixed* \$access, [*mixed* \$func]) inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **deny** (*mixed* \$roleName, *mixed* \$resourceName, *mixed* \$access, [*mixed* \$func]) inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **isAllowed** (*mixed* \$roleName, *mixed* \$resourceName, *mixed* \$access, [*array* \$parameters]) inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **getRoles** () inherited from *Phalcon\Acl\AdapterInterface*

...

abstract public **getResources** () inherited from *Phalcon\Acl\AdapterInterface*

...

Class Phalcon\Acl\Adapter\Memory

extends abstract class *Phalcon\Acl\Adapter*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\Acl\AdapterInterface*

Manages ACL lists in memory

```
<?php

$acl = new \Phalcon\Acl\Adapter\Memory();

$acl->setDefaultAction(
    \Phalcon\Acl::DENY
);

// Register roles
$roles = [
    "users" => new \Phalcon\Acl\Role("Users"),
    "guests" => new \Phalcon\Acl\Role("Guests"),
];
foreach ($roles as $role) {
    $acl->addRole($role);
}

// Private area resources
$privateResources = [
    "companies" => ["index", "search", "new", "edit", "save", "create", "delete"],
    "products" => ["index", "search", "new", "edit", "save", "create", "delete"],
    "invoices" => ["index", "profile"],
];

foreach ($privateResources as $resourceName => $actions) {
    $acl->addResource(
        new \Phalcon\Acl\Resource($resourceName),
        $actions
    );
}

// Public area resources
$publicResources = [
    "index" => ["index"],
    "about" => ["index"],
    "session" => ["index", "register", "start", "end"],
    "contact" => ["index", "send"],
];
```

```
];

foreach ($publicResources as $resourceName => $actions) {
    $acl->addResource(
        new \Phalcon\Acl\Resource($resourceName),
        $actions
    );
}

// Grant access to public areas to both users and guests
foreach ($roles as $role){
    foreach ($publicResources as $resource => $actions) {
        $acl->allow($role->getName(), $resource, "*");
    }
}

// Grant access to private area to role Users
foreach ($privateResources as $resource => $actions) {
    foreach ($actions as $action) {
        $acl->allow("Users", $resource, $action);
    }
}
```

Methods

public **__construct** ()

Phalcon\Acl\Adapter\Memory constructor

public **addRole** (*RoleInterface* | *string* \$role, [*array* | *string* \$accessInherits])

Adds a role to the ACL list. Second parameter allows inheriting access data from other existing role Example:

```
<?php

$acl->addRole(
    new Phalcon\Acl\Role("administrator"),
    "consultant"
);

$acl->addRole("administrator", "consultant");
```

public **addInherit** (*mixed* \$roleName, *mixed* \$roleToInherit)

Do a role inherit from another existing role

public **isRole** (*mixed* \$roleName)

Check whether role exist in the roles list

public **isResource** (*mixed* \$resourceName)

Check whether resource exist in the resources list

public **addResource** (*Phalcon\Acl\Resource* | *string* \$resourceValue, *array* | *string* \$accessList)

Adds a resource to the ACL list Access names can be a particular action, by example search, update, delete, etc or a list of them Example:

```

<?php

// Add a resource to the the list allowing access to an action
$acl->addResource(
    new Phalcon\Acl\Resource("customers"),
    "search"
);

$acl->addResource("customers", "search");

// Add a resource with an access list
$acl->addResource(
    new Phalcon\Acl\Resource("customers"),
    [
        "create",
        "search",
    ]
);

$acl->addResource(
    "customers",
    [
        "create",
        "search",
    ]
);

```

public **addResourceAccess** (*mixed* \$resourceName, *array* | *string* \$accessList)

Adds access to resources

public **dropResourceAccess** (*mixed* \$resourceName, *array* | *string* \$accessList)

Removes an access from a resource

protected **_allowOrDeny** (*mixed* \$roleName, *mixed* \$resourceName, *mixed* \$access, *mixed* \$action, [*mixed* \$func])

Checks if a role has access to a resource

public **allow** (*mixed* \$roleName, *mixed* \$resourceName, *mixed* \$access, [*mixed* \$func])

Allow access to a role on a resource You can use '*' as wildcard Example:

```

<?php

//Allow access to guests to search on customers
$acl->allow("guests", "customers", "search");

//Allow access to guests to search or create on customers
$acl->allow("guests", "customers", ["search", "create"]);

//Allow access to any role to browse on products
$acl->allow("*", "products", "browse");

//Allow access to any role to browse on any resource
$acl->allow("*", "*", "browse");

```

public **deny** (*mixed* \$roleName, *mixed* \$resourceName, *mixed* \$access, [*mixed* \$func])

Deny access to a role on a resource You can use '*' as wildcard Example:

```
<?php

//Deny access to guests to search on customers
$acl->deny("guests", "customers", "search");

//Deny access to guests to search or create on customers
$acl->deny("guests", "customers", ["search", "create"]);

//Deny access to any role to browse on products
$acl->deny("*", "products", "browse");

//Deny access to any role to browse on any resource
$acl->deny("*", "*", "browse");
```

public **isAllowed** (*RoleInterface* | *RoleAware* | string \$roleName, *ResourceInterface* | *ResourceAware* | string \$resourceName, mixed \$access, [array \$parameters])

Check whether a role is allowed to access an action from a resource

```
<?php

//Does andres have access to the customers resource to create?
$acl->isAllowed("andres", "Products", "create");

//Do guests have access to any resource to edit?
$acl->isAllowed("guests", "*", "edit");
```

public **setNoArgumentsDefaultAction** (mixed \$defaultAccess)

Sets the default access level (Phalcon\Acl::ALLOW or Phalcon\Acl::DENY) for no arguments provided in isAllowed action if there exists func for accessKey

public **getNoArgumentsDefaultAction** ()

Returns the default ACL access level for no arguments provided in isAllowed action if there exists func for accessKey

public **getRoles** ()

Return an array with every role registered in the list

public **getResources** ()

Return an array with every resource registered in the list

public **getActiveRole** () inherited from *Phalcon\Acl\Adapter*

Role which the list is checking if it's allowed to certain resource/access

public **getActiveResource** () inherited from *Phalcon\Acl\Adapter*

Resource which the list is checking if some role can access it

public **getActiveAccess** () inherited from *Phalcon\Acl\Adapter*

Active access which the list is checking if some role can access it

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Acl\Adapter*

Sets the events manager

public **getEventManager** () inherited from *Phalcon\Acl\Adapter*

Returns the internal event manager

public **setDefaultAction** (mixed \$defaultAccess) inherited from *Phalcon\Acl\Adapter*

Sets the default access level (Phalcon\Acl::ALLOW or Phalcon\Acl::DENY)

public **getDefaultAction** () inherited from *Phalcon\Acl\Adapter*

Returns the default ACL access level

Class Phalcon\Acl\Exception

extends class Phalcon\Exception

implements Throwable

Methods

final private **Exception** **__clone** () inherited from **Exception**

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [**Exception** \$previous]) inherited from **Exception**

Exception constructor

public **__wakeup** () inherited from **Exception**

...

final public *string* **getMessage** () inherited from **Exception**

Gets the Exception message

final public *int* **getCode** () inherited from **Exception**

Gets the Exception code

final public *string* **getFile** () inherited from **Exception**

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from **Exception**

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from **Exception**

Gets the stack trace

final public **Exception** **getPrevious** () inherited from **Exception**

Returns previous Exception

final public **Exception** **getTraceAsString** () inherited from **Exception**

Gets the stack trace as a string

public *string* **__toString** () inherited from **Exception**

String representation of the exception

Class Phalcon\Acl\Resource

implements Phalcon\Acl\ResourceInterface

This class defines resource entity and its description

Methods

public **getName** ()

Resource name

public **__toString** ()

Resource name

public **getDescription** ()

Resource description

public **__construct** (*mixed* \$name, [*mixed* \$description])

Phalcon\Acl\Resource constructor

Class Phalcon\Acl\Role

implements [Phalcon\Acl\RoleInterface](#)

This class defines role entity and its description

Methods

public **getName** ()

Role name

public **__toString** ()

Role name

public **getDescription** ()

Role description

public **__construct** (*mixed* \$name, [*mixed* \$description])

Phalcon\Acl\Role constructor

Abstract class Phalcon\Annotations\Adapter

implements [Phalcon\Annotations\AdapterInterface](#)

This is the base class for Phalcon\Annotations adapters

Methods

public **setReader** ([Phalcon\Annotations\ReaderInterface](#) \$reader)

Sets the annotations parser

public **getReader** ()

Returns the annotation reader

public **get** (*string* | *object* \$className)

Parses or retrieves all the annotations found in a class

public **getMethods** (*mixed* \$className)

Returns the annotations found in all the class' methods

public **getMethod** (*mixed* \$className, *mixed* \$methodName)

Returns the annotations found in a specific method

public **getProperties** (*mixed* \$className)

Returns the annotations found in all the class' methods

public **getProperty** (*mixed* \$className, *mixed* \$propertyName)

Returns the annotations found in a specific property

Class `Phalcon\Annotations\Adapter\Apc`

extends abstract class `Phalcon\Annotations\Adapter`

implements `Phalcon\Annotations\AdapterInterface`

Stores the parsed annotations in APC. This adapter is suitable for production

```
<?php

use Phalcon\Annotations\Adapter\Apc;

$annotations = new Apc();
```

Methods

public **__construct** ([array \$options])

`Phalcon\Annotations\Adapter\Apc` constructor

public **read** (*mixed* \$key)

Reads parsed annotations from APC

public **write** (*mixed* \$key, `Phalcon\Annotations\Reflection` \$data)

Writes parsed annotations to APC

public **setReader** (`Phalcon\Annotations\ReaderInterface` \$reader) inherited from `Phalcon\Annotations\Adapter`

Sets the annotations parser

public **getReader** () inherited from `Phalcon\Annotations\Adapter`

Returns the annotation reader

public **get** (*string* | *object* \$className) inherited from `Phalcon\Annotations\Adapter`

Parses or retrieves all the annotations found in a class

public **getMethods** (*mixed* \$className) inherited from `Phalcon\Annotations\Adapter`

Returns the annotations found in all the class' methods

public **getMethod** (*mixed* \$className, *mixed* \$methodName) inherited from `Phalcon\Annotations\Adapter`

Returns the annotations found in a specific method

public **getProperties** (*mixed* \$className) inherited from `Phalcon\Annotations\Adapter`

Returns the annotations found in all the class' methods

public **getProperty** (*mixed* \$className, *mixed* \$propertyName) inherited from *Phalcon\Annotations\Adapter*

Returns the annotations found in a specific property

Class *Phalcon\Annotations\Adapter\Files*

extends abstract class *Phalcon\Annotations\Adapter*

implements *Phalcon\Annotations\AdapterInterface*

Stores the parsed annotations in files. This adapter is suitable for production

```
<?php

use Phalcon\Annotations\Adapter\Files;

$annotations = new Files(
    [
        "annotationsDir" => "app/cache/annotations/",
    ]
);
```

Methods

public **__construct** ([array \$options])

Phalcon\Annotations\Adapter\Files constructor

public *Phalcon\Annotations\Reflection* **read** (*string* \$key)

Reads parsed annotations from files

public **write** (*mixed* \$key, *Phalcon\Annotations\Reflection* \$data)

Writes parsed annotations to files

public **setReader** (*Phalcon\Annotations\ReaderInterface* \$reader) inherited from *Phalcon\Annotations\Adapter*

Sets the annotations parser

public **getReader** () inherited from *Phalcon\Annotations\Adapter*

Returns the annotation reader

public **get** (*string* | *object* \$className) inherited from *Phalcon\Annotations\Adapter*

Parses or retrieves all the annotations found in a class

public **getMethods** (*mixed* \$className) inherited from *Phalcon\Annotations\Adapter*

Returns the annotations found in all the class' methods

public **getMethod** (*mixed* \$className, *mixed* \$methodName) inherited from *Phalcon\Annotations\Adapter*

Returns the annotations found in a specific method

public **getProperties** (*mixed* \$className) inherited from *Phalcon\Annotations\Adapter*

Returns the annotations found in all the class' methods

public **getProperty** (*mixed* \$className, *mixed* \$propertyName) inherited from *Phalcon\Annotations\Adapter*

Returns the annotations found in a specific property

Class `Phalcon\Annotations\Adapter\Memory`

extends abstract class `Phalcon\Annotations\Adapter`

implements `Phalcon\Annotations\AdapterInterface`

Stores the parsed annotations in memory. This adapter is the suitable development/testing

Methods

public **read** (*mixed* \$key)

Reads parsed annotations from memory

public **write** (*mixed* \$key, `Phalcon\Annotations\Reflection` \$data)

Writes parsed annotations to memory

public **setReader** (`Phalcon\Annotations\ReaderInterface` \$reader) inherited from `Phalcon\Annotations\Adapter`

Sets the annotations parser

public **getReader** () inherited from `Phalcon\Annotations\Adapter`

Returns the annotation reader

public **get** (*string* | *object* \$className) inherited from `Phalcon\Annotations\Adapter`

Parses or retrieves all the annotations found in a class

public **getMethods** (*mixed* \$className) inherited from `Phalcon\Annotations\Adapter`

Returns the annotations found in all the class' methods

public **getMethod** (*mixed* \$className, *mixed* \$methodName) inherited from `Phalcon\Annotations\Adapter`

Returns the annotations found in a specific method

public **getProperties** (*mixed* \$className) inherited from `Phalcon\Annotations\Adapter`

Returns the annotations found in all the class' methods

public **getProperty** (*mixed* \$className, *mixed* \$propertyName) inherited from `Phalcon\Annotations\Adapter`

Returns the annotations found in a specific property

Class `Phalcon\Annotations\Adapter\Xcache`

extends abstract class `Phalcon\Annotations\Adapter`

implements `Phalcon\Annotations\AdapterInterface`

Stores the parsed annotations to XCache. This adapter is suitable for production

```
<?php
$annotations = new \Phalcon\Annotations\Adapter\Xcache();
```

Methods

public *Phalcon\Annotations\Reflection* **read** (*string* \$key)

Reads parsed annotations from XCache

public **write** (*mixed* \$key, *Phalcon\Annotations\Reflection* \$data)

Writes parsed annotations to XCache

public **setReader** (*Phalcon\Annotations\ReaderInterface* \$reader) inherited from *Phalcon\Annotations\Adapter*

Sets the annotations parser

public **getReader** () inherited from *Phalcon\Annotations\Adapter*

Returns the annotation reader

public **get** (*string* | *object* \$className) inherited from *Phalcon\Annotations\Adapter*

Parses or retrieves all the annotations found in a class

public **getMethods** (*mixed* \$className) inherited from *Phalcon\Annotations\Adapter*

Returns the annotations found in all the class' methods

public **getMethod** (*mixed* \$className, *mixed* \$methodName) inherited from *Phalcon\Annotations\Adapter*

Returns the annotations found in a specific method

public **getProperties** (*mixed* \$className) inherited from *Phalcon\Annotations\Adapter*

Returns the annotations found in all the class' methods

public **getProperty** (*mixed* \$className, *mixed* \$propertyName) inherited from *Phalcon\Annotations\Adapter*

Returns the annotations found in a specific property

Class *Phalcon\Annotations\Annotation*

Represents a single annotation in an annotations collection

Methods

public **__construct** (*array* \$reflectionData)

Phalcon\Annotations\Annotation constructor

public **getName** ()

Returns the annotation's name

public *mixed* **getExpression** (*array* \$expr)

Resolves an annotation expression

public *array* **getExprArguments** ()

Returns the expression arguments without resolving

public *array* **getArguments** ()

Returns the expression arguments

public **numberArguments** ()

Returns the number of arguments that the annotation has

public *mixed* **getArgument** (*int* | *string* \$position)

Returns an argument in a specific position

public *boolean* **hasArgument** (*int* | *string* \$position)

Returns an argument in a specific position

public *mixed* **getNamedArgument** (*mixed* \$name)

Returns a named argument

public *mixed* **getNamedParameter** (*mixed* \$name)

Returns a named parameter

Class `Phalcon\Annotations\Collection`

implements `Iterator`, `Traversable`, `Countable`

Represents a collection of annotations. This class allows to traverse a group of annotations easily

```
<?php

//Traverse annotations
foreach ($classAnnotations as $annotation) {
    echo "Name=", $annotation->getName(), PHP_EOL;
}

//Check if the annotations has a specific
var_dump($classAnnotations->has("Cacheable"));

//Get an specific annotation in the collection
$annotation = $classAnnotations->get("Cacheable");
```

Methods

public **__construct** ([*array* \$reflectionData])

Phalcon\Annotations\Collection constructor

public **count** ()

Returns the number of annotations in the collection

public **rewind** ()

Rewinds the internal iterator

public *Phalcon\Annotations\Annotation* **current** ()

Returns the current annotation in the iterator

public **key** ()

Returns the current position/key in the iterator

public **next** ()

Moves the internal iteration pointer to the next position

public **valid** ()

Check if the current annotation in the iterator is valid

public **getAnnotations** ()

Returns the internal annotations as an array

public **get** (*mixed* \$name)

Returns the first annotation that match a name

public **getAll** (*mixed* \$name)

Returns all the annotations that match a name

public **has** (*mixed* \$name)

Check if an annotation exists in a collection

Class Phalcon\Annotations\Exception

extends class [Exception](#)

implements [Throwable](#)

Methods

final private [Exception](#) **__clone** () inherited from [Exception](#)

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [[Exception](#) \$previous]) inherited from [Exception](#)

Exception constructor

public **__wakeup** () inherited from [Exception](#)

...

final public *string* **getMessage** () inherited from [Exception](#)

Gets the Exception message

final public *int* **getCode** () inherited from [Exception](#)

Gets the Exception code

final public *string* **getFile** () inherited from [Exception](#)

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from [Exception](#)

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Annotations\Reader](#)

implements [Phalcon\Annotations\ReaderInterface](#)

Parses docblocks returning an array with the found annotations

Methods

public **parse** (*mixed* \$className)

Reads annotations from the class docblocks, its methods and/or properties

public static **parseDocBlock** (*mixed* \$docBlock, [*mixed* \$file], [*mixed* \$line])

Parses a raw doc block returning the annotations found

Class [Phalcon\Annotations\Reflection](#)

Allows to manipulate the annotations reflection in an OO manner

```
<?php

use Phalcon\Annotations\Reader;
use Phalcon\Annotations\Reflection;

// Parse the annotations in a class
$reader = new Reader();
$parsing = $reader->parse("MyComponent");

// Create the reflection
$reflection = new Reflection($parsing);

// Get the annotations in the class docblock
$classAnnotations = $reflection->getClassAnnotations();
```

Methods

public **__construct** ([*array* \$reflectionData])

[Phalcon\Annotations\Reflection](#) constructor

public **getClassAnnotations** ()

Returns the annotations found in the class docblock

public **getMethodsAnnotations** ()

Returns the annotations found in the methods' docblocks

public **getPropertiesAnnotations** ()

Returns the annotations found in the properties' docblocks

public *array* **getReflectionData** ()

Returns the raw parsing intermediate definitions used to construct the reflection

public static array data **__set_state** (*mixed* \$data)

Restores the state of a Phalcon\Annotations\Reflection variable export

Abstract class Phalcon\Application

extends abstract class *Phalcon\Di\Injectable*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\Di\InjectionAwareInterface*

Base class for Phalcon\Cli\Console and Phalcon\Mvc\Application.

Methods

public **__construct** ([*Phalcon\DiInterface* \$dependencyInjector])

public **setEventsManager** (*Phalcon\Events\ManagerInterface* \$eventsManager)

Sets the events manager

public **getEventsManager** ()

Returns the internal event manager

public **registerModules** (array \$modules, [*mixed* \$merge])

Register an array of modules present in the application

```
<?php
$this->registerModules(
    [
        "frontend" => [
            "className" => "Multiple\\Frontend\\Module",
            "path"      => "../apps/frontend/Module.php",
        ],
        "backend" => [
            "className" => "Multiple\\Backend\\Module",
            "path"      => "../apps/backend/Module.php",
        ],
    ],
    $merge
);
```

public **getModules** ()

Return the modules registered in the application

public **getModule** (*mixed* \$name)

Gets the module definition registered in the application via module name

public **setDefaultModule** (*mixed* \$defaultModule)

Sets the module name to be used if the router doesn't return a valid module

public **getDefaultModule** ()

Returns the default module name

abstract public **handle** ()

Handles a request

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di\Injectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method **__get**

Class *Phalcon\Application\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Class Phalcon\Assets\Collection

implements Countable, Iterator, Traversable

Represents a collection of resources

Methods

public **getPrefix** ()

...

public **getLocal** ()

...

public **getResources** ()

...

public **getCodes** ()

...

public **getPosition** ()

...

public **getFilters** ()

...

public **getAttributes** ()

...

public **getJoin** ()

...

public **getTargetUri** ()

...

public **getTargetPath** ()

...

public **getTargetLocal** ()

...

public **getSourcePath** ()

...

public **add** (*Phalcon\Assets\Resource* \$resource)

Adds a resource to the collection

public **addInline** (*Phalcon\Assets\Inline* \$code)

Adds an inline code to the collection

public **addCss** (*mixed* \$path, [*mixed* \$local], [*mixed* \$filter], [*mixed* \$attributes])

Adds a CSS resource to the collection

public **addInlineCss** (*mixed* \$content, [*mixed* \$filter], [*mixed* \$attributes])

Adds an inline CSS to the collection

public *Phalcon\Assets\Collection* **addJs** (*string* \$path, [*boolean* \$local], [*boolean* \$filter], [*array* \$attributes])

Adds a javascript resource to the collection

public **addInlineJs** (*mixed* \$content, [*mixed* \$filter], [*mixed* \$attributes])

Adds an inline javascript to the collection

public **count** ()

Returns the number of elements in the form

public **rewind** ()

Rewinds the internal iterator

public **current** ()

Returns the current resource in the iterator

public *int* **key** ()

Returns the current position/key in the iterator

public **next** ()

Moves the internal iteration pointer to the next position

public **valid** ()

Check if the current element in the iterator is valid

public **setTargetPath** (*mixed* \$targetPath)

Sets the target path of the file for the filtered/join output

public **setSourcePath** (*mixed* \$sourcePath)

Sets a base source path for all the resources in this collection

public **setTargetUri** (*mixed* \$targetUri)

Sets a target uri for the generated HTML

public **setPrefix** (*mixed* \$prefix)

Sets a common prefix for all the resources

public **setLocal** (*mixed* \$local)

Sets if the collection uses local resources by default

public **setAttributes** (*array* \$attributes)

Sets extra HTML attributes

public **setFilters** (*array* \$filters)

Sets an array of filters in the collection

public **setTargetLocal** (*mixed* \$targetLocal)

Sets the target local

public **join** (*mixed* \$join)

Sets if all filtered resources in the collection must be joined in a single result file

public **getRealTargetPath** (*mixed* \$basePath)

Returns the complete location where the joined/filtered collection must be written

public **addFilter** (*Phalcon\Assets\FilterInterface* \$filter)

Adds a filter to the collection

Class *Phalcon\Assets\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Class *Phalcon\Assets\Filters\Cssmin*

implements *Phalcon\Assets\FilterInterface*

Minify the css - removes comments removes newlines and line feeds keeping removes last semicolon from last property

Methods

public **filter** (*mixed* \$content)

Filters the content using CSSMIN

Class Phalcon\Assets\Filters\Jsmin

implements *Phalcon\Assets\FilterInterface*

Deletes the characters which are insignificant to JavaScript. Comments will be removed. Tabs will be replaced with spaces. Carriage returns will be replaced with linefeeds. Most spaces and linefeeds will be removed.

Methods

public **filter** (*mixed* \$content)

Filters the content using JSMIN

Class Phalcon\Assets\Filters\None

implements *Phalcon\Assets\FilterInterface*

Returns the content without make any modification to the original source

Methods

public **filter** (*mixed* \$content)

Returns the content without be touched

Class Phalcon\Assets\Inline

Represents an inline asset

```
<?php
$inline = new \Phalcon\Assets\Inline("js", "alert('hello world');");
```

Methods

public **getType** ()

...

public **getContent** ()

...

public **getFilter** ()

...

public **getAttributes** ()

...

public **__construct** (*string* \$type, *string* \$content, [*boolean* \$filter], [*array* \$attributes])

Phalcon\Assets\Inline constructor

public **setType** (*mixed* \$type)

Sets the inline's type

public **setFilter** (*mixed* \$filter)

Sets if the resource must be filtered or not

public **setAttributes** (*array* \$attributes)

Sets extra HTML attributes

Class Phalcon\Assets\Inline\Css

extends class Phalcon\Assets\Inline

Represents an inlined CSS

Methods

public **__construct** (*string* \$content, [*boolean* \$filter], [*array* \$attributes])

public **getType** () inherited from *Phalcon\Assets\Inline*

...

public **getContent** () inherited from *Phalcon\Assets\Inline*

...

public **getFilter** () inherited from *Phalcon\Assets\Inline*

...

public **getAttributes** () inherited from *Phalcon\Assets\Inline*

...

public **setType** (*mixed* \$type) inherited from *Phalcon\Assets\Inline*

Sets the inline's type

public **setFilter** (*mixed* \$filter) inherited from *Phalcon\Assets\Inline*

Sets if the resource must be filtered or not

public **setAttributes** (*array* \$attributes) inherited from *Phalcon\Assets\Inline*

Sets extra HTML attributes

Class Phalcon\Assets\Inline\Js

extends class Phalcon\Assets\Inline

Represents an inline Javascript

Methods

public **__construct** (*string* \$content, [*boolean* \$filter], [*array* \$attributes])
public **getType** () inherited from *Phalcon\Assets\Inline*
...
public **getContent** () inherited from *Phalcon\Assets\Inline*
...
public **getFilter** () inherited from *Phalcon\Assets\Inline*
...
public **getAttributes** () inherited from *Phalcon\Assets\Inline*
...
public **setType** (*mixed* \$type) inherited from *Phalcon\Assets\Inline*
Sets the inline's type
public **setFilter** (*mixed* \$filter) inherited from *Phalcon\Assets\Inline*
Sets if the resource must be filtered or not
public **setAttributes** (*array* \$attributes) inherited from *Phalcon\Assets\Inline*
Sets extra HTML attributes

Class **Phalcon\Assets\Manager**

Manages collections of CSS/Javascript assets

Methods

public **__construct** ([*array* \$options])
public **setOptions** (*array* \$options)
Sets the manager options
public **getOptions** ()
Returns the manager options
public **useImplicitOutput** (*mixed* \$implicitOutput)
Sets if the HTML generated must be directly printed or returned
public **addCss** (*mixed* \$path, [*mixed* \$local], [*mixed* \$filter], [*mixed* \$attributes])
Adds a Css resource to the 'css' collection

```
<?php
$assets->addCss("css/bootstrap.css");
$assets->addCss("http://bootstrap.my-cdn.com/style.css", false);
```

public **addInlineCss** (*mixed* \$content, [*mixed* \$filter], [*mixed* \$attributes])

Adds an inline Css to the 'css' collection

public **addJs** (*mixed* \$path, [*mixed* \$local], [*mixed* \$filter], [*mixed* \$attributes])

Adds a javascript resource to the 'js' collection

```
<?php
$assets->addJs("scripts/jquery.js");
$assets->addJs("http://jquery.my-cdn.com/jquery.js", false);
```

public **addInlineJs** (*mixed* \$content, [*mixed* \$filter], [*mixed* \$attributes])

Adds an inline javascript to the 'js' collection

public **addResourceByType** (*mixed* \$type, *Phalcon\Assets\Resource* \$resource)

Adds a resource by its type

```
<?php
$assets->addResourceByType("css",
    new \Phalcon\Assets\Resource\Css("css/style.css")
);
```

public **addInlineCodeByType** (*mixed* \$type, *Phalcon\Assets\Inline* \$code)

Adds an inline code by its type

public **addResource** (*Phalcon\Assets\Resource* \$resource)

Adds a raw resource to the manager

```
<?php
$assets->addResource(
    new \Phalcon\Assets\Resource("css", "css/style.css")
);
```

public **addInlineCode** (*Phalcon\Assets\Inline* \$code)

Adds a raw inline code to the manager

public **set** (*mixed* \$id, *Phalcon\Assets\Collection* \$collection)

Sets a collection in the Assets Manager

```
<?php
$assets->set("js", $collection);
```

public **get** (*mixed* \$id)

Returns a collection by its id

```
<?php
$scripts = $assets->get("js");
```

public **getCss** ()

Returns the CSS collection of assets

public **getJs** ()

Returns the CSS collection of assets

public **collection** (*mixed* \$name)

Creates/Returns a collection of resources

public **output** (*Phalcon\Assets\Collection* \$collection, *callback* \$callback, *string* \$type)

Traverses a collection calling the callback to generate its HTML

public **outputInline** (*Phalcon\Assets\Collection* \$collection, *string* \$type)

Traverses a collection and generate its HTML

public **outputCss** ([*string* \$collectionName])

Prints the HTML for CSS resources

public **outputInlineCss** ([*string* \$collectionName])

Prints the HTML for inline CSS

public **outputJs** ([*string* \$collectionName])

Prints the HTML for JS resources

public **outputInlineJs** ([*string* \$collectionName])

Prints the HTML for inline JS

public **getCollections** ()

Returns existing collections in the manager

public **exists** (*mixed* \$id)

Returns true or false if collection exists

Class **Phalcon\Assets\Resource**

Represents an asset resource

```
<?php
$resource = new \Phalcon\Assets\Resource("js", "javascripts/jquery.js");
```

Methods

public **getType** ()

public **getPath** ()

public **getLocal** ()

public **getFilter** ()

public **getAttributes** ()

public **getSourcePath** ()

...

public **getTargetPath** ()

...

public **getTargetUri** ()

...

public **__construct** (*string* \$type, *string* \$path, [*boolean* \$local], [*boolean* \$filter], [*array* \$attributes])

Phalcon\Assets\Resource constructor

public **setType** (*mixed* \$type)

Sets the resource's type

public **setPath** (*mixed* \$path)

Sets the resource's path

public **setLocal** (*mixed* \$local)

Sets if the resource is local or external

public **setFilter** (*mixed* \$filter)

Sets if the resource must be filtered or not

public **setAttributes** (*array* \$attributes)

Sets extra HTML attributes

public **setTargetUri** (*mixed* \$targetUri)

Sets a target uri for the generated HTML

public **setSourcePath** (*mixed* \$sourcePath)

Sets the resource's source path

public **setTargetPath** (*mixed* \$targetPath)

Sets the resource's target path

public **getContent** ([*mixed* \$basePath])

Returns the content of the resource as an string Optionally a base path where the resource is located can be set

public **getRealTargetUri** ()

Returns the real target uri for the generated HTML

public **getRealSourcePath** ([*mixed* \$basePath])

Returns the complete location where the resource is located

public **getRealTargetPath** ([*mixed* \$basePath])

Returns the complete location where the resource must be written

Class Phalcon\Assets\Resource\Css

extends class *Phalcon\Assets\Resource*

Represents CSS resources

Methods

public **__construct** (*string* \$path, [*boolean* \$local], [*boolean* \$filter], [*array* \$attributes])

public **getType** () inherited from *Phalcon\Assets\Resource*

public **getPath** () inherited from *Phalcon\Assets\Resource*

public **getLocal** () inherited from *Phalcon\Assets\Resource*

public **getFilter** () inherited from *Phalcon\Assets\Resource*

public **getAttributes** () inherited from *Phalcon\Assets\Resource*

public **getSourcePath** () inherited from *Phalcon\Assets\Resource*

...

public **getTargetPath** () inherited from *Phalcon\Assets\Resource*

...

public **getTargetUri** () inherited from *Phalcon\Assets\Resource*

...

public **setType** (*mixed* \$type) inherited from *Phalcon\Assets\Resource*

Sets the resource's type

public **setPath** (*mixed* \$path) inherited from *Phalcon\Assets\Resource*

Sets the resource's path

public **setLocal** (*mixed* \$local) inherited from *Phalcon\Assets\Resource*

Sets if the resource is local or external

public **setFilter** (*mixed* \$filter) inherited from *Phalcon\Assets\Resource*

Sets if the resource must be filtered or not

public **setAttributes** (*array* \$attributes) inherited from *Phalcon\Assets\Resource*

Sets extra HTML attributes

public **setTargetUri** (*mixed* \$targetUri) inherited from *Phalcon\Assets\Resource*

Sets a target uri for the generated HTML

public **setSourcePath** (*mixed* \$sourcePath) inherited from *Phalcon\Assets\Resource*

Sets the resource's source path

public **setTargetPath** (*mixed* \$targetPath) inherited from *Phalcon\Assets\Resource*

Sets the resource's target path

public **getContent** ([*mixed* \$basePath]) inherited from *Phalcon\Assets\Resource*

Returns the content of the resource as an string Optionally a base path where the resource is located can be set

public **getRealTargetUri** () inherited from *Phalcon\Assets\Resource*

Returns the real target uri for the generated HTML

public **getRealSourcePath** ([*mixed* \$basePath]) inherited from *Phalcon\Assets\Resource*

Returns the complete location where the resource is located

public **getRealTargetPath** ([*mixed* \$basePath]) inherited from *Phalcon\Assets\Resource*

Returns the complete location where the resource must be written

Class `Phalcon\Assets\Resource\Js`

extends class `Phalcon\Assets\Resource`

Represents Javascript resources

Methods

public **__construct** (*string* \$path, [*boolean* \$local], [*boolean* \$filter], [*array* \$attributes])

public **getType** () inherited from *Phalcon\Assets\Resource*

public **getPath** () inherited from *Phalcon\Assets\Resource*

public **getLocal** () inherited from *Phalcon\Assets\Resource*

public **getFilter** () inherited from *Phalcon\Assets\Resource*

public **getAttributes** () inherited from *Phalcon\Assets\Resource*

public **getSourcePath** () inherited from *Phalcon\Assets\Resource*

...

public **getTargetPath** () inherited from *Phalcon\Assets\Resource*

...

public **getTargetUri** () inherited from *Phalcon\Assets\Resource*

...

public **setType** (*mixed* \$type) inherited from *Phalcon\Assets\Resource*

Sets the resource's type

public **setPath** (*mixed* \$path) inherited from *Phalcon\Assets\Resource*

Sets the resource's path

public **setLocal** (*mixed* \$local) inherited from *Phalcon\Assets\Resource*

Sets if the resource is local or external

public **setFilter** (*mixed* \$filter) inherited from *Phalcon\Assets\Resource*

Sets if the resource must be filtered or not

public **setAttributes** (*array* \$attributes) inherited from *Phalcon\Assets\Resource*

Sets extra HTML attributes

public **setTargetUri** (*mixed* \$targetUri) inherited from *Phalcon\Assets\Resource*

Sets a target uri for the generated HTML

public **setSourcePath** (*mixed* \$sourcePath) inherited from *Phalcon\Assets\Resource*

Sets the resource's source path

public **setTargetPath** (*mixed* \$targetPath) inherited from *Phalcon\Assets\Resource*

Sets the resource's target path

public **getContent** ([*mixed* \$basePath]) inherited from *Phalcon\Assets\Resource*

Returns the content of the resource as an string Optionally a base path where the resource is located can be set

public **getRealTargetUri** () inherited from *Phalcon\Assets\Resource*

Returns the real target uri for the generated HTML

public **getRealSourcePath** ([*mixed* \$basePath]) inherited from *Phalcon\Assets\Resource*

Returns the complete location where the resource is located

public **getRealTargetPath** ([*mixed* \$basePath]) inherited from *Phalcon\Assets\Resource*

Returns the complete location where the resource must be written

Abstract class **Phalcon\Cache\Backend**

implements Phalcon\Cache\BackendInterface

This class implements common functionality for backend adapters. A backend cache adapter may extend this class

Methods

public **getFrontend** ()

...

public **setFrontend** (*mixed* \$frontend)

...

public **getOptions** ()

...

public **setOptions** (*mixed* \$options)

...

public **getLastKey** ()

...

public **setLastKey** (*mixed* \$lastKey)

...

public **__construct** (*Phalcon\Cache\FrontendInterface* \$frontend, [*array* \$options])

Phalcon\Cache\Backend constructor

public *mixed* **start** (*int* | *string* \$keyName, [*int* \$lifetime])

Starts a cache. The keyname allows to identify the created fragment

public **stop** ([*mixed* \$stopBuffer])

Stops the frontend without store any cached content

public **isFresh** ()

Checks whether the last cache is fresh or cached

public **isStarted** ()

Checks whether the cache has starting buffering or not

public *int* **getLifetime** ()

Gets the last lifetime set

abstract public **get** (*mixed* \$keyName, [*mixed* \$lifetime]) inherited from *Phalcon\Cache\BackendInterface*

...

abstract public **save** ([*mixed* \$keyName], [*mixed* \$content], [*mixed* \$lifetime], [*mixed* \$stopBuffer]) inherited from *Phalcon\Cache\BackendInterface*

...

abstract public **delete** (*mixed* \$keyName) inherited from *Phalcon\Cache\BackendInterface*

...

abstract public **queryKeys** ([*mixed* \$prefix]) inherited from *Phalcon\Cache\BackendInterface*

...

abstract public **exists** ([*mixed* \$keyName], [*mixed* \$lifetime]) inherited from *Phalcon\Cache\BackendInterface*

...

Class *Phalcon\Cache\Backend\Apc*

extends abstract class *Phalcon\Cache\Backend*

implements *Phalcon\Cache\BackendInterface*

Allows to cache output fragments, PHP data and raw data using an APC backend

```
<?php

use Phalcon\Cache\Backend\Apc;
use Phalcon\Cache\Frontend\Data as FrontData;

// Cache data for 2 days
$frontCache = new FrontData(
    [
        "lifetime" => 172800,
    ]
);

$cache = new Apc(
    $frontCache,
    [
        "prefix" => "app-data",
    ]
);

// Cache arbitrary data
$cache->save("my-data", [1, 2, 3, 4, 5]);

// Get data
$data = $cache->get("my-data");
```

Methods

public **get** (*mixed* \$keyName, [*mixed* \$lifetime])

Returns a cached content

public **save** ([*string* | *int* \$keyName], [*string* \$content], [*int* \$lifetime], [*boolean* \$stopBuffer])

Stores cached content into the APC backend and stops the frontend

public **increment** ([*string* \$keyName], [*mixed* \$value])

Increment of a given key, by number \$value

public **decrement** ([*string* \$keyName], [*mixed* \$value])

Decrement of a given key, by number \$value

public **delete** (*mixed* \$keyName)

Deletes a value from the cache by its key

public **queryKeys** ([*mixed* \$prefix])

Query the existing cached keys.

```
<?php

$cache->save("users-ids", [1, 2, 3]);
$cache->save("projects-ids", [4, 5, 6]);

var_dump($cache->queryKeys("users")); // ["users-ids"]
```

public **exists** ([*string* | *int* \$keyName], [*int* \$lifetime])

Checks if cache exists and it hasn't expired

public **flush** ()

Immediately invalidates all existing items.

```
<?php

use Phalcon\Cache\Backend\Apc;

$cache = new Apc($frontCache, ["prefix" => "app-data"]);

$cache->save("my-data", [1, 2, 3, 4, 5]);

// 'my-data' and all other used keys are deleted
$cache->flush();
```

public **getFrontend** () inherited from *Phalcon\Cache\Backend*

...

public **setFrontend** (*mixed* \$frontend) inherited from *Phalcon\Cache\Backend*

...

public **getOptions** () inherited from *Phalcon\Cache\Backend*

...

public **setOptions** (*mixed* \$options) inherited from *Phalcon\Cache\Backend*

...

public **getLastKey** () inherited from *Phalcon\Cache\Backend*

...

public **setLastKey** (*mixed* \$lastKey) inherited from *Phalcon\Cache\Backend*

...

public **__construct** (*Phalcon\Cache\FrontendInterface* \$frontend, [*array* \$options]) inherited from *Phalcon\Cache\Backend*

Phalcon\Cache\Backend constructor

public *mixed* **start** (*int* | *string* \$keyName, [*int* \$lifetime]) inherited from *Phalcon\Cache\Backend*

Starts a cache. The keyname allows to identify the created fragment

public **stop** ([*mixed* \$stopBuffer]) inherited from *Phalcon\Cache\Backend*

Stops the frontend without store any cached content

public **isFresh** () inherited from *Phalcon\Cache\Backend*

Checks whether the last cache is fresh or cached

public **isStarted** () inherited from *Phalcon\Cache\Backend*

Checks whether the cache has starting buffering or not

public *int* **getLifetime** () inherited from *Phalcon\Cache\Backend*

Gets the last lifetime set

Class Phalcon\Cache\Backend\File

extends abstract class *Phalcon\Cache\Backend*

implements *Phalcon\Cache\BackendInterface*

Allows to cache output fragments using a file backend

```
<?php

use Phalcon\Cache\Backend\File;
use Phalcon\Cache\Frontend\Output as FrontOutput;

// Cache the file for 2 days
$frontendOptions = [
    "lifetime" => 172800,
];

// Create an output cache
$frontCache = FrontOutput($frontendOptions);

// Set the cache directory
$backendOptions = [
    "cacheDir" => "../app/cache/",
];

// Create the File backend
$cache = new File($frontCache, $backendOptions);

$content = $cache->start("my-cache");

if ($content === null) {
    echo "<h1>", time(), "</h1>";
}
```



```

    $cache->save();
} else {
    echo $content;
}

```

Methods

public **__construct** (*Phalcon\Cache\FrontendInterface* \$frontend, array \$options)

Phalcon\Cache\Backend\File constructor

public **get** (*mixed* \$keyName, [*mixed* \$lifetime])

Returns a cached content

public **save** ([*int* | *string* \$keyName], [*string* \$content], [*int* \$lifetime], [*boolean* \$stopBuffer])

Stores cached content into the file backend and stops the frontend

public **delete** (*int* | *string* \$keyName)

Deletes a value from the cache by its key

public **queryKeys** ([*mixed* \$prefix])

Query the existing cached keys.

```

<?php

$cache->save("users-ids", [1, 2, 3]);
$cache->save("projects-ids", [4, 5, 6]);

var_dump($cache->queryKeys("users")); // ["users-ids"]

```

public **exists** ([*string* | *int* \$keyName], [*int* \$lifetime])

Checks if cache exists and it isn't expired

public **increment** ([*string* | *int* \$keyName], [*mixed* \$value])

Increment of a given key, by number \$value

public **decrement** ([*string* | *int* \$keyName], [*mixed* \$value])

Decrement of a given key, by number \$value

public **flush** ()

Immediately invalidates all existing items.

public **getKey** (*mixed* \$key)

Return a file-system safe identifier for a given key

public **useSafeKey** (*mixed* \$useSafeKey)

Set whether to use the safekey or not

public **getFrontend** () inherited from *Phalcon\Cache\Backend*

...

public **setFrontend** (*mixed* \$frontend) inherited from *Phalcon\Cache\Backend*

...

public **getOptions** () inherited from *Phalcon\Cache\Backend*

...

public **setOptions** (*mixed* \$options) inherited from *Phalcon\Cache\Backend*

...

public **getLastKey** () inherited from *Phalcon\Cache\Backend*

...

public **setLastKey** (*mixed* \$lastKey) inherited from *Phalcon\Cache\Backend*

...

public *mixed* **start** (*int* | *string* \$keyName, [*int* \$lifetime]) inherited from *Phalcon\Cache\Backend*

Starts a cache. The keyname allows to identify the created fragment

public **stop** ([*mixed* \$stopBuffer]) inherited from *Phalcon\Cache\Backend*

Stops the frontend without store any cached content

public **isFresh** () inherited from *Phalcon\Cache\Backend*

Checks whether the last cache is fresh or cached

public **isStarted** () inherited from *Phalcon\Cache\Backend*

Checks whether the cache has starting buffering or not

public *int* **getLifetime** () inherited from *Phalcon\Cache\Backend*

Gets the last lifetime set

Class *Phalcon\Cache\Backend\Libmemcached*

extends abstract class *Phalcon\Cache\Backend*

implements *Phalcon\Cache\BackendInterface*

Allows to cache output fragments, PHP data or raw data to a libmemcached backend. Per default persistent memcached connection pools are used.

```
<?php

use Phalcon\Cache\Backend\Libmemcached;
use Phalcon\Cache\Frontend\Data as FrontData;

// Cache data for 2 days
$frontCache = new FrontData(
    [
        "lifetime" => 172800,
    ]
);

// Create the Cache setting memcached connection options
$cache = new Libmemcached(
    $frontCache,
    [
        "servers" => [
            [
                "host"    => "127.0.0.1",
```

```

        "port"    => 11211,
        "weight" => 1,
    ],
    ],
    "client" => [
        \Memcached::OPT_HASH           => \Memcached::HASH_MD5,
        \Memcached::OPT_PREFIX_KEY => "prefix.",
    ],
    ],
];

);

// Cache arbitrary data
$cache->save("my-data", [1, 2, 3, 4, 5]);

// Get data
$data = $cache->get("my-data");

```

Methods

public **__construct** (*Phalcon\Cache\FrontendInterface* \$frontend, [*array* \$options])

Phalcon\Cache\Backend\Memcache constructor

public **_connect** ()

Create internal connection to memcached

public **get** (*mixed* \$keyName, [*mixed* \$lifetime])

Returns a cached content

public **save** ([*int* | *string* \$keyName], [*string* \$content], [*int* \$lifetime], [*boolean* \$stopBuffer])

Stores cached content into the file backend and stops the frontend

public *boolean* **delete** (*int* | *string* \$keyName)

Deletes a value from the cache by its key

public **queryKeys** ([*mixed* \$prefix])

Query the existing cached keys.

```

<?php

$cache->save("users-ids", [1, 2, 3]);
$cache->save("projects-ids", [4, 5, 6]);

var_dump($cache->queryKeys("users")); // ["users-ids"]

```

public **exists** ([*string* \$keyName], [*int* \$lifetime])

Checks if cache exists and it isn't expired

public **increment** ([*string* \$keyName], [*mixed* \$value])

Increment of given \$keyName by \$value

public **decrement** ([*string* \$keyName], [*mixed* \$value])

Decrement of \$keyName by given \$value

public **flush** ()

Immediately invalidates all existing items. Memcached does not support flush() per default. If you require flush() support, set \$config["statsKey"]. All modified keys are stored in "statsKey". Note: statsKey has a negative performance impact.

```
<?php
$cache = new \Phalcon\Cache\Backend\Libmemcached(
    $frontCache,
    [
        "statsKey" => "_PHCM",
    ]
);

$cache->save("my-data", [1, 2, 3, 4, 5]);

// 'my-data' and all other used keys are deleted
$cache->flush();
```

public **getFrontend** () inherited from *Phalcon\Cache\Backend*

...

public **setFrontend** (mixed \$frontend) inherited from *Phalcon\Cache\Backend*

...

public **getOptions** () inherited from *Phalcon\Cache\Backend*

...

public **setOptions** (mixed \$options) inherited from *Phalcon\Cache\Backend*

...

public **getLastKey** () inherited from *Phalcon\Cache\Backend*

...

public **setLastKey** (mixed \$lastKey) inherited from *Phalcon\Cache\Backend*

...

public mixed **start** (int | string \$keyName, [int \$lifetime]) inherited from *Phalcon\Cache\Backend*

Starts a cache. The keyname allows to identify the created fragment

public **stop** ([mixed \$stopBuffer]) inherited from *Phalcon\Cache\Backend*

Stops the frontend without store any cached content

public **isFresh** () inherited from *Phalcon\Cache\Backend*

Checks whether the last cache is fresh or cached

public **isStarted** () inherited from *Phalcon\Cache\Backend*

Checks whether the cache has starting buffering or not

public int **getLifetime** () inherited from *Phalcon\Cache\Backend*

Gets the last lifetime set

Class Phalcon\Cache\Backend\Memcache

extends abstract class *Phalcon\Cache\Backend*

implements *Phalcon\Cache\BackendInterface*

Allows to cache output fragments, PHP data or raw data to a memcache backend

This adapter uses the special memcached key “_PHCM” to store all the keys internally used by the adapter

```

<?php

use Phalcon\Cache\Backend\Memcache;
use Phalcon\Cache\Frontend\Data as FrontData;

// Cache data for 2 days
$frontCache = new FrontData(
    [
        "lifetime" => 172800,
    ]
);

// Create the Cache setting memcached connection options
$cache = new Memcache(
    $frontCache,
    [
        "host"      => "localhost",
        "port"      => 11211,
        "persistent" => false,
    ]
);

// Cache arbitrary data
$cache->save("my-data", [1, 2, 3, 4, 5]);

// Get data
$data = $cache->get("my-data");

```

Methods

public **__construct** (*Phalcon\Cache\FrontendInterface* \$frontend, [*array* \$options])

Phalcon\Cache\Backend\Memcache constructor

public **_connect** ()

Create internal connection to memcached

public **addServers** (*mixed* \$host, *mixed* \$port, [*mixed* \$persistent])

Add servers to memcache pool

public **get** (*mixed* \$keyName, [*mixed* \$lifetime])

Returns a cached content

public **save** ([*int* | *string* \$keyName], [*string* \$content], [*int* \$lifetime], [*boolean* \$stopBuffer])

Stores cached content into the file backend and stops the frontend

public *boolean* **delete** (*int* | *string* \$keyName)

Deletes a value from the cache by its key

public **queryKeys** ([mixed \$prefix])

Query the existing cached keys.

```
<?php

$cache->save("users-ids", [1, 2, 3]);
$cache->save("projects-ids", [4, 5, 6]);

var_dump($cache->queryKeys("users")); // ["users-ids"]
```

public **exists** ([string \$keyName], [int \$lifetime])

Checks if cache exists and it isn't expired

public **increment** ([string \$keyName], [mixed \$value])

Increment of given \$keyName by \$value

public **decrement** ([string \$keyName], [mixed \$value])

Decrement of \$keyName by given \$value

public **flush** ()

Immediately invalidates all existing items.

public **getFrontend** () inherited from *Phalcon\Cache\Backend*

...

public **setFrontend** (mixed \$frontend) inherited from *Phalcon\Cache\Backend*

...

public **getOptions** () inherited from *Phalcon\Cache\Backend*

...

public **setOptions** (mixed \$options) inherited from *Phalcon\Cache\Backend*

...

public **getLastKey** () inherited from *Phalcon\Cache\Backend*

...

public **setLastKey** (mixed \$lastKey) inherited from *Phalcon\Cache\Backend*

...

public mixed **start** (int | string \$keyName, [int \$lifetime]) inherited from *Phalcon\Cache\Backend*

Starts a cache. The keyname allows to identify the created fragment

public **stop** ([mixed \$stopBuffer]) inherited from *Phalcon\Cache\Backend*

Stops the frontend without store any cached content

public **isFresh** () inherited from *Phalcon\Cache\Backend*

Checks whether the last cache is fresh or cached

public **isStarted** () inherited from *Phalcon\Cache\Backend*

Checks whether the cache has starting buffering or not

public int **getLifetime** () inherited from *Phalcon\Cache\Backend*

Gets the last lifetime set

Class `Phalcon\Cache\Backend\Memory`

extends abstract class `Phalcon\Cache\Backend`

implements `Phalcon\Cache\BackendInterface`, `Serializable`

Stores content in memory. Data is lost when the request is finished

```
<?php

use Phalcon\Cache\Backend\Memory;
use Phalcon\Cache\Frontend\Data as FrontData;

// Cache data
$frontCache = new FrontData();

$cache = new Memory($frontCache);

// Cache arbitrary data
$cache->save("my-data", [1, 2, 3, 4, 5]);

// Get data
$data = $cache->get("my-data");
```

Methods

public **get** (*mixed* \$keyName, [*mixed* \$lifetime])

Returns a cached content

public **save** ([*string* \$keyName], [*string* \$content], [*int* \$lifetime], [*boolean* \$stopBuffer])

Stores cached content into the backend and stops the frontend

public *boolean* **delete** (*string* \$keyName)

Deletes a value from the cache by its key

public **queryKeys** ([*mixed* \$prefix])

Query the existing cached keys.

```
<?php

$cache->save("users-ids", [1, 2, 3]);
$cache->save("projects-ids", [4, 5, 6]);

var_dump($cache->queryKeys("users")); // ["users-ids"]
```

public **exists** ([*string* | *int* \$keyName], [*int* \$lifetime])

Checks if cache exists and it hasn't expired

public **increment** ([*string* \$keyName], [*mixed* \$value])

Increment of given \$keyName by \$value

public **decrement** ([*string* \$keyName], [*mixed* \$value])

Decrement of \$keyName by given \$value

public **flush** ()

Immediately invalidates all existing items.

public **serialize** ()

Required for interface \Serializable

public **unserialize** (*mixed* \$data)

Required for interface \Serializable

public **getFrontend** () inherited from *Phalcon\Cache\Backend*

...

public **setFrontend** (*mixed* \$frontend) inherited from *Phalcon\Cache\Backend*

...

public **getOptions** () inherited from *Phalcon\Cache\Backend*

...

public **setOptions** (*mixed* \$options) inherited from *Phalcon\Cache\Backend*

...

public **getLastKey** () inherited from *Phalcon\Cache\Backend*

...

public **setLastKey** (*mixed* \$lastKey) inherited from *Phalcon\Cache\Backend*

...

public **__construct** (*Phalcon\Cache\FrontendInterface* \$frontend, [array \$options]) inherited from *Phalcon\Cache\Backend*

Phalcon\Cache\Backend constructor

public *mixed* **start** (*int* | *string* \$keyName, [*int* \$lifetime]) inherited from *Phalcon\Cache\Backend*

Starts a cache. The keyname allows to identify the created fragment

public **stop** ([*mixed* \$stopBuffer]) inherited from *Phalcon\Cache\Backend*

Stops the frontend without store any cached content

public **isFresh** () inherited from *Phalcon\Cache\Backend*

Checks whether the last cache is fresh or cached

public **isStarted** () inherited from *Phalcon\Cache\Backend*

Checks whether the cache has starting buffering or not

public *int* **getLifetime** () inherited from *Phalcon\Cache\Backend*

Gets the last lifetime set

Class *Phalcon\Cache\Backend\Mongo*

extends abstract class *Phalcon\Cache\Backend*

implements *Phalcon\Cache\BackendInterface*

Allows to cache output fragments, PHP data or raw data to a MongoDB backend

```
<?php

use Phalcon\Cache\Backend\Mongo;
use Phalcon\Cache\Frontend\Base64;

// Cache data for 2 days
$frontCache = new Base64(
    [
        "lifetime" => 172800,
    ]
);

// Create a MongoDB cache
$cache = new Mongo(
    $frontCache,
    [
        "server"      => "mongodb://localhost",
        "db"           => "caches",
        "collection" => "images",
    ]
);

// Cache arbitrary data
$cache->save(
    "my-data",
    file_get_contents("some-image.jpg")
);

// Get data
$data = $cache->get("my-data");
```

Methods

public **__construct** (*Phalcon\Cache\FrontendInterface* \$frontend, [*array* \$options])

Phalcon\Cache\Backend\Mongo constructor

final protected *MongoCollection* **_getCollection** ()

Returns a MongoDB collection based on the backend parameters

public **get** (*mixed* \$keyName, [*mixed* \$lifetime])

Returns a cached content

public **save** ([*int* | *string* \$keyName], [*string* \$content], [*int* \$lifetime], [*boolean* \$stopBuffer])

Stores cached content into the file backend and stops the frontend

public *boolean* **delete** (*int* | *string* \$keyName)

Deletes a value from the cache by its key

public **queryKeys** ([*mixed* \$prefix])

Query the existing cached keys.

```
<?php
$cache->save("users-ids", [1, 2, 3]);
$cache->save("projects-ids", [4, 5, 6]);

var_dump($cache->queryKeys("users")); // ["users-ids"]
```

public **exists** ([string \$keyName], [int \$lifetime])

Checks if cache exists and it isn't expired

public *collection*->**remove**(...) **gc** ()

gc

public **increment** (int | string \$keyName, [mixed \$value])

Increment of a given key by \$value

public **decrement** (int | string \$keyName, [mixed \$value])

Decrement of a given key by \$value

public **flush** ()

Immediately invalidates all existing items.

public **getFrontend** () inherited from *Phalcon\Cache\Backend*

...

public **setFrontend** (mixed \$frontend) inherited from *Phalcon\Cache\Backend*

...

public **getOptions** () inherited from *Phalcon\Cache\Backend*

...

public **setOptions** (mixed \$options) inherited from *Phalcon\Cache\Backend*

...

public **getLastKey** () inherited from *Phalcon\Cache\Backend*

...

public **setLastKey** (mixed \$lastKey) inherited from *Phalcon\Cache\Backend*

...

public mixed **start** (int | string \$keyName, [int \$lifetime]) inherited from *Phalcon\Cache\Backend*

Starts a cache. The keyname allows to identify the created fragment

public **stop** ([mixed \$stopBuffer]) inherited from *Phalcon\Cache\Backend*

Stops the frontend without store any cached content

public **isFresh** () inherited from *Phalcon\Cache\Backend*

Checks whether the last cache is fresh or cached

public **isStarted** () inherited from *Phalcon\Cache\Backend*

Checks whether the cache has starting buffering or not

public int **getLifetime** () inherited from *Phalcon\Cache\Backend*

Gets the last lifetime set

Class `Phalcon\Cache\Backend\Redis`

extends abstract class `Phalcon\Cache\Backend`

implements `Phalcon\Cache\BackendInterface`

Allows to cache output fragments, PHP data or raw data to a redis backend

This adapter uses the special redis key “_PHCR” to store all the keys internally used by the adapter

```
<?php

use Phalcon\Cache\Backend\Redis;
use Phalcon\Cache\Frontend\Data as FrontData;

// Cache data for 2 days
$frontCache = new FrontData(
    [
        "lifetime" => 172800,
    ]
);

// Create the Cache setting redis connection options
$cache = new Redis(
    $frontCache,
    [
        "host"      => "localhost",
        "port"      => 6379,
        "auth"      => "foobared",
        "persistent" => false,
        "index"     => 0,
    ]
);

// Cache arbitrary data
$cache->save("my-data", [1, 2, 3, 4, 5]);

// Get data
$data = $cache->get("my-data");
```

Methods

public **__construct** (*Phalcon\Cache\FrontendInterface* \$frontend, [array \$options])

Phalcon\Cache\Backend\Redis constructor

public **_connect** ()

Create internal connection to redis

public **get** (*mixed* \$keyName, [*mixed* \$lifetime])

Returns a cached content

public **save** ([*int* | *string* \$keyName], [*string* \$content], [*int* \$lifetime], [*boolean* \$stopBuffer])

Stores cached content into the file backend and stops the frontend

```
<?php

$cache->save("my-key", $data);

// Save data termlessly
$cache->save("my-key", $data, -1);
```

public **delete** (*int* | *string* \$keyName)

Deletes a value from the cache by its key

public **queryKeys** (*mixed* \$prefix)

Query the existing cached keys.

```
<?php

$cache->save("users-ids", [1, 2, 3]);
$cache->save("projects-ids", [4, 5, 6]);

var_dump($cache->queryKeys("users")); // ["users-ids"]
```

public **exists** (*string* \$keyName), (*int* \$lifetime)

Checks if cache exists and it isn't expired

public **increment** (*string* \$keyName), (*mixed* \$value)

Increment of given \$keyName by \$value

public **decrement** (*string* \$keyName), (*mixed* \$value)

Decrement of \$keyName by given \$value

public **flush** ()

Immediately invalidates all existing items.

public **getFrontend** () inherited from *Phalcon\Cache\Backend*

...

public **setFrontend** (*mixed* \$frontend) inherited from *Phalcon\Cache\Backend*

...

public **getOptions** () inherited from *Phalcon\Cache\Backend*

...

public **setOptions** (*mixed* \$options) inherited from *Phalcon\Cache\Backend*

...

public **getLastKey** () inherited from *Phalcon\Cache\Backend*

...

public **setLastKey** (*mixed* \$lastKey) inherited from *Phalcon\Cache\Backend*

...

public *mixed* **start** (*int* | *string* \$keyName, (*int* \$lifetime)) inherited from *Phalcon\Cache\Backend*

Starts a cache. The keyname allows to identify the created fragment

public **stop** (*mixed* \$stopBuffer) inherited from *Phalcon\Cache\Backend*

Stops the frontend without store any cached content

public **isFresh** () inherited from *Phalcon\Cache\Backend*

Checks whether the last cache is fresh or cached

public **isStarted** () inherited from *Phalcon\Cache\Backend*

Checks whether the cache has starting buffering or not

public **int getLifetime** () inherited from *Phalcon\Cache\Backend*

Gets the last lifetime set

Class *Phalcon\Cache\Backend\Xcache*

extends abstract class *Phalcon\Cache\Backend*

implements *Phalcon\Cache\BackendInterface*

Allows to cache output fragments, PHP data and raw data using an XCache backend

```
<?php

use Phalcon\Cache\Backend\Xcache;
use Phalcon\Cache\Frontend\Data as FrontData;

// Cache data for 2 days
$frontCache = new FrontData(
    [
        "lifetime" => 172800,
    ]
);

$cache = new Xcache(
    $frontCache,
    [
        "prefix" => "app-data",
    ]
);

// Cache arbitrary data
$cache->save("my-data", [1, 2, 3, 4, 5]);

// Get data
$data = $cache->get("my-data");
```

Methods

public **__construct** (*Phalcon\Cache\FrontendInterface* \$frontend, [array \$options])

Phalcon\Cache\Backend\Xcache constructor

public **get** (*mixed* \$keyName, [*mixed* \$lifetime])

Returns a cached content

public **save** ([*int* | *string* \$keyName], [*string* \$content], [*int* \$lifetime], [*boolean* \$stopBuffer])

Stores cached content into the file backend and stops the frontend

public *boolean* **delete** (*int* | *string* \$keyName)

Deletes a value from the cache by its key

public **queryKeys** (*mixed* \$prefix)

Query the existing cached keys.

```
<?php
$cache->save("users-ids", [1, 2, 3]);
$cache->save("projects-ids", [4, 5, 6]);

var_dump($cache->queryKeys("users")); // ["users-ids"]
```

public **exists** (*string* \$keyName, [*int* \$lifetime])

Checks if cache exists and it isn't expired

public **increment** (*string* \$keyName, [*mixed* \$value])

Atomic increment of a given key, by number \$value

public **decrement** (*string* \$keyName, [*mixed* \$value])

Atomic decrement of a given key, by number \$value

public **flush** ()

Immediately invalidates all existing items.

public **getFrontend** () inherited from *Phalcon\Cache\Backend*

...

public **setFrontend** (*mixed* \$frontend) inherited from *Phalcon\Cache\Backend*

...

public **getOptions** () inherited from *Phalcon\Cache\Backend*

...

public **setOptions** (*mixed* \$options) inherited from *Phalcon\Cache\Backend*

...

public **getLastKey** () inherited from *Phalcon\Cache\Backend*

...

public **setLastKey** (*mixed* \$lastKey) inherited from *Phalcon\Cache\Backend*

...

public *mixed* **start** (*int* | *string* \$keyName, [*int* \$lifetime]) inherited from *Phalcon\Cache\Backend*

Starts a cache. The keyname allows to identify the created fragment

public **stop** ([*mixed* \$stopBuffer]) inherited from *Phalcon\Cache\Backend*

Stops the frontend without store any cached content

public **isFresh** () inherited from *Phalcon\Cache\Backend*

Checks whether the last cache is fresh or cached

public **isStarted** () inherited from *Phalcon\Cache\Backend*

Checks whether the cache has starting buffering or not

public **int** **getLifetime** () inherited from *Phalcon\Cache\Backend*

Gets the last lifetime set

Class **Phalcon\Cache\Exception**

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private **Exception** **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public **int** **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public **int** **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Class **Phalcon\Cache\Frontend\Base64**

implements *Phalcon\Cache\FrontendInterface*

Allows to cache data converting/deconverting them to base64.

This adapter uses the base64_encode/base64_decode PHP's functions

```
<?php

<?php

// Cache the files for 2 days using a Base64 frontend
$frontCache = new \Phalcon\Cache\Frontend\Base64 (
    [
        "lifetime" => 172800,
    ]
);

//Create a MongoDB cache
$cache = new \Phalcon\Cache\Backend\Mongo(
    $frontCache,
    [
        "server"      => "mongodb://localhost",
        "db"           => "caches",
        "collection" => "images",
    ]
);

$cacheKey = "some-image.jpg.cache";

// Try to get cached image
$image = $cache->get($cacheKey);

if ($image === null) {
    // Store the image in the cache
    $cache->save(
        $cacheKey,
        file_get_contents("tmp-dir/some-image.jpg")
    );
}

header("Content-Type: image/jpeg");

echo $image;
```

Methods

public **__construct** ([array \$frontendOptions])

Phalcon\Cache\Frontend\Base64 constructor

public **getLifetime** ()

Returns the cache lifetime

public **isBuffering** ()

Check whether if frontend is buffering output

public **start** ()

Starts output frontend. Actually, does nothing in this adapter

public *string* **getContent** ()

Returns output cached content

public **stop** ()

Stops output frontend

public **beforeStore** (*mixed* \$data)

Serializes data before storing them

public **afterRetrieve** (*mixed* \$data)

Unserializes data after retrieval

Class Phalcon\Cache\Frontend\Data

implements Phalcon\Cache\FrontendInterface

Allows to cache native PHP data in a serialized form

```
<?php

use Phalcon\Cache\Backend\File;
use Phalcon\Cache\Frontend\Data;

// Cache the files for 2 days using a Data frontend
$frontCache = new Data(
    [
        "lifetime" => 172800,
    ]
);

// Create the component that will cache "Data" to a 'File' backend
// Set the cache file directory - important to keep the '/' at the end of
// of the value for the folder
$cache = new File(
    $frontCache,
    [
        "cacheDir" => "../app/cache/",
    ]
);

$cacheKey = "robots_order_id.cache";

// Try to get cached records
$robots = $cache->get($cacheKey);

if ($robots === null) {
    // $robots is null due to cache expiration or data does not exist
    // Make the database call and populate the variable
    $robots = Robots::find(
        [
            "order" => "id",
        ]
    );

    // Store it in the cache
    $cache->save($cacheKey, $robots);
}

// Use $robots :)
foreach ($robots as $robot) {
```

```
    echo $robot->name, "\n";  
}
```

Methods

public **__construct** ([array \$frontendOptions])

Phalcon\Cache\Frontend\Data constructor

public **getLifetime** ()

Returns the cache lifetime

public **isBuffering** ()

Check whether if frontend is buffering output

public **start** ()

Starts output frontend. Actually, does nothing

public *string* **getContent** ()

Returns output cached content

public **stop** ()

Stops output frontend

public **beforeStore** (*mixed* \$data)

Serializes data before storing them

public **afterRetrieve** (*mixed* \$data)

Unserializes data after retrieval

Class Phalcon\Cache\Frontend\Igbinary

extends class *Phalcon\Cache\Frontend\Data*

implements *Phalcon\Cache\FrontendInterface*

Allows to cache native PHP data in a serialized form using igbinary extension

```
<?php  
  
// Cache the files for 2 days using Igbinary frontend  
$frontCache = new \Phalcon\Cache\Frontend\Igbinary(  
    [  
        "lifetime" => 172800,  
    ]  
);  
  
// Create the component that will cache "Igbinary" to a "File" backend  
// Set the cache file directory - important to keep the "/" at the end of  
// of the value for the folder  
$cache = new \Phalcon\Cache\Backend\File(  
    $frontCache,  
    [  
        "cacheDir" => "../app/cache/",  
    ]  
);
```

```

    ]
);

$cacheKey = "robots_order_id.cache";

// Try to get cached records
$robots = $cache->get($cacheKey);

if ($robots === null) {
    // $robots is null due to cache expiration or data do not exist
    // Make the database call and populate the variable
    $robots = Robots::find(
        [
            "order" => "id",
        ]
    );

    // Store it in the cache
    $cache->save($cacheKey, $robots);
}

// Use $robots :)
foreach ($robots as $robot) {
    echo $robot->name, "\n";
}

```

Methods

public **__construct** ([array \$frontendOptions])

Phalcon\Cache\Frontend\Data constructor

public **getLifetime** ()

Returns the cache lifetime

public **isBuffering** ()

Check whether if frontend is buffering output

public **start** ()

Starts output frontend. Actually, does nothing

public *string* **getContent** ()

Returns output cached content

public **stop** ()

Stops output frontend

public **beforeStore** (*mixed* \$data)

Serializes data before storing them

public **afterRetrieve** (*mixed* \$data)

Unserializes data after retrieval

Class `Phalcon\Cache\Frontend\Json`

implements `Phalcon\Cache\FrontendInterface`

Allows to cache data converting/deconverting them to JSON.

This adapter uses the `json_encode/json_decode` PHP's functions

As the data is encoded in JSON other systems accessing the same backend could process them

```
<?php

<?php

// Cache the data for 2 days
$frontCache = new \Phalcon\Cache\Frontend\Json(
    [
        "lifetime" => 172800,
    ]
);

// Create the Cache setting memcached connection options
$cache = new \Phalcon\Cache\Backend\Memcache(
    $frontCache,
    [
        "host"      => "localhost",
        "port"      => 11211,
        "persistent" => false,
    ]
);

// Cache arbitrary data
$cache->save("my-data", [1, 2, 3, 4, 5]);

// Get data
$data = $cache->get("my-data");
```

Methods

public **__construct** ([array \$frontendOptions])

Phalcon\Cache\Frontend\Base64 constructor

public **getLifetime** ()

Returns the cache lifetime

public **isBuffering** ()

Check whether if frontend is buffering output

public **start** ()

Starts output frontend. Actually, does nothing

public *string* **getContent** ()

Returns output cached content

public **stop** ()

Stops output frontend

public **beforeStore** (*mixed* \$data)

Serializes data before storing them

public **afterRetrieve** (*mixed* \$data)

Unserializes data after retrieval

Class Phalcon\Cache\Frontend\Msgpack

extends class Phalcon\Cache\Frontend\Data

implements Phalcon\Cache\FrontendInterface

Allows to cache native PHP data in a serialized form using msgpack extension This adapter uses a Msgpack frontend to store the cached content and requires msgpack extension.

```
<?php

use Phalcon\Cache\Backend\File;
use Phalcon\Cache\Frontend\Msgpack;

// Cache the files for 2 days using Msgpack frontend
$frontCache = new Msgpack(
    [
        "lifetime" => 172800,
    ]
);

// Create the component that will cache "Msgpack" to a "File" backend
// Set the cache file directory - important to keep the "/" at the end of
// of the value for the folder
$cache = new File(
    $frontCache,
    [
        "cacheDir" => "../app/cache/",
    ]
);

$cacheKey = "robots_order_id.cache";

// Try to get cached records
$robots = $cache->get($cacheKey);

if ($robots === null) {
    // $robots is null due to cache expiration or data do not exist
    // Make the database call and populate the variable
    $robots = Robots::find(
        [
            "order" => "id",
        ]
    );

    // Store it in the cache
    $cache->save($cacheKey, $robots);
}

// Use $robots
foreach ($robots as $robot) {
```

```
    echo $robot->name, "\n";
}
```

Methods

public **__construct** ([array \$frontendOptions])

Phalcon\Cache\Frontend\Msgpack constructor

public **getLifetime** ()

Returns the cache lifetime

public **isBuffering** ()

Check whether if frontend is buffering output

public **start** ()

Starts output frontend. Actually, does nothing

public **getContent** ()

Returns output cached content

public **stop** ()

Stops output frontend

public **beforeStore** (mixed \$data)

Serializes data before storing them

public **afterRetrieve** (mixed \$data)

Unserializes data after retrieval

Class Phalcon\Cache\Frontend\None

implements *Phalcon\Cache\FrontendInterface*

Discards any kind of frontend data input. This frontend does not have expiration time or any other options

```
<?php

<?php

//Create a None Cache
$frontCache = new \Phalcon\Cache\Frontend\None();

// Create the component that will cache "Data" to a "Memcached" backend
// Memcached connection settings
$cache = new \Phalcon\Cache\Backend\Memcache(
    $frontCache,
    [
        "host" => "localhost",
        "port" => "11211",
    ]
);

$cacheKey = "robots_order_id.cache";
```

```
// This Frontend always return the data as it's returned by the backend
$robots = $cache->get($cacheKey);

if ($robots === null) {
    // This cache doesn't perform any expiration checking, so the data is always_
    ↪expired
    // Make the database call and populate the variable
    $robots = Robots::find(
        [
            "order" => "id",
        ]
    );

    $cache->save($cacheKey, $robots);
}

// Use $robots :)
foreach ($robots as $robot) {
    echo $robot->name, "\n";
}
```

Methods

public **getLifetime** ()

Returns cache lifetime, always one second expiring content

public **isBuffering** ()

Check whether if frontend is buffering output, always false

public **start** ()

Starts output frontend

public *string* **getContent** ()

Returns output cached content

public **stop** ()

Stops output frontend

public **beforeStore** (*mixed* \$data)

Prepare data to be stored

public **afterRetrieve** (*mixed* \$data)

Prepares data to be retrieved to user

Class **Phalcon\Cache\Frontend\Output**

implements *Phalcon\Cache\FrontendInterface*

Allows to cache output fragments captured with ob_* functions

```
<?php

*
* use Phalcon\Tag;
* use Phalcon\Cache\Backend\File;
* use Phalcon\Cache\Frontend\Output;
*
* // Create an Output frontend. Cache the files for 2 days
* $frontCache = new Output(
*     [
*         "lifetime" => 172800,
*     ]
* );
*
* // Create the component that will cache from the "Output" to a "File" backend
* // Set the cache file directory - it's important to keep the "/" at the end of
* // the value for the folder
* $cache = new File(
*     $frontCache,
*     [
*         "cacheDir" => "../app/cache/",
*     ]
* );
*
* // Get/Set the cache file to ../app/cache/my-cache.html
* $content = $cache->start("my-cache.html");
*
* // If $content is null then the content will be generated for the cache
* if (null === $content) {
*     // Print date and time
*     echo date("r");
*
*     // Generate a link to the sign-up action
*     echo Tag::linkTo(
*         [
*             "user/signup",
*             "Sign Up",
*             "class" => "signup-button",
*         ]
*     );
*
*     // Store the output into the cache file
*     $cache->save();
* } else {
*     // Echo the cached output
*     echo $content;
* }
```

•

Methods

public **__construct** ([array \$frontendOptions])

Phalcon\Cache\Frontend\Output constructor

public **getLifetime** ()

Returns the cache lifetime

public **isBuffering** ()

Check whether if frontend is buffering output

public **start** ()

Starts output frontend. Currently, does nothing

public *string* **getContent** ()

Returns output cached content

public **stop** ()

Stops output frontend

public **beforeStore** (*mixed* \$data)

Serializes data before storing them

public **afterRetrieve** (*mixed* \$data)

Unserializes data after retrieval

Class Phalcon\Cache\Multiple

Allows to read to chained backend adapters writing to multiple backends

```
<?php

use Phalcon\Cache\Frontend\Data as DataFrontend;
use Phalcon\Cache\Multiple;
use Phalcon\Cache\Backend\Apc as ApcCache;
use Phalcon\Cache\Backend\Memcache as MemcacheCache;
use Phalcon\Cache\Backend\File as FileCache;

$ultraFastFrontend = new DataFrontend(
    [
        "lifetime" => 3600,
    ]
);

$fastFrontend = new DataFrontend(
    [
        "lifetime" => 86400,
    ]
);

$slowFrontend = new DataFrontend(
    [
        "lifetime" => 604800,
    ]
);

//Backends are registered from the fastest to the slower
$cache = new Multiple(
    [
        new ApcCache(
            $ultraFastFrontend,
            [
```

```
        "prefix" => "cache",
    ],
),
new MemcacheCache(
    $fastFrontend,
    [
        "prefix" => "cache",
        "host"    => "localhost",
        "port"    => "11211",
    ]
),
new FileCache(
    $slowFrontend,
    [
        "prefix"    => "cache",
        "cacheDir" => "../app/cache/",
    ]
),
];

//Save, saves in every backend
$cache->save("my-key", $data);
```

Methods

public **__construct** ([*Phalcon\Cache\BackendInterface*[] \$backends])

Phalcon\Cache\Multiple constructor

public **push** (*Phalcon\Cache\BackendInterface* \$backend)

Adds a backend

public *mixed* **get** (*string* | *int* \$keyName, [*int* \$lifetime])

Returns a cached content reading the internal backends

public **start** (*string* | *int* \$keyName, [*int* \$lifetime])

Starts every backend

public **save** ([*string* \$keyName], [*string* \$content], [*int* \$lifetime], [*boolean* \$stopBuffer])

Stores cached content into all backends and stops the frontend

public *boolean* **delete** (*string* | *int* \$keyName)

Deletes a value from each backend

public **exists** ([*string* | *int* \$keyName], [*int* \$lifetime])

Checks if cache exists in at least one backend

public **flush** ()

Flush all backend(s)

Class *Phalcon\Cli\Console*

extends abstract class *Phalcon\Application*

implements *Phalcon\Di\InjectionAwareInterface*, *Phalcon\Events\EventsAwareInterface*

This component allows to create CLI applications using Phalcon

Methods

public **addModules** (array \$modules)

Merge modules with the existing ones

```
<?php

$application->addModules (
    [
        "admin" => [
            "className" => "Multiple\\Admin\\Module",
            "path"       => "../apps/admin/Module.php",
        ],
    ]
);
```

public **handle** ([array \$arguments])

Handle the whole command-line tasks

public **setArgument** ([array \$arguments], [mixed \$str], [mixed \$shift])

Set an specific argument

public **__construct** ([*Phalcon\DiInterface* \$dependencyInjector]) inherited from *Phalcon\Application*

Phalcon\Application

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Application*

Sets the events manager

public **getEventManager** () inherited from *Phalcon\Application*

Returns the internal event manager

public **registerModules** (array \$modules, [mixed \$merge]) inherited from *Phalcon\Application*

Register an array of modules present in the application

```
<?php

$this->registerModules (
    [
        "frontend" => [
            "className" => "Multiple\\Frontend\\Module",
            "path"       => "../apps/frontend/Module.php",
        ],
        "backend" => [
            "className" => "Multiple\\Backend\\Module",
            "path"       => "../apps/backend/Module.php",
        ],
    ]
);
```

public **getModules** () inherited from *Phalcon\Application*

Return the modules registered in the application

public **getModule** (*mixed* \$name) inherited from *Phalcon\Application*

Gets the module definition registered in the application via module name

public **setDefaultModule** (*mixed* \$defaultModule) inherited from *Phalcon\Application*

Sets the module name to be used if the router doesn't return a valid module

public **getDefaultModule** () inherited from *Phalcon\Application*

Returns the default module name

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di\Injectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method __get

Class *Phalcon\Cli\Console\Exception*

extends class *Phalcon\Application\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public **Exception** **getTraceAsString** () inherited from **Exception**

Gets the stack trace as a string

public *string* **__toString** () inherited from **Exception**

String representation of the exception

Class **Phalcon\Cli\Dispatcher**

extends abstract class *Phalcon\Dispatcher*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\Di\InjectionAwareInterface*, *Phalcon\DispatcherInterface*, *Phalcon\Cli\DispatcherInterface*

Dispatching is the process of taking the command-line arguments, extracting the module name, task name, action name, and optional parameters contained in it, and then instantiating a task and calling an action on it.

```
<?php

$di = new \Phalcon\Di();

$dispatcher = new \Phalcon\Cli\Dispatcher();

$dispatcher->setDi($di);

$dispatcher->setTaskName("posts");
$dispatcher->setActionName("index");
$dispatcher->setParams([]);

$handle = $dispatcher->dispatch();
```

Constants

integer **EXCEPTION_NO_DI**

integer **EXCEPTION_CYCLIC_ROUTING**

integer **EXCEPTION_HANDLER_NOT_FOUND**

integer **EXCEPTION_INVALID_HANDLER**

integer **EXCEPTION_INVALID_PARAMS**

integer **EXCEPTION_ACTION_NOT_FOUND**

Methods

public **setTaskSuffix** (*mixed* \$taskSuffix)

Sets the default task suffix

public **setDefaultTask** (*mixed* \$taskName)

Sets the default task name

public **setTaskName** (*mixed* \$taskName)

Sets the task name to be dispatched

public **getTaskName** ()

Gets last dispatched task name

protected **_throwDispatchException** (*mixed* \$message, [*mixed* \$exceptionCode])

Throws an internal exception

protected **_handleException** (*Exception* \$exception)

Handles a user exception

public **getLastTask** ()

Returns the latest dispatched controller

public **getActiveTask** ()

Returns the active task in the dispatcher

public **setOptions** (*array* \$options)

Set the options to be dispatched

public **getOptions** ()

Get dispatched options

public **callActionMethod** (*mixed* \$handler, *mixed* \$actionMethod, [*array* \$params])

...

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Dispatcher*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Dispatcher*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Dispatcher*

Sets the events manager

public **getEventManager** () inherited from *Phalcon\Dispatcher*

Returns the internal event manager

public **setActionSuffix** (*mixed* \$actionSuffix) inherited from *Phalcon\Dispatcher*

Sets the default action suffix

public **getActionSuffix** () inherited from *Phalcon\Dispatcher*

Gets the default action suffix

public **setModuleName** (*mixed* \$moduleName) inherited from *Phalcon\Dispatcher*

Sets the module where the controller is (only informative)

public **getModuleName** () inherited from *Phalcon\Dispatcher*

Gets the module where the controller class is

public **setNamespaceName** (*mixed* \$namespaceName) inherited from *Phalcon\Dispatcher*

Sets the namespace where the controller class is

public **getNamespaceName** () inherited from *Phalcon\Dispatcher*

Gets a namespace to be prepended to the current handler name

public **setDefaultNamespace** (*mixed* \$namespaceName) inherited from *Phalcon\Dispatcher*

Sets the default namespace

public **getDefaultNamespace** () inherited from *Phalcon\Dispatcher*

Returns the default namespace

public **setDefaultAction** (*mixed* \$actionName) inherited from *Phalcon\Dispatcher*

Sets the default action name

public **setActionName** (*mixed* \$actionName) inherited from *Phalcon\Dispatcher*

Sets the action name to be dispatched

public **getActionName** () inherited from *Phalcon\Dispatcher*

Gets the latest dispatched action name

public **setParams** (*array* \$params) inherited from *Phalcon\Dispatcher*

Sets action params to be dispatched

public **getParams** () inherited from *Phalcon\Dispatcher*

Gets action params

public **setParam** (*mixed* \$param, *mixed* \$value) inherited from *Phalcon\Dispatcher*

Set a param by its name or numeric index

public *mixed* **getParam** (*mixed* \$param, [*string* | *array* \$filters], [*mixed* \$defaultValue]) inherited from *Phalcon\Dispatcher*

Gets a param by its name or numeric index

public *boolean* **hasParam** (*mixed* \$param) inherited from *Phalcon\Dispatcher*

Check if a param exists

public **getActiveMethod** () inherited from *Phalcon\Dispatcher*

Returns the current method to be/executed in the dispatcher

public **isFinished** () inherited from *Phalcon\Dispatcher*

Checks if the dispatch loop is finished or has more pendent controllers/tasks to dispatch

public **setReturnedValue** (*mixed* \$value) inherited from *Phalcon\Dispatcher*

Sets the latest returned value by an action manually

public *mixed* **getReturnedValue** () inherited from *Phalcon\Dispatcher*

Returns value returned by the latest dispatched action

public **setModelBinding** (*mixed* \$value, [*mixed* \$cache]) inherited from *Phalcon\Dispatcher*

Enable/Disable model binding during dispatch

```
<?php
$di->set('dispatcher', function() {
    $dispatcher = new Dispatcher();

    $dispatcher->setModelBinding(true, 'cache');
```

```
        return $dispatcher;
    });
```

public **setModelBinder** (Phalcon\Mvc\Model\BinderInterface \$modelBinder, [*mixed* \$cache]) inherited from *Phalcon\Dispatcher*

Enable model binding during dispatch

```
<?php

$di->set('dispatcher', function() {
    $dispatcher = new Dispatcher();

    $dispatcher->setModelBinder(new Binder(), 'cache');
    return $dispatcher;
});
```

public **getModelBinder** () inherited from *Phalcon\Dispatcher*

Gets model binder

public *object* **dispatch** () inherited from *Phalcon\Dispatcher*

Dispatches a handle action taking into account the routing parameters

protected *object* **_dispatch** () inherited from *Phalcon\Dispatcher*

Dispatches a handle action taking into account the routing parameters

public **forward** (array \$forward) inherited from *Phalcon\Dispatcher*

Forwards the execution flow to another controller/action Dispatchers are unique per module. Forwarding between modules is not allowed

```
<?php

$this->dispatcher->forward(
    [
        "controller" => "posts",
        "action"      => "index",
    ]
);
```

public **wasForwarded** () inherited from *Phalcon\Dispatcher*

Check if the current executed action was forwarded by another one

public **getHandlerClass** () inherited from *Phalcon\Dispatcher*

Possible class name that will be located to dispatch the request

public **getBoundModels** () inherited from *Phalcon\Dispatcher*

Returns bound models from binder instance

```
<?php

class UserController extends Controller
{
    public function showAction(User $user)
    {
        $boundModels = $this->dispatcher->getBoundModels(); // return array with $user
    }
}
```



```
}  
}
```

protected **_resolveEmptyProperties** () inherited from *Phalcon\Dispatcher*

Set empty properties to their defaults (where defaults are available)

Class *Phalcon\Cli\Dispatcher\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Class Phalcon\Cli\Router

implements Phalcon\Di\InjectionAwareInterface

Phalcon\Cli\Router is the standard framework router. Routing is the process of taking a command-line arguments and decomposing it into parameters to determine which module, task, and action of that task should receive the request

```
<?php

$router = new \Phalcon\Cli\Router();

$router->handle(
    [
        "module" => "main",
        "task"    => "videos",
        "action"  => "process",
    ]
);

echo $router->getTaskName();
```

Methods

public **__construct** ([*mixed* \$defaultRoutes])

Phalcon\Cli\Router constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injector

public **getDI** ()

Returns the internal dependency injector

public **setDefaultModule** (*mixed* \$moduleName)

Sets the name of the default module

public **setDefaultTask** (*mixed* \$taskName)

Sets the default controller name

public **setDefaultAction** (*mixed* \$actionName)

Sets the default action name

public **setDefaults** (*array* \$defaults)

Sets an array of default paths. If a route is missing a path the router will use the defined here This method must not be used to set a 404 route

```
<?php

$router->setDefaults(
    [
        "module" => "common",
        "action"  => "index",
    ]
);
```

public **handle** ([array \$arguments])

Handles routing information received from command-line arguments

public *Phalcon\Cli\Router\Route* **add** (string \$pattern, [string/array \$paths])

Adds a route to the router

```
<?php
$router->add("/about", "About::main");
```

public **getModuleName** ()

Returns processed module name

public **getTaskName** ()

Returns processed task name

public **getActionName** ()

Returns processed action name

public array **getParams** ()

Returns processed extra params

public **getMatchedRoute** ()

Returns the route that matches the handled URI

public array **getMatches** ()

Returns the sub expressions in the regular expression matched

public **wasMatched** ()

Checks if the router matches any of the defined routes

public **getRoutes** ()

Returns all the routes defined in the router

public *Phalcon\Cli\Router\Route* **getRouteById** (int \$id)

Returns a route object by its id

public **getRouteByName** (mixed \$name)

Returns a route object by its name

Class *Phalcon\Cli\Router\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([string \$message], [int \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from [Exception](#)

...

final public *string* **getMessage** () inherited from [Exception](#)

Gets the Exception message

final public *int* **getCode** () inherited from [Exception](#)

Gets the Exception code

final public *string* **getFile** () inherited from [Exception](#)

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from [Exception](#)

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Cli\Router\Route](#)

This class represents every route added to the router

Constants

string **DEFAULT_DELIMITER**

Methods

public **__construct** (*string* \$pattern, [*array* \$paths])

[Phalcon\Cli\Router\Route](#) constructor

public **compilePattern** (*mixed* \$pattern)

Replaces placeholders from pattern returning a valid PCRE regular expression

public *array* | *boolean* **extractNamedParams** (*string* \$pattern)

Extracts parameters from a string

public **reConfigure** (*string* \$pattern, [*array* \$paths])

Reconfigure the route adding a new pattern and a set of paths

public **getName** ()

Returns the route's name

public **setName** (*mixed* \$name)

Sets the route's name

```
<?php
$router->add(
    "/about",
    [
        "controller" => "about",
    ]
)->setName("about");
```

public *Phalcon\CLI\Router\Route* **beforeMatch** (*callback* \$callback)

Sets a callback that is called if the route is matched. The developer can implement any arbitrary conditions here. If the callback returns false the route is treated as not matched.

public *mixed* **getBeforeMatch** ()

Returns the 'before match' callback if any

public **getRouteId** ()

Returns the route's id

public **getPattern** ()

Returns the route's pattern

public **getCompiledPattern** ()

Returns the route's compiled pattern

public **getPaths** ()

Returns the paths

public **getReversedPaths** ()

Returns the paths using positions as keys and names as values

public *Phalcon\CLI\Router\Route* **convert** (*string* \$name, *callable* \$converter)

Adds a converter to perform an additional transformation for certain parameter

public **getConverters** ()

Returns the router converter

public static **reset** ()

Resets the internal route id generator

public static **delimiter** ([*mixed* \$delimiter])

Set the routing delimiter

public static **getDelimiter** ()

Get routing delimiter

Class Phalcon\Cli\Task

extends abstract class *Phalcon\Di\Injectable*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\Di\InjectionAwareInterface*, *Phalcon\Cli\TaskInterface*

Every command-line task should extend this class that encapsulates all the task functionality

A task can be used to run “tasks” such as migrations, cronjobs, unit-tests, or anything that you want. The Task class should at least have a “mainAction” method

```
<?php

class HelloTask extends \Phalcon\Cli\Task
{
    // This action will be executed by default
    public function mainAction()
    {

    }

    public function findAction()
    {

    }
}
```

Methods

final public **__construct** ()

Phalcon\Cli\Task constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di\Injectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Di\Injectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Di\Injectable*

Returns the internal event manager

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method __get

Class Phalcon\Config

implements *ArrayAccess*, *Countable*

Phalcon\Config is designed to simplify the access to, and the use of, configuration data within applications. It provides a nested object property based user interface for accessing this configuration data within application code.

```
<?php

$config = new \Phalcon\Config(
    [
        "database" => [
            "adapter" => "Mysql",
            "host"     => "localhost",
            "username" => "scott",
            "password" => "cheetah",
            "dbname"   => "test_db",
        ],
        "phalcon" => [
            "controllersDir" => "../app/controllers/",
            "modelsDir"      => "../app/models/",
            "viewsDir"       => "../app/views/",
        ],
    ],
);
```

Methods

public **__construct** ([array \$arrayConfig])

Phalcon\Config constructor

public **offsetExists** (mixed \$index)

Allows to check whether an attribute is defined using the array-syntax

```
<?php

var_dump(
    isset($config["database"])
);
```

public **get** (mixed \$index, [mixed \$defaultValue])

Gets an attribute from the configuration, if the attribute isn't defined returns null If the value is exactly null or is not defined the default value will be used instead

```
<?php

echo $config->get("controllersDir", "../app/controllers/");
```

public **offsetGet** (mixed \$index)

Gets an attribute using the array-syntax

```
<?php

print_r(
    $config["database"]
);
```

public **offsetSet** (mixed \$index, mixed \$value)

Sets an attribute using the array-syntax

```
<?php

$config["database"] = [
    "type" => "Sqlite",
];
```

public **offsetUnset** (*mixed* \$index)

Unsets an attribute using the array-syntax

```
<?php

unset($config["database"]);
```

public **merge** (*Phalcon\Config* \$config)

Merges a configuration into the current one

```
<?php

$appConfig = new \Phalcon\Config(
    [
        "database" => [
            "host" => "localhost",
        ],
    ]
);

$globalConfig->merge($appConfig);
```

public **toArray** ()

Converts recursively the object to an array

```
<?php

print_r(
    $config->toArray()
);
```

public **count** ()

Returns the count of properties set in the config

```
<?php

print count($config);
```

or

```
<?php

print $config->count();
```

public static **__set_state** (array \$data)

Restores the state of a *Phalcon\Config* object

final protected *Config merged config* **_merge** (*Config* \$config, [*mixed* \$instance])

Helper method for merge configs (forwarding nested config instance)

Class Phalcon\Config\Adapter\Ini

extends class *Phalcon\Config*

implements *Countable*, *ArrayAccess*

Reads ini files and converts them to Phalcon\Config objects.

Given the next configuration file:

```
<?php

[database]
adapter = Mysql
host = localhost
username = scott
password = cheetah
dbname = test_db

[phalcon]
controllersDir = "../app/controllers/"
modelsDir = "../app/models/"
viewsDir = "../app/views/"
```

You can read it as follows:

```
<?php

$config = new \Phalcon\Config\Adapter\Ini("path/config.ini");

echo $config->phalcon->controllersDir;
echo $config->database->username;
```

PHP constants may also be parsed in the ini file, so if you define a constant as an ini value before calling the constructor, the constant's value will be integrated into the results. To use it this way you must specify the optional second parameter as `INI_SCANNER_NORMAL` when calling the constructor:

```
<?php

$config = new \Phalcon\Config\Adapter\Ini(
    "path/config-with-constants.ini",
    INI_SCANNER_NORMAL
);
```

Methods

public **__construct** (*mixed* \$filePath, [*mixed* \$mode])

Phalcon\Config\Adapter\Ini constructor

protected **_parseIniString** (*mixed* \$path, *mixed* \$value)

Build multidimensional array from string

```
<?php

$this->_parseIniString("path.hello.world", "value for last key");
```

```
// result
[
    "path" => [
        "hello" => [
            "world" => "value for last key",
        ],
    ],
];
```

protected **_cast** (*mixed* \$ini)

We have to cast values manually because `parse_ini_file()` has a poor implementation.

public **offsetExists** (*mixed* \$index) inherited from *Phalcon\Config*

Allows to check whether an attribute is defined using the array-syntax

```
<?php
var_dump(
    isset($config["database"])
);
```

public **get** (*mixed* \$index, [*mixed* \$defaultValue]) inherited from *Phalcon\Config*

Gets an attribute from the configuration, if the attribute isn't defined returns null If the value is exactly null or is not defined the default value will be used instead

```
<?php
echo $config->get("controllersDir", "../app/controllers/");
```

public **offsetGet** (*mixed* \$index) inherited from *Phalcon\Config*

Gets an attribute using the array-syntax

```
<?php
print_r(
    $config["database"]
);
```

public **offsetSet** (*mixed* \$index, *mixed* \$value) inherited from *Phalcon\Config*

Sets an attribute using the array-syntax

```
<?php
$config["database"] = [
    "type" => "Sqlite",
];
```

public **offsetUnset** (*mixed* \$index) inherited from *Phalcon\Config*

Unsets an attribute using the array-syntax

```
<?php
unset($config["database"]);
```

public **merge** (*Phalcon\Config* \$config) inherited from *Phalcon\Config*

Merges a configuration into the current one

```
<?php
$appConfig = new \Phalcon\Config(
    [
        "database" => [
            "host" => "localhost",
        ],
    ]
);

$globalConfig->merge($appConfig);
```

public **toArray** () inherited from *Phalcon\Config*

Converts recursively the object to an array

```
<?php
print_r(
    $config->toArray()
);
```

public **count** () inherited from *Phalcon\Config*

Returns the count of properties set in the config

```
<?php
print count($config);
```

or

```
<?php
print $config->count();
```

public static **__set_state** (array \$data) inherited from *Phalcon\Config*

Restores the state of a *Phalcon\Config* object

final protected *Config merged config* **_merge** (*Config* \$config, [*mixed* \$instance]) inherited from *Phalcon\Config*

Helper method for merge configs (forwarding nested config instance)

Class *Phalcon\Config\Adapter\Json*

extends class *Phalcon\Config*

implements *Countable*, *ArrayAccess*

Reads JSON files and converts them to *Phalcon\Config* objects.

Given the following configuration file:

```
<?php
{"phalcon":{"baseuri":"\\/phalcon\\/"},"models":{"metadata":"memory"}}
```

You can read it as follows:

```
<?php

$config = new Phalcon\Config\Adapter\Json("path/config.json");

echo $config->phalcon->baseuri;
echo $config->models->metadata;
```

Methods

public **__construct** (*mixed* \$filePath)

Phalcon\Config\Adapter\Json constructor

public **offsetExists** (*mixed* \$index) inherited from *Phalcon\Config*

Allows to check whether an attribute is defined using the array-syntax

```
<?php

var_dump(
    isset($config["database"])
);
```

public **get** (*mixed* \$index, [*mixed* \$defaultValue]) inherited from *Phalcon\Config*

Gets an attribute from the configuration, if the attribute isn't defined returns null If the value is exactly null or is not defined the default value will be used instead

```
<?php

echo $config->get("controllersDir", "../app/controllers/");
```

public **offsetGet** (*mixed* \$index) inherited from *Phalcon\Config*

Gets an attribute using the array-syntax

```
<?php

print_r(
    $config["database"]
);
```

public **offsetSet** (*mixed* \$index, *mixed* \$value) inherited from *Phalcon\Config*

Sets an attribute using the array-syntax

```
<?php

$config["database"] = [
    "type" => "Sqlite",
];
```

public **offsetUnset** (*mixed* \$index) inherited from *Phalcon\Config*

Unsets an attribute using the array-syntax

```
<?php
unset($config["database"]);
```

public **merge** (*Phalcon\Config* \$config) inherited from *Phalcon\Config*

Merges a configuration into the current one

```
<?php

$appConfig = new \Phalcon\Config(
    [
        "database" => [
            "host" => "localhost",
        ],
    ],
);

$globalConfig->merge($appConfig);
```

public **toArray** () inherited from *Phalcon\Config*

Converts recursively the object to an array

```
<?php

print_r(
    $config->toArray()
);
```

public **count** () inherited from *Phalcon\Config*

Returns the count of properties set in the config

```
<?php

print count($config);
```

or

```
<?php

print $config->count();
```

public static **__set_state** (array \$data) inherited from *Phalcon\Config*

Restores the state of a *Phalcon\Config* object

final protected *Config merged config* **_merge** (*Config* \$config, [*mixed* \$instance]) inherited from *Phalcon\Config*

Helper method for merge configs (forwarding nested config instance)

Class *Phalcon\Config\Adapter\Php*

extends class *Phalcon\Config*

implements *Countable*, *ArrayAccess*

Reads php files and converts them to *Phalcon\Config* objects.

Given the next configuration file:

```
<?php

<?php

return [
    "database" => [
        "adapter" => "Mysql",
        "host"     => "localhost",
        "username" => "scott",
        "password" => "cheetah",
        "dbname"  => "test_db",
    ],
    "phalcon" => [
        "controllersDir" => "../app/controllers/",
        "modelsDir"      => "../app/models/",
        "viewsDir"       => "../app/views/",
    ],
];
```

You can read it as follows:

```
<?php

$config = new \Phalcon\Config\Adapter\Php("path/config.php");

echo $config->phalcon->controllersDir;
echo $config->database->username;
```

Methods

public **__construct** (*mixed* \$filePath)

Phalcon\Config\Adapter\Php constructor

public **offsetExists** (*mixed* \$index) inherited from *Phalcon\Config*

Allows to check whether an attribute is defined using the array-syntax

```
<?php

var_dump(
    isset($config["database"])
);
```

public **get** (*mixed* \$index, [*mixed* \$defaultValue]) inherited from *Phalcon\Config*

Gets an attribute from the configuration, if the attribute isn't defined returns null If the value is exactly null or is not defined the default value will be used instead

```
<?php

echo $config->get("controllersDir", "../app/controllers/");
```

public **offsetGet** (*mixed* \$index) inherited from *Phalcon\Config*

Gets an attribute using the array-syntax

```
<?php

print_r(
    $config["database"]
);
```

public **offsetSet** (*mixed* \$index, *mixed* \$value) inherited from *Phalcon\Config*

Sets an attribute using the array-syntax

```
<?php

$config["database"] = [
    "type" => "Sqlite",
];
```

public **offsetUnset** (*mixed* \$index) inherited from *Phalcon\Config*

Unsets an attribute using the array-syntax

```
<?php

unset($config["database"]);
```

public **merge** (*Phalcon\Config* \$config) inherited from *Phalcon\Config*

Merges a configuration into the current one

```
<?php

$appConfig = new \Phalcon\Config(
    [
        "database" => [
            "host" => "localhost",
        ],
    ]
);

$globalConfig->merge($appConfig);
```

public **toArray** () inherited from *Phalcon\Config*

Converts recursively the object to an array

```
<?php

print_r(
    $config->toArray()
);
```

public **count** () inherited from *Phalcon\Config*

Returns the count of properties set in the config

```
<?php

print count($config);
```

or

```
<?php  
  
print $config->count();
```

public static **__set_state** (array \$data) inherited from *Phalcon\Config*

Restores the state of a Phalcon\Config object

final protected *Config merged config* **_merge** (Config \$config, [mixed \$instance]) inherited from *Phalcon\Config*

Helper method for merge configs (forwarding nested config instance)

Class Phalcon\Config\Adapter\Yaml

extends class *Phalcon\Config*

implements Countable, ArrayAccess

Reads YAML files and converts them to Phalcon\Config objects.

Given the following configuration file:

```
<?php  
  
phalcon:  
  baseuri:      /phalcon/  
  controllersDir: !approot /app/controllers/  
models:  
  metadata: memory
```

You can read it as follows:

```
<?php  
  
define(  
    "APPROOT",  
    dirname(__DIR__)  
);  
  
$config = new \Phalcon\Config\Adapter\Yaml(  
    "path/config.yaml",  
    [  
        "!approot" => function($value) {  
            return APPROOT . $value;  
        },  
    ],  
);  
  
echo $config->phalcon->controllersDir;  
echo $config->phalcon->baseuri;  
echo $config->models->metadata;
```

Methods

public **__construct** (mixed \$filePath, [array \$callbacks])

Phalcon\Config\Adapter\Yaml constructor

public **offsetExists** (*mixed* \$index) inherited from *Phalcon\Config*

Allows to check whether an attribute is defined using the array-syntax

```
<?php
var_dump(
    isset($config["database"])
);
```

public **get** (*mixed* \$index, [*mixed* \$defaultValue]) inherited from *Phalcon\Config*

Gets an attribute from the configuration, if the attribute isn't defined returns null If the value is exactly null or is not defined the default value will be used instead

```
<?php
echo $config->get("controllersDir", "../app/controllers/");
```

public **offsetGet** (*mixed* \$index) inherited from *Phalcon\Config*

Gets an attribute using the array-syntax

```
<?php
print_r(
    $config["database"]
);
```

public **offsetSet** (*mixed* \$index, *mixed* \$value) inherited from *Phalcon\Config*

Sets an attribute using the array-syntax

```
<?php
$config["database"] = [
    "type" => "Sqlite",
];
```

public **offsetUnset** (*mixed* \$index) inherited from *Phalcon\Config*

Unsets an attribute using the array-syntax

```
<?php
unset($config["database"]);
```

public **merge** (*Phalcon\Config* \$config) inherited from *Phalcon\Config*

Merges a configuration into the current one

```
<?php
$appConfig = new \Phalcon\Config(
    [
        "database" => [
            "host" => "localhost",
        ],
    ]
);
```

```
$globalConfig->merge($appConfig);
```

public **toArray** () inherited from *Phalcon\Config*

Converts recursively the object to an array

```
<?php  
  
print_r(  
    $config->toArray()  
);
```

public **count** () inherited from *Phalcon\Config*

Returns the count of properties set in the config

```
<?php  
  
print count($config);
```

or

```
<?php  
  
print $config->count();
```

public static **__set_state** (array \$data) inherited from *Phalcon\Config*

Restores the state of a *Phalcon\Config* object

final protected *Config merged config* **_merge** (*Config* \$config, [*mixed* \$instance]) inherited from *Phalcon\Config*

Helper method for merge configs (forwarding nested config instance)

Class *Phalcon\Config\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Class *Phalcon\Crypt*

implements *Phalcon\CryptInterface*

Provides encryption facilities to phalcon applications

```
<?php

$crypt = new \Phalcon\Crypt();

$key   = "le password";
$text  = "This is a secret text";

$encrypted = $crypt->encrypt($text, $key);

echo $crypt->decrypt($encrypted, $key);
```

Constants

integer **PADDING_DEFAULT**

integer **PADDING_ANSI_X_923**

integer **PADDING_PKCS7**

integer **PADDING_ISO_10126**

integer **PADDING_ISO_IEC_7816_4**

integer **PADDING_ZERO**

integer **PADDING_SPACE**

Methods

public **setPadding** (*mixed* \$scheme)

Changes the padding scheme used

public **setCipher** (*mixed* \$cipher)

Sets the cipher algorithm

public **getCipher** ()

Returns the current cipher

public **setKey** (*mixed* \$key)

Sets the encryption key

public **getKey** ()

Returns the encryption key

protected **_cryptPadText** (*mixed* \$text, *mixed* \$mode, *mixed* \$blockSize, *mixed* \$paddingType)

Pads texts before encryption

protected **_cryptUnpadText** (*mixed* \$text, *mixed* \$mode, *mixed* \$blockSize, *mixed* \$paddingType)

If the function detects that the text was not padded, it will return it unmodified

public **encrypt** (*mixed* \$text, [*mixed* \$key])

Encrypts a text

```
<?php
$encrypted = $crypt->encrypt("Ultra-secret text", "encrypt password");
```

public **decrypt** (*mixed* \$text, [*mixed* \$key])

Decrypts an encrypted text

```
<?php
echo $crypt->decrypt($encrypted, "decrypt password");
```

public **encryptBase64** (*mixed* \$text, [*mixed* \$key], [*mixed* \$safe])

Encrypts a text returning the result as a base64 string

public **decryptBase64** (*mixed* \$text, [*mixed* \$key], [*mixed* \$safe])

Decrypt a text that is coded as a base64 string

public **getAvailableCiphers** ()

Returns a list of available ciphers

Class **Phalcon\Crypt\Exception**

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private `Exception` **__clone** () inherited from `Exception`

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [`Exception` \$previous]) inherited from `Exception`

Exception constructor

public **__wakeup** () inherited from `Exception`

...

final public *string* **getMessage** () inherited from `Exception`

Gets the Exception message

final public *int* **getCode** () inherited from `Exception`

Gets the Exception code

final public *string* **getFile** () inherited from `Exception`

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from `Exception`

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from `Exception`

Gets the stack trace

final public `Exception` **getPrevious** () inherited from `Exception`

Returns previous Exception

final public `Exception` **getTraceAsString** () inherited from `Exception`

Gets the stack trace as a string

public *string* **__toString** () inherited from `Exception`

String representation of the exception

Abstract class Phalcon\Db

Phalcon\Db and its related classes provide a simple SQL database interface for Phalcon Framework. The Phalcon\Db is the basic class you use to connect your PHP application to an RDBMS. There is a different adapter class for each brand of RDBMS.

This component is intended to lower level database operations. If you want to interact with databases using higher level of abstraction use Phalcon\Mvc\Model.

Phalcon\Db is an abstract class. You only can use it with a database adapter like Phalcon\Db\Adapter\Pdo

```
<?php
use Phalcon\Db;
use Phalcon\Db\Exception;
use Phalcon\Db\Adapter\Pdo\Mysql as MySqlConnection;

try {
    $connection = new MySqlConnection(
```

```
[
    "host"      => "192.168.0.11",
    "username"  => "sigma",
    "password"  => "secret",
    "dbname"    => "blog",
    "port"      => "3306",
]

);

$result = $connection->query(
    "SELECT * FROM robots LIMIT 5"
);

$result->setFetchMode(Db::FETCH_NUM);

while ($robot = $result->fetch()) {
    print_r($robot);
}
} catch (Exception $e) {
    echo $e->getMessage(), PHP_EOL;
}
```

Constants

integer **FETCH_LAZY**

integer **FETCH_ASSOC**

integer **FETCH_NAMED**

integer **FETCH_NUM**

integer **FETCH_BOTH**

integer **FETCH_OBJ**

integer **FETCH_BOUND**

integer **FETCH_COLUMN**

integer **FETCH_CLASS**

integer **FETCH_INTO**

integer **FETCH_FUNC**

integer **FETCH_GROUP**

integer **FETCH_UNIQUE**

integer **FETCH_KEY_PAIR**

integer **FETCH_CLASSTYPE**

integer **FETCH_SERIALIZE**

integer **FETCH_PROPS_LATE**

Methods

public static **setup** (*array* \$options)

Enables/disables options in the Database component

Abstract class **Phalcon\Db\Adapter**

*implements **Phalcon\Db\AdapterInterface**, **Phalcon\Events\EventsAwareInterface***

Base class for Phalcon\Db adapters

Methods

public **getDialectType** ()

Name of the dialect used

public **getType** ()

Type of database system the adapter is used for

public **getSqlVariables** ()

Active SQL bound parameter variables

public **__construct** (array \$descriptor)

Phalcon\Db\Adapter constructor

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager)

Sets the event manager

public **getEventManager** ()

Returns the internal event manager

public **setDialect** (*Phalcon\Db\DialectInterface* \$dialect)

Sets the dialect used to produce the SQL

public **getDialect** ()

Returns internal dialect instance

public **fetchOne** (*mixed* \$sqlQuery, [*mixed* \$fetchMode], [*mixed* \$bindParams], [*mixed* \$bindTypes])

Returns the first row in a SQL query result

```
<?php

// Getting first robot
$robot = $connection->fetchOne("SELECT * FROM robots");
print_r($robot);

// Getting first robot with associative indexes only
$robot = $connection->fetchOne("SELECT * FROM robots", \Phalcon\Db::FETCH_ASSOC);
print_r($robot);
```

public array **fetchAll** (string \$sqlQuery, [int \$fetchMode], [array \$bindParams], [array \$bindTypes])

Dumps the complete result of a query into an array

```
<?php

// Getting all robots with associative indexes only
$robots = $connection->fetchAll(
    "SELECT * FROM robots",
    \Phalcon\Db::FETCH_ASSOC
);

foreach ($robots as $robot) {
    print_r($robot);
}

// Getting all robots that contains word "robot" withing the name
$robots = $connection->fetchAll(
    "SELECT * FROM robots WHERE name LIKE :name",
    \Phalcon\Db::FETCH_ASSOC,
    [
        "name" => "%robot%",
    ]
);
foreach($robots as $robot) {
    print_r($robot);
}
```

public *string* | **fetchColumn** (*string* \$sqlQuery, [*array* \$placeholders], [*int* | *string* \$column])

Returns the n'th field of first row in a SQL query result

```
<?php

// Getting count of robots
$robotsCount = $connection->fetchColumn("SELECT count(*) FROM robots");
print_r($robotsCount);

// Getting name of last edited robot
$robot = $connection->fetchColumn(
    "SELECT id, name FROM robots order by modified desc",
    1
);
print_r($robot);
```

public *boolean* **insert** (*string* | *array* \$table, *array* \$values, [*array* \$fields], [*array* \$dataTypes])

Inserts data into a table using custom RDBMS SQL syntax

```
<?php

// Inserting a new robot
$success = $connection->insert(
    "robots",
    ["Astro Boy", 1952],
    ["name", "year"]
);

// Next SQL sentence is sent to the database system
INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public *boolean* **insertAsDict** (*string* \$table, *array* \$data, [*array* \$dataTypes])

Inserts data into a table using custom RBDM SQL syntax

```
<?php

// Inserting a new robot
$success = $connection->insertAsDict(
    "robots",
    [
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);

// Next SQL sentence is sent to the database system
INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public *boolean* **update** (*string* | *array* \$table, *array* \$fields, *array* \$values, [*string* | *array* \$whereCondition], [*array* \$dataTypes])

Updates data on a table using custom RBDM SQL syntax

```
<?php

// Updating existing robot
$success = $connection->update(
    "robots",
    ["name"],
    ["New Astro Boy"],
    "id = 101"
);

// Next SQL sentence is sent to the database system
UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101

// Updating existing robot with array condition and $dataTypes
$success = $connection->update(
    "robots",
    ["name"],
    ["New Astro Boy"],
    [
        "conditions" => "id = ?",
        "bind"        => [$some_unsafe_id],
        "bindTypes"   => [PDO::PARAM_INT], // use only if you use $dataTypes param
    ],
    [
        PDO::PARAM_STR
    ]
);
```

Warning! If \$whereCondition is string it not escaped.

public *boolean* **updateAsDict** (*string* \$table, *array* \$data, [*string* \$whereCondition], [*array* \$dataTypes])

Updates data on a table using custom RBDM SQL syntax Another, more convenient syntax

```
<?php

// Updating existing robot
$success = $connection->updateAsDict(
    "robots",
```

```
[
    "name" => "New Astro Boy",
],
"id = 101"
);

// Next SQL sentence is sent to the database system
UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101
```

public **boolean delete** (*string* | *array* \$table, [*string* \$whereCondition], [*array* \$placeholders], [*array* \$dataTypes])

Deletes data from a table using custom RBDM SQL syntax

```
<?php

// Deleting existing robot
$success = $connection->delete(
    "robots",
    "id = 101"
);

// Next SQL sentence is generated
DELETE FROM `robots` WHERE `id` = 101
```

public **escapeIdentifier** (*array* | *string* \$identifier)

Escapes a column/table/schema name

```
<?php

$escapedTable = $connection->escapeIdentifier(
    "robots"
);

$escapedTable = $connection->escapeIdentifier(
    [
        "store",
        "robots",
    ]
);
```

public **string getColumnList** (*array* \$columnList)

Gets a list of columns

public **limit** (*mixed* \$sqlQuery, *mixed* \$number)

Appends a LIMIT clause to \$sqlQuery argument

```
<?php

echo $connection->limit("SELECT * FROM robots", 5);
```

public **tableExists** (*mixed* \$tableName, [*mixed* \$schemaName])

Generates SQL checking for the existence of a schema.table

```
<?php

var_dump (
```

```
$connection->tableExists("blog", "posts")
);
```

public **viewExists** (*mixed* \$viewName, [*mixed* \$schemaName])

Generates SQL checking for the existence of a schema.view

```
<?php
var_dump (
    $connection->viewExists("active_users", "posts")
);
```

public **forUpdate** (*mixed* \$sqlQuery)

Returns a SQL modified with a FOR UPDATE clause

public **sharedLock** (*mixed* \$sqlQuery)

Returns a SQL modified with a LOCK IN SHARE MODE clause

public **createTable** (*mixed* \$tableName, *mixed* \$schemaName, *array* \$definition)

Creates a table

public **dropTable** (*mixed* \$tableName, [*mixed* \$schemaName], [*mixed* \$ifExists])

Drops a table from a schema/database

public **createView** (*mixed* \$viewName, *array* \$definition, [*mixed* \$schemaName])

Creates a view

public **dropView** (*mixed* \$viewName, [*mixed* \$schemaName], [*mixed* \$ifExists])

Drops a view

public **addColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column)

Adds a column to a table

public **modifyColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column, [*Phalcon\Db\ColumnInterface* \$currentColumn])

Modifies a table column based on a definition

public **dropColumn** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$columnName)

Drops a column from a table

public **addIndex** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index)

Adds an index to a table

public **dropIndex** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$indexName)

Drop an index from a table

public **addPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index)

Adds a primary key to a table

public **dropPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName)

Drops a table's primary key

public **addForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ReferenceInterface* \$reference)

Adds a foreign key to a table

public **dropForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$referenceName)

Drops a foreign key from a table

public **getColumnDefinition** (*Phalcon\Db\ColumnInterface* \$column)

Returns the SQL column definition from a column

public **listTables** ([*mixed* \$schemaName])

List all tables on a database

```
<?php

print_r(
    $connection->listTables("blog")
);
```

public **listViews** ([*mixed* \$schemaName])

List all views on a database

```
<?php

print_r(
    $connection->listViews("blog")
);
```

public *Phalcon\Db\Index*[] **describeIndexes** (*string* \$table, [*string* \$schema])

Lists table indexes

```
<?php

print_r(
    $connection->describeIndexes("robots_parts")
);
```

public **describeReferences** (*mixed* \$table, [*mixed* \$schema])

Lists table references

```
<?php

print_r(
    $connection->describeReferences("robots_parts")
);
```

public **tableOptions** (*mixed* \$tableName, [*mixed* \$schemaName])

Gets creation options from a table

```
<?php

print_r(
    $connection->tableOptions("robots")
);
```

public **createSavepoint** (*mixed* \$name)

Creates a new savepoint

public **releaseSavepoint** (*mixed* \$name)

Releases given savepoint

public **rollbackSavepoint** (*mixed* \$name)

Rollbacks given savepoint

public **setNestedTransactionsWithSavepoints** (*mixed* \$nestedTransactionsWithSavepoints)

Set if nested transactions should use savepoints

public **isNestedTransactionsWithSavepoints** ()

Returns if nested transactions should use savepoints

public **getNestedTransactionSavepointName** ()

Returns the savepoint name to use for nested transactions

public **getDefaultIdValue** ()

Returns the default identity value to be inserted in an identity column

```
<?php
// Inserting a new robot with a valid default value for the column 'id'
$success = $connection->insert(
    "robots",
    [
        $connection->getDefaultIdValue(),
        "Astro Boy",
        1952,
    ],
    [
        "id",
        "name",
        "year",
    ]
);
```

public **getDefaultValue** ()

Returns the default value to make the RDBM use the default value declared in the table definition

```
<?php
// Inserting a new robot with a valid default value for the column 'year'
$success = $connection->insert(
    "robots",
    [
        "Astro Boy",
        $connection->getDefaultValue()
    ],
    [
        "name",
        "year",
    ]
);
```

public **supportSequences** ()

Check whether the database system requires a sequence to produce auto-numeric values

public **useExplicitIdValue** ()

Check whether the database system requires an explicit value for identity columns

public **getDescriptor** ()

Return descriptor used to connect to the active database

public *string* **getConnectionId** ()

Gets the active connection unique identifier

public **getSQLStatement** ()

Active SQL statement in the object

public **getRealSQLStatement** ()

Active SQL statement in the object without replace bound parameters

public *array* **getSQLBindTypes** ()

Active SQL statement in the object

abstract public **connect** ([*array* \$descriptor]) inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **query** (*mixed* \$sqlStatement, [*mixed* \$placeholders], [*mixed* \$dataTypes]) inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **execute** (*mixed* \$sqlStatement, [*mixed* \$placeholders], [*mixed* \$dataTypes]) inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **affectedRows** () inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **close** () inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **escapeString** (*mixed* \$str) inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **lastInsertId** ([*mixed* \$sequenceName]) inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **begin** ([*mixed* \$nesting]) inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **rollback** ([*mixed* \$nesting]) inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **commit** ([*mixed* \$nesting]) inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **isUnderTransaction** () inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **getInternalHandler** () inherited from *Phalcon\Db\AdapterInterface*

...

abstract public **describeColumns** (*mixed* \$table, [*mixed* \$schema]) inherited from *Phalcon\Db\AdapterInterface*

...

Abstract class Phalcon\Db\Adapter\Pdo

extends abstract class *Phalcon\Db\Adapter*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\Db\AdapterInterface*

Phalcon\Db\Adapter\Pdo is the Phalcon\Db that internally uses PDO to connect to a database

```
<?php

use Phalcon\Db\Adapter\Pdo\Mysql;

$config = [
    "host"      => "localhost",
    "dbname"    => "blog",
    "port"      => 3306,
    "username"  => "sigma",
    "password"  => "secret",
];

$connection = new Mysql($config);
```

Methods

public **__construct** (*array* \$descriptor)

Constructor for Phalcon\Db\Adapter\Pdo

public **connect** ([*array* \$descriptor])

This method is automatically called in \Phalcon\Db\Adapter\Pdo constructor. Call it when you need to restore a database connection.

```
<?php

use Phalcon\Db\Adapter\Pdo\Mysql;

// Make a connection
$connection = new Mysql(
    [
        "host"      => "localhost",
        "username"  => "sigma",
        "password"  => "secret",
        "dbname"    => "blog",
        "port"      => 3306,
    ]
);

// Reconnect
$connection->connect();
```

public **prepare** (*mixed* \$sqlStatement)

Returns a PDO prepared statement to be executed with 'executePrepared'

```
<?php
use Phalcon\Db\Column;

$stmt = $db->prepare(
    "SELECT * FROM robots WHERE name = :name"
);

$result = $connection->executePrepared(
    $stmt,
    [
        "name" => "Voltron",
    ],
    [
        "name" => Column::BIND_PARAM_INT,
    ]
);
```

public **PDOStatement executePrepared** (PDOStatement \$statement, array \$placeholders, array \$dataTypes)

Executes a prepared statement binding. This function uses integer indexes starting from zero

```
<?php
use Phalcon\Db\Column;

$stmt = $db->prepare(
    "SELECT * FROM robots WHERE name = :name"
);

$result = $connection->executePrepared(
    $stmt,
    [
        "name" => "Voltron",
    ],
    [
        "name" => Column::BIND_PARAM_INT,
    ]
);
```

public **query** (*mixed* \$sqlStatement, [*mixed* \$bindParam], [*mixed* \$bindTypes])

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server is returning rows

```
<?php

// Querying data
$resultset = $connection->query(
    "SELECT * FROM robots WHERE type = 'mechanical'"
);

$resultset = $connection->query(
    "SELECT * FROM robots WHERE type = ?",
    [
        "mechanical",
    ]
);
```



```
    ]
);
```

public **execute** (*mixed* \$sqlStatement, [*mixed* \$bindParam], [*mixed* \$bindTypes])

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server doesn't return any rows

```
<?php

// Inserting data
$success = $connection->execute(
    "INSERT INTO robots VALUES (1, 'Astro Boy')"
);

$success = $connection->execute(
    "INSERT INTO robots VALUES (?, ?)",
    [
        1,
        "Astro Boy",
    ]
);
```

public **affectedRows** ()

Returns the number of affected rows by the latest INSERT/UPDATE/DELETE executed in the database system

```
<?php

$connection->execute(
    "DELETE FROM robots"
);

echo $connection->affectedRows(), " were deleted";
```

public **close** ()

Closes the active connection returning success. Phalcon automatically closes and destroys active connections when the request ends

public **escapeString** (*mixed* \$str)

Escapes a value to avoid SQL injections according to the active charset in the connection

```
<?php

$escapedStr = $connection->escapeString("some dangerous value");
```

public **convertBoundParams** (*mixed* \$sql, [*array* \$params])

Converts bound parameters such as :name: or ?1 into PDO bind params ?

```
<?php

print_r(
    $connection->convertBoundParams(
        "SELECT * FROM robots WHERE name = :name:",
        [
            "Bender",
        ]
    )
);
```

```
    )  
);
```

public **int | boolean** **lastInsertId** ([*string* \$sequenceName])

Returns the insert id for the auto_increment/serial column inserted in the latest executed SQL statement

```
<?php  
  
// Inserting a new robot  
$success = $connection->insert(  
    "robots",  
    [  
        "Astro Boy",  
        1952,  
    ],  
    [  
        "name",  
        "year",  
    ]  
);  
  
// Getting the generated id  
$id = $connection->lastInsertId();
```

public **begin** ([*mixed* \$nesting])

Starts a transaction in the connection

public **rollback** ([*mixed* \$nesting])

Rollbacks the active transaction in the connection

public **commit** ([*mixed* \$nesting])

Commits the active transaction in the connection

public **getTransactionLevel** ()

Returns the current transaction nesting level

public **isUnderTransaction** ()

Checks whether the connection is under a transaction

```
<?php  
  
$connection->begin();  
  
// true  
var_dump(  
    $connection->isUnderTransaction()  
);
```

public **getInternalHandler** ()

Return internal PDO handler

public **array** **getErrorInfo** ()

Return the error info, if any

public **getDialectType** () inherited from *Phalcon\Db\Adapter*

Name of the dialect used

public **getType** () inherited from *Phalcon\Db\Adapter*

Type of database system the adapter is used for

public **getSqlVariables** () inherited from *Phalcon\Db\Adapter*

Active SQL bound parameter variables

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Db\Adapter*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Db\Adapter*

Returns the internal event manager

public **setDialect** (*Phalcon\Db\DialectInterface* \$dialect) inherited from *Phalcon\Db\Adapter*

Sets the dialect used to produce the SQL

public **getDialect** () inherited from *Phalcon\Db\Adapter*

Returns internal dialect instance

public **fetchOne** (*mixed* \$sqlQuery, [*mixed* \$fetchMode], [*mixed* \$bindParam], [*mixed* \$bindTypes]) inherited from *Phalcon\Db\Adapter*

Returns the first row in a SQL query result

```
<?php

// Getting first robot
$robot = $connection->fetchOne("SELECT * FROM robots");
print_r($robot);

// Getting first robot with associative indexes only
$robot = $connection->fetchOne("SELECT * FROM robots", \Phalcon\Db::FETCH_ASSOC);
print_r($robot);
```

public array **fetchAll** (*string* \$sqlQuery, [*int* \$fetchMode], [*array* \$bindParam], [*array* \$bindTypes]) inherited from *Phalcon\Db\Adapter*

Dumps the complete result of a query into an array

```
<?php

// Getting all robots with associative indexes only
$robots = $connection->fetchAll(
    "SELECT * FROM robots",
    \Phalcon\Db::FETCH_ASSOC
);

foreach ($robots as $robot) {
    print_r($robot);
}

// Getting all robots that contains word "robot" within the name
$robots = $connection->fetchAll(
    "SELECT * FROM robots WHERE name LIKE :name",
    \Phalcon\Db::FETCH_ASSOC,
    [
        "name" => "%robot%",
    ]
);
```

```
    ]
);
foreach($robots as $robot) {
    print_r($robot);
}
```

public *string* | **fetchColumn** (*string* \$sqlQuery, [*array* \$placeholders], [*int* | *string* \$column]) inherited from *Phalcon\Db\Adapter*

Returns the nth field of first row in a SQL query result

```
<?php

// Getting count of robots
$robotsCount = $connection->fetchColumn("SELECT count(*) FROM robots");
print_r($robotsCount);

// Getting name of last edited robot
$robot = $connection->fetchColumn(
    "SELECT id, name FROM robots order by modified desc",
    1
);
print_r($robot);
```

public *boolean* **insert** (*string* | *array* \$table, *array* \$values, [*array* \$fields], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Inserts data into a table using custom RDBMS SQL syntax

```
<?php

// Inserting a new robot
$success = $connection->insert(
    "robots",
    ["Astro Boy", 1952],
    ["name", "year"]
);

// Next SQL sentence is sent to the database system
INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public *boolean* **insertAsDict** (*string* \$table, *array* \$data, [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Inserts data into a table using custom RBDM SQL syntax

```
<?php

// Inserting a new robot
$success = $connection->insertAsDict(
    "robots",
    [
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);

// Next SQL sentence is sent to the database system
INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public *boolean* **update** (*string* | *array* \$table, *array* \$fields, *array* \$values, [*string* | *array* \$whereCondition], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Updates data on a table using custom RBDM SQL syntax

```
<?php

// Updating existing robot
$success = $connection->update(
    "robots",
    ["name"],
    ["New Astro Boy"],
    "id = 101"
);

// Next SQL sentence is sent to the database system
UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101

// Updating existing robot with array condition and $dataTypes
$success = $connection->update(
    "robots",
    ["name"],
    ["New Astro Boy"],
    [
        "conditions" => "id = ?",
        "bind"        => [$some_unsafe_id],
        "bindTypes"   => [PDO::PARAM_INT], // use only if you use $dataTypes param
    ],
    [
        PDO::PARAM_STR
    ]
);
```

Warning! If \$whereCondition is string it not escaped.

public *boolean* **updateAsDict** (*string* \$table, *array* \$data, [*string* \$whereCondition], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Updates data on a table using custom RBDM SQL syntax Another, more convenient syntax

```
<?php

// Updating existing robot
$success = $connection->updateAsDict(
    "robots",
    [
        "name" => "New Astro Boy",
    ],
    "id = 101"
);

// Next SQL sentence is sent to the database system
UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101
```

public *boolean* **delete** (*string* | *array* \$table, [*string* \$whereCondition], [*array* \$placeholders], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Deletes data from a table using custom RBDM SQL syntax

```
<?php

// Deleting existing robot
$success = $connection->delete(
    "robots",
    "id = 101"
);

// Next SQL sentence is generated
DELETE FROM `robots` WHERE `id` = 101
```

public **escapeIdentifier** (*array* | *string* \$identifier) inherited from *Phalcon\Db\Adapter*

Escapes a column/table/schema name

```
<?php

$escapedTable = $connection->escapeIdentifier(
    "robots"
);

$escapedTable = $connection->escapeIdentifier(
    [
        "store",
        "robots",
    ]
);
```

public **getColumnList** (*array* \$columnList) inherited from *Phalcon\Db\Adapter*

Gets a list of columns

public **limit** (*mixed* \$sqlQuery, *mixed* \$number) inherited from *Phalcon\Db\Adapter*

Appends a LIMIT clause to \$sqlQuery argument

```
<?php

echo $connection->limit("SELECT * FROM robots", 5);
```

public **tableExists** (*mixed* \$tableName, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Generates SQL checking for the existence of a schema.table

```
<?php

var_dump(
    $connection->tableExists("blog", "posts")
);
```

public **viewExists** (*mixed* \$viewName, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Generates SQL checking for the existence of a schema.view

```
<?php

var_dump(
    $connection->viewExists("active_users", "posts")
);
```

public **forUpdate** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Adapter*

Returns a SQL modified with a FOR UPDATE clause

public **sharedLock** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Adapter*

Returns a SQL modified with a LOCK IN SHARE MODE clause

public **createTable** (*mixed* \$tableName, *mixed* \$schemaName, *array* \$definition) inherited from *Phalcon\Db\Adapter*

Creates a table

public **dropTable** (*mixed* \$tableName, [*mixed* \$schemaName], [*mixed* \$ifExists]) inherited from *Phalcon\Db\Adapter*

Drops a table from a schema/database

public **createView** (*mixed* \$viewName, *array* \$definition, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Creates a view

public **dropView** (*mixed* \$viewName, [*mixed* \$schemaName], [*mixed* \$ifExists]) inherited from *Phalcon\Db\Adapter*

Drops a view

public **addColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column) inherited from *Phalcon\Db\Adapter*

Adds a column to a table

public **modifyColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column, [*Phalcon\Db\ColumnInterface* \$currentColumn]) inherited from *Phalcon\Db\Adapter*

Modifies a table column based on a definition

public **dropColumn** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$columnName) inherited from *Phalcon\Db\Adapter*

Drops a column from a table

public **addIndex** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index) inherited from *Phalcon\Db\Adapter*

Adds an index to a table

public **dropIndex** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$indexName) inherited from *Phalcon\Db\Adapter*

Drop an index from a table

public **addPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index) inherited from *Phalcon\Db\Adapter*

Adds a primary key to a table

public **dropPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName) inherited from *Phalcon\Db\Adapter*

Drops a table's primary key

public **addForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ReferenceInterface* \$reference) inherited from *Phalcon\Db\Adapter*

Adds a foreign key to a table

public **dropForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$referenceName) inherited from *Phalcon\Db\Adapter*

Drops a foreign key from a table

public **getColumnDefinition** (*Phalcon\Db\ColumnInterface* \$column) inherited from *Phalcon\Db\Adapter*

Returns the SQL column definition from a column

public **listTables** ([mixed \$schemaName]) inherited from *Phalcon\Db\Adapter*

List all tables on a database

```
<?php

print_r (
    $connection->listTables ("blog")
);
```

public **listViews** ([mixed \$schemaName]) inherited from *Phalcon\Db\Adapter*

List all views on a database

```
<?php

print_r (
    $connection->listViews ("blog")
);
```

public *Phalcon\Db\Index*[] **describeIndexes** (string \$table, [string \$schema]) inherited from *Phalcon\Db\Adapter*

Lists table indexes

```
<?php

print_r (
    $connection->describeIndexes ("robots_parts")
);
```

public **describeReferences** (mixed \$table, [mixed \$schema]) inherited from *Phalcon\Db\Adapter*

Lists table references

```
<?php

print_r (
    $connection->describeReferences ("robots_parts")
);
```

public **tableOptions** (mixed \$tableName, [mixed \$schemaName]) inherited from *Phalcon\Db\Adapter*

Gets creation options from a table

```
<?php

print_r (
    $connection->tableOptions ("robots")
);
```

public **createSavepoint** (mixed \$name) inherited from *Phalcon\Db\Adapter*

Creates a new savepoint

public **releaseSavepoint** (mixed \$name) inherited from *Phalcon\Db\Adapter*

Releases given savepoint

public **rollbackSavepoint** (mixed \$name) inherited from *Phalcon\Db\Adapter*

Rollbacks given savepoint

public **setNestedTransactionsWithSavepoints** (*mixed* \$nestedTransactionsWithSavepoints) inherited from *Phalcon\Db\Adapter*

Set if nested transactions should use savepoints

public **isNestedTransactionsWithSavepoints** () inherited from *Phalcon\Db\Adapter*

Returns if nested transactions should use savepoints

public **getNestedTransactionSavepointName** () inherited from *Phalcon\Db\Adapter*

Returns the savepoint name to use for nested transactions

public **getDefaultIdValue** () inherited from *Phalcon\Db\Adapter*

Returns the default identity value to be inserted in an identity column

```
<?php

// Inserting a new robot with a valid default value for the column 'id'
$success = $connection->insert(
    "robots",
    [
        $connection->getDefaultIdValue(),
        "Astro Boy",
        1952,
    ],
    [
        "id",
        "name",
        "year",
    ]
);
```

public **getDefaultValue** () inherited from *Phalcon\Db\Adapter*

Returns the default value to make the RDBM use the default value declared in the table definition

```
<?php

// Inserting a new robot with a valid default value for the column 'year'
$success = $connection->insert(
    "robots",
    [
        "Astro Boy",
        $connection->getDefaultValue()
    ],
    [
        "name",
        "year",
    ]
);
```

public **supportSequences** () inherited from *Phalcon\Db\Adapter*

Check whether the database system requires a sequence to produce auto-numeric values

public **useExplicitIdValue** () inherited from *Phalcon\Db\Adapter*

Check whether the database system requires an explicit value for identity columns

public **getDescriptor** () inherited from *Phalcon\Db\Adapter*

Return descriptor used to connect to the active database

public *string* **getConnectionId** () inherited from *Phalcon\Db\Adapter*

Gets the active connection unique identifier

public **getSQLStatement** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object

public **getRealSQLStatement** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object without replace bound parameters

public *array* **getSQLBindTypes** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object

abstract public **describeColumns** (*mixed* \$table, [*mixed* \$schema]) inherited from *Phalcon\Db\AdapterInterface*

...

Class *Phalcon\Db\Adapter\Pdo\Mysql*

extends abstract class *Phalcon\Db\Adapter\Pdo*

implements *Phalcon\Db\AdapterInterface*, *Phalcon\Events\EventsAwareInterface*

Specific functions for the Mysql database system

```
<?php

use Phalcon\Db\Adapter\Pdo\Mysql;

$config = [
    "host"      => "localhost",
    "dbname"    => "blog",
    "port"      => 3306,
    "username"  => "sigma",
    "password"  => "secret",
];

$connection = new Mysql($config);
```

Methods

public **describeColumns** (*mixed* \$table, [*mixed* \$schema])

Returns an array of *Phalcon\Db\Column* objects describing a table

```
<?php

print_r(
    $connection->describeColumns("posts")
);
```

public *Phalcon\Db\IndexInterface*[] **describeIndexes** (*string* \$table, [*string* \$schema])

Lists table indexes

```
<?php

print_r(
    $connection->describeIndexes("robots_parts")
);
```

public **describeReferences** (*mixed* \$table, [*mixed* \$schema])

Lists table references

```
<?php

print_r(
    $connection->describeReferences("robots_parts")
);
```

public **__construct** (*array* \$descriptor) inherited from *Phalcon\Db\Adapter\Pdo*

Constructor for Phalcon\Db\Adapter\Pdo

public **connect** ([*array* \$descriptor]) inherited from *Phalcon\Db\Adapter\Pdo*

This method is automatically called in \Phalcon\Db\Adapter\Pdo constructor. Call it when you need to restore a database connection.

```
<?php

use Phalcon\Db\Adapter\Pdo\Mysql;

// Make a connection
$connection = new Mysql(
    [
        "host"      => "localhost",
        "username"  => "sigma",
        "password"  => "secret",
        "dbname"    => "blog",
        "port"      => 3306,
    ]
);

// Reconnect
$connection->connect();
```

public **prepare** (*mixed* \$sqlStatement) inherited from *Phalcon\Db\Adapter\Pdo*

Returns a PDO prepared statement to be executed with 'executePrepared'

```
<?php

use Phalcon\Db\Column;

$statement = $db->prepare(
    "SELECT * FROM robots WHERE name = :name"
);

$result = $connection->executePrepared(
    $statement,
    [
        "name" => "Voltron",
    ],
);
```

```
[
    "name" => Column::BIND_PARAM_INT,
]
);
```

public **PDOSStatement** **executePrepared** (PDOSStatement \$statement, array \$placeholders, array \$dataTypes) inherited from *Phalcon\Db\Adapter\Pdo*

Executes a prepared statement binding. This function uses integer indexes starting from zero

```
<?php

use Phalcon\Db\Column;

$statement = $db->prepare(
    "SELECT * FROM robots WHERE name = :name"
);

$result = $connection->executePrepared(
    $statement,
    [
        "name" => "Voltron",
    ],
    [
        "name" => Column::BIND_PARAM_INT,
    ]
);
```

public **query** (mixed \$sqlStatement, [mixed \$bindParam], [mixed \$bindTypes]) inherited from *Phalcon\Db\Adapter\Pdo*

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server is returning rows

```
<?php

// Querying data
$resultset = $connection->query(
    "SELECT * FROM robots WHERE type = 'mechanical'"
);

$resultset = $connection->query(
    "SELECT * FROM robots WHERE type = ?",
    [
        "mechanical",
    ]
);
```

public **execute** (mixed \$sqlStatement, [mixed \$bindParam], [mixed \$bindTypes]) inherited from *Phalcon\Db\Adapter\Pdo*

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server doesn't return any rows

```
<?php

// Inserting data
$success = $connection->execute(
    "INSERT INTO robots VALUES (1, 'Astro Boy')"
```

```
);

$success = $connection->execute(
    "INSERT INTO robots VALUES (?, ?)",
    [
        1,
        "Astro Boy",
    ]
);
```

public **affectedRows** () inherited from *Phalcon\Db\Adapter\Pdo*

Returns the number of affected rows by the latest INSERT/UPDATE/DELETE executed in the database system

```
<?php

$connection->execute(
    "DELETE FROM robots"
);

echo $connection->affectedRows(), " were deleted";
```

public **close** () inherited from *Phalcon\Db\Adapter\Pdo*

Closes the active connection returning success. Phalcon automatically closes and destroys active connections when the request ends

public **escapeString** (mixed \$str) inherited from *Phalcon\Db\Adapter\Pdo*

Escapes a value to avoid SQL injections according to the active charset in the connection

```
<?php

$escapedStr = $connection->escapeString("some dangerous value");
```

public **convertBoundParams** (mixed \$sql, [array \$params]) inherited from *Phalcon\Db\Adapter\Pdo*

Converts bound parameters such as :name: or ?1 into PDO bind params ?

```
<?php

print_r(
    $connection->convertBoundParams(
        "SELECT * FROM robots WHERE name = :name:",
        [
            "Bender",
        ]
    )
);
```

public int | boolean **lastInsertId** ([string \$sequenceName]) inherited from *Phalcon\Db\Adapter\Pdo*

Returns the insert id for the auto_increment/serial column inserted in the latest executed SQL statement

```
<?php

// Inserting a new robot
$success = $connection->insert(
    "robots",
    [
```

```
        "Astro Boy",
        1952,
    ],
    [
        "name",
        "year",
    ]
);

// Getting the generated id
$id = $connection->lastInsertId();
```

public **begin** ([mixed \$nesting]) inherited from *Phalcon\Db\Adapter\Pdo*

Starts a transaction in the connection

public **rollback** ([mixed \$nesting]) inherited from *Phalcon\Db\Adapter\Pdo*

Rollbacks the active transaction in the connection

public **commit** ([mixed \$nesting]) inherited from *Phalcon\Db\Adapter\Pdo*

Commits the active transaction in the connection

public **getTransactionLevel** () inherited from *Phalcon\Db\Adapter\Pdo*

Returns the current transaction nesting level

public **isUnderTransaction** () inherited from *Phalcon\Db\Adapter\Pdo*

Checks whether the connection is under a transaction

```
<?php

$connection->begin();

// true
var_dump(
    $connection->isUnderTransaction()
);
```

public **getInternalHandler** () inherited from *Phalcon\Db\Adapter\Pdo*

Return internal PDO handler

public **getErrorInfo** () inherited from *Phalcon\Db\Adapter\Pdo*

Return the error info, if any

public **getDialectType** () inherited from *Phalcon\Db\Adapter*

Name of the dialect used

public **getType** () inherited from *Phalcon\Db\Adapter*

Type of database system the adapter is used for

public **getSqlVariables** () inherited from *Phalcon\Db\Adapter*

Active SQL bound parameter variables

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Db\Adapter*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Db\Adapter*

Returns the internal event manager

public **setDialect** (*Phalcon\Db\DialectInterface* \$dialect) inherited from *Phalcon\Db\Adapter*

Sets the dialect used to produce the SQL

public **getDialect** () inherited from *Phalcon\Db\Adapter*

Returns internal dialect instance

public **fetchOne** (*mixed* \$sqlQuery, [*mixed* \$fetchMode], [*mixed* \$bindParam], [*mixed* \$bindTypes]) inherited from *Phalcon\Db\Adapter*

Returns the first row in a SQL query result

```
<?php

// Getting first robot
$robot = $connection->fetchOne("SELECT * FROM robots");
print_r($robot);

// Getting first robot with associative indexes only
$robot = $connection->fetchOne("SELECT * FROM robots", \Phalcon\Db::FETCH_ASSOC);
print_r($robot);
```

public **fetchAll** (*string* \$sqlQuery, [*int* \$fetchMode], [*array* \$bindParam], [*array* \$bindTypes]) inherited from *Phalcon\Db\Adapter*

Dumps the complete result of a query into an array

```
<?php

// Getting all robots with associative indexes only
$robots = $connection->fetchAll(
    "SELECT * FROM robots",
    \Phalcon\Db::FETCH_ASSOC
);

foreach ($robots as $robot) {
    print_r($robot);
}

// Getting all robots that contains word "robot" within the name
$robots = $connection->fetchAll(
    "SELECT * FROM robots WHERE name LIKE :name",
    \Phalcon\Db::FETCH_ASSOC,
    [
        "name" => "%robot%",
    ]
);
foreach($robots as $robot) {
    print_r($robot);
}
```

public *string* **fetchColumn** (*string* \$sqlQuery, [*array* \$placeholders], [*int* | *string* \$column]) inherited from *Phalcon\Db\Adapter*

Returns the n'th field of first row in a SQL query result

```
<?php
```

```
// Getting count of robots
$robotsCount = $connection->fetchColumn("SELECT count(*) FROM robots");
print_r($robotsCount);

// Getting name of last edited robot
$robot = $connection->fetchColumn(
    "SELECT id, name FROM robots order by modified desc",
    1
);
print_r($robot);
```

public boolean **insert** (string | array \$table, array \$values, [array \$fields], [array \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Inserts data into a table using custom RDBMS SQL syntax

```
<?php

// Inserting a new robot
$success = $connection->insert(
    "robots",
    ["Astro Boy", 1952],
    ["name", "year"]
);

// Next SQL sentence is sent to the database system
INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public boolean **insertAsDict** (string \$table, array \$data, [array \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Inserts data into a table using custom RBDM SQL syntax

```
<?php

// Inserting a new robot
$success = $connection->insertAsDict(
    "robots",
    [
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);

// Next SQL sentence is sent to the database system
INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public boolean **update** (string | array \$table, array \$fields, array \$values, [string | array \$whereCondition], [array \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Updates data on a table using custom RBDM SQL syntax

```
<?php

// Updating existing robot
$success = $connection->update(
    "robots",
    ["name"],
    ["New Astro Boy"],
    "id = 101"
```



```
);

// Next SQL sentence is sent to the database system
UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101

// Updating existing robot with array condition and $dataTypes
$success = $connection->update(
    "robots",
    ["name"],
    ["New Astro Boy"],
    [
        "conditions" => "id = ?",
        "bind"        => [$some_unsafe_id],
        "bindTypes"   => [PDO::PARAM_INT], // use only if you use $dataTypes param
    ],
    [
        PDO::PARAM_STR
    ]
);
```

Warning! If \$whereCondition is string it not escaped.

public **boolean** **updateAsDict** (*string* \$table, *array* \$data, [*string* \$whereCondition], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Updates data on a table using custom RBDM SQL syntax Another, more convenient syntax

```
<?php

// Updating existing robot
$success = $connection->updateAsDict(
    "robots",
    [
        "name" => "New Astro Boy",
    ],
    "id = 101"
);

// Next SQL sentence is sent to the database system
UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101
```

public **boolean** **delete** (*string* | *array* \$table, [*string* \$whereCondition], [*array* \$placeholders], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Deletes data from a table using custom RBDM SQL syntax

```
<?php

// Deleting existing robot
$success = $connection->delete(
    "robots",
    "id = 101"
);

// Next SQL sentence is generated
DELETE FROM `robots` WHERE `id` = 101
```

public **escapeIdentifier** (*array* | *string* \$identifier) inherited from *Phalcon\Db\Adapter*

Escapes a column/table/schema name

```
<?php

$escapedTable = $connection->escapeIdentifier(
    "robots"
);

$escapedTable = $connection->escapeIdentifier(
    [
        "store",
        "robots",
    ]
);
```

public **string** **getColumnList** (array \$columnList) inherited from *Phalcon\Db\Adapter*

Gets a list of columns

public **limit** (mixed \$sqlQuery, mixed \$number) inherited from *Phalcon\Db\Adapter*

Appends a LIMIT clause to \$sqlQuery argument

```
<?php

echo $connection->limit("SELECT * FROM robots", 5);
```

public **boolean** **tableExists** (mixed \$tableName, [mixed \$schemaName]) inherited from *Phalcon\Db\Adapter*

Generates SQL checking for the existence of a schema.table

```
<?php

var_dump(
    $connection->tableExists("blog", "posts")
);
```

public **boolean** **viewExists** (mixed \$viewName, [mixed \$schemaName]) inherited from *Phalcon\Db\Adapter*

Generates SQL checking for the existence of a schema.view

```
<?php

var_dump(
    $connection->viewExists("active_users", "posts")
);
```

public **string** **forUpdate** (mixed \$sqlQuery) inherited from *Phalcon\Db\Adapter*

Returns a SQL modified with a FOR UPDATE clause

public **string** **sharedLock** (mixed \$sqlQuery) inherited from *Phalcon\Db\Adapter*

Returns a SQL modified with a LOCK IN SHARE MODE clause

public **boolean** **createTable** (mixed \$tableName, mixed \$schemaName, array \$definition) inherited from *Phalcon\Db\Adapter*

Creates a table

public **boolean** **dropTable** (mixed \$tableName, [mixed \$schemaName], [mixed \$ifExists]) inherited from *Phalcon\Db\Adapter*

Drops a table from a schema/database

public **createView** (*mixed* \$viewName, *array* \$definition, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Creates a view

public **dropView** (*mixed* \$viewName, [*mixed* \$schemaName], [*mixed* \$ifExists]) inherited from *Phalcon\Db\Adapter*

Drops a view

public **addColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column) inherited from *Phalcon\Db\Adapter*

Adds a column to a table

public **modifyColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column, [*Phalcon\Db\ColumnInterface* \$currentColumn]) inherited from *Phalcon\Db\Adapter*

Modifies a table column based on a definition

public **dropColumn** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$columnName) inherited from *Phalcon\Db\Adapter*

Drops a column from a table

public **addIndex** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index) inherited from *Phalcon\Db\Adapter*

Adds an index to a table

public **dropIndex** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$indexName) inherited from *Phalcon\Db\Adapter*

Drop an index from a table

public **addPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index) inherited from *Phalcon\Db\Adapter*

Adds a primary key to a table

public **dropPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName) inherited from *Phalcon\Db\Adapter*

Drops a table's primary key

public **addForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ReferenceInterface* \$reference) inherited from *Phalcon\Db\Adapter*

Adds a foreign key to a table

public **dropForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$referenceName) inherited from *Phalcon\Db\Adapter*

Drops a foreign key from a table

public **getColumnDefinition** (*Phalcon\Db\ColumnInterface* \$column) inherited from *Phalcon\Db\Adapter*

Returns the SQL column definition from a column

public **listTables** ([*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

List all tables on a database

```
<?php
print_r(
    $connection->listTables("blog")
);
```

public **listViews** ([mixed \$schemaName]) inherited from *Phalcon\Db\Adapter*

List all views on a database

```
<?php

print_r(
    $connection->listViews("blog")
);
```

public **tableOptions** (mixed \$tableName, [mixed \$schemaName]) inherited from *Phalcon\Db\Adapter*

Gets creation options from a table

```
<?php

print_r(
    $connection->tableOptions("robots")
);
```

public **createSavepoint** (mixed \$name) inherited from *Phalcon\Db\Adapter*

Creates a new savepoint

public **releaseSavepoint** (mixed \$name) inherited from *Phalcon\Db\Adapter*

Releases given savepoint

public **rollbackSavepoint** (mixed \$name) inherited from *Phalcon\Db\Adapter*

Rollbacks given savepoint

public **setNestedTransactionsWithSavepoints** (mixed \$nestedTransactionsWithSavepoints) inherited from *Phalcon\Db\Adapter*

Set if nested transactions should use savepoints

public **isNestedTransactionsWithSavepoints** () inherited from *Phalcon\Db\Adapter*

Returns if nested transactions should use savepoints

public **getNestedTransactionSavepointName** () inherited from *Phalcon\Db\Adapter*

Returns the savepoint name to use for nested transactions

public **getDefaultIdValue** () inherited from *Phalcon\Db\Adapter*

Returns the default identity value to be inserted in an identity column

```
<?php

// Inserting a new robot with a valid default value for the column 'id'
$success = $connection->insert(
    "robots",
    [
        $connection->getDefaultIdValue(),
        "Astro Boy",
        1952,
    ],
    [
        "id",
        "name",
        "year",
    ],
);
```

```
    ]
);
```

public **getDefaultValue** () inherited from *Phalcon\Db\Adapter*

Returns the default value to make the RDBM use the default value declared in the table definition

```
<?php

// Inserting a new robot with a valid default value for the column 'year'
$success = $connection->insert(
    "robots",
    [
        "Astro Boy",
        $connection->getDefaultValue()
    ],
    [
        "name",
        "year",
    ]
);
```

public **supportSequences** () inherited from *Phalcon\Db\Adapter*

Check whether the database system requires a sequence to produce auto-numeric values

public **useExplicitIdValue** () inherited from *Phalcon\Db\Adapter*

Check whether the database system requires an explicit value for identity columns

public **getDescriptor** () inherited from *Phalcon\Db\Adapter*

Return descriptor used to connect to the active database

public *string* **getConnectionId** () inherited from *Phalcon\Db\Adapter*

Gets the active connection unique identifier

public **getSQLStatement** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object

public **getRealSQLStatement** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object without replace bound parameters

public *array* **getSQLBindTypes** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object

Class *Phalcon\Db\Adapter\Pdo\Postgresql*

extends abstract class *Phalcon\Db\Adapter\Pdo*

implements *Phalcon\Db\AdapterInterface*, *Phalcon\Events\EventsAwareInterface*

Specific functions for the Postgresql database system

```
<?php

use Phalcon\Db\Adapter\Pdo\Postgresql;

$config = [
```

```
"host"      => "localhost",
"dbname"    => "blog",
"port"      => 5432,
"username"  => "postgres",
"password"  => "secret",
];

$connection = new Postgresql($config);
```

Methods

public **connect** ([array \$descriptor])

This method is automatically called in `Phalcon\Db\Adapter\Pdo` constructor. Call it when you need to restore a database connection.

public **describeColumns** (*mixed* \$table, [*mixed* \$schema])

Returns an array of `Phalcon\Db\Column` objects describing a table

```
<?php

print_r(
    $connection->describeColumns("posts")
);
```

public **createTable** (*mixed* \$tableName, *mixed* \$schemaName, array \$definition)

Creates a table

public **modifyColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column, [*Phalcon\Db\ColumnInterface* \$currentColumn])

Modifies a table column based on a definition

public **useExplicitIdValue** ()

Check whether the database system requires an explicit value for identity columns

public **getDefaultIdValue** ()

Returns the default identity value to be inserted in an identity column

```
<?php

// Inserting a new robot with a valid default value for the column 'id'
$success = $connection->insert(
    "robots",
    [
        $connection->getDefaultIdValue(),
        "Astro Boy",
        1952,
    ],
    [
        "id",
        "name",
        "year",
    ]
);
```

public **supportSequences** ()

Check whether the database system requires a sequence to produce auto-numeric values

public **__construct** (array \$descriptor) inherited from *Phalcon\Db\Adapter\Pdo*

Constructor for Phalcon\Db\Adapter\Pdo

public **prepare** (mixed \$sqlStatement) inherited from *Phalcon\Db\Adapter\Pdo*

Returns a PDO prepared statement to be executed with 'executePrepared'

```
<?php
use Phalcon\Db\Column;

$stmt = $db->prepare(
    "SELECT * FROM robots WHERE name = :name"
);

$result = $connection->executePrepared(
    $stmt,
    [
        "name" => "Voltron",
    ],
    [
        "name" => Column::BIND_PARAM_INT,
    ]
);
```

public **PDOStatement executePrepared** (PDOStatement \$statement, array \$placeholders, array \$dataTypes) inherited from *Phalcon\Db\Adapter\Pdo*

Executes a prepared statement binding. This function uses integer indexes starting from zero

```
<?php
use Phalcon\Db\Column;

$stmt = $db->prepare(
    "SELECT * FROM robots WHERE name = :name"
);

$result = $connection->executePrepared(
    $stmt,
    [
        "name" => "Voltron",
    ],
    [
        "name" => Column::BIND_PARAM_INT,
    ]
);
```

public **query** (mixed \$sqlStatement, [mixed \$bindParams], [mixed \$bindTypes]) inherited from *Phalcon\Db\Adapter\Pdo*

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server is returning rows

```
<?php
```

```
// Querying data
$resultset = $connection->query(
    "SELECT * FROM robots WHERE type = 'mechanical'"
);

$resultset = $connection->query(
    "SELECT * FROM robots WHERE type = ?",
    [
        "mechanical",
    ]
);
```

public **execute** (*mixed* \$sqlStatement, [*mixed* \$bindParam], [*mixed* \$bindTypes]) inherited from *Phalcon\Db\Adapter\Pdo*

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server doesn't return any rows

```
<?php

// Inserting data
$success = $connection->execute(
    "INSERT INTO robots VALUES (1, 'Astro Boy')"
);

$success = $connection->execute(
    "INSERT INTO robots VALUES (?, ?)",
    [
        1,
        "Astro Boy",
    ]
);
```

public **affectedRows** () inherited from *Phalcon\Db\Adapter\Pdo*

Returns the number of affected rows by the latest INSERT/UPDATE/DELETE executed in the database system

```
<?php

$connection->execute(
    "DELETE FROM robots"
);

echo $connection->affectedRows(), " were deleted";
```

public **close** () inherited from *Phalcon\Db\Adapter\Pdo*

Closes the active connection returning success. Phalcon automatically closes and destroys active connections when the request ends

public **escapeString** (*mixed* \$str) inherited from *Phalcon\Db\Adapter\Pdo*

Escapes a value to avoid SQL injections according to the active charset in the connection

```
<?php

$escapedStr = $connection->escapeString("some dangerous value");
```

public **convertBoundParams** (*mixed* \$sql, *array* \$params) inherited from *Phalcon\Db\Adapter\Pdo*

Converts bound parameters such as :name: or ?1 into PDO bind params ?

```
<?php

print_r(
    $connection->convertBoundParams(
        "SELECT * FROM robots WHERE name = :name:",
        [
            "Bender",
        ]
    )
);
```

public **int|boolean lastInsertId** ([string \$sequenceName]) inherited from *Phalcon\Db\Adapter\Pdo*

Returns the insert id for the auto_increment/serial column inserted in the latest executed SQL statement

```
<?php

// Inserting a new robot
$success = $connection->insert(
    "robots",
    [
        "Astro Boy",
        1952,
    ],
    [
        "name",
        "year",
    ]
);

// Getting the generated id
$id = $connection->lastInsertId();
```

public **begin** ([mixed \$nesting]) inherited from *Phalcon\Db\Adapter\Pdo*

Starts a transaction in the connection

public **rollback** ([mixed \$nesting]) inherited from *Phalcon\Db\Adapter\Pdo*

Rollbacks the active transaction in the connection

public **commit** ([mixed \$nesting]) inherited from *Phalcon\Db\Adapter\Pdo*

Commits the active transaction in the connection

public **getTransactionLevel** () inherited from *Phalcon\Db\Adapter\Pdo*

Returns the current transaction nesting level

public **isUnderTransaction** () inherited from *Phalcon\Db\Adapter\Pdo*

Checks whether the connection is under a transaction

```
<?php

$connection->begin();

// true
var_dump(
```

```
$connection->isUnderTransaction()  
);
```

public **getInternalHandler** () inherited from *Phalcon\Db\Adapter\Pdo*

Return internal PDO handler

public array **getErrorInfo** () inherited from *Phalcon\Db\Adapter\Pdo*

Return the error info, if any

public **getDialectType** () inherited from *Phalcon\Db\Adapter*

Name of the dialect used

public **getType** () inherited from *Phalcon\Db\Adapter*

Type of database system the adapter is used for

public **getSqlVariables** () inherited from *Phalcon\Db\Adapter*

Active SQL bound parameter variables

public **setEventsManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Db\Adapter*

Sets the event manager

public **getEventsManager** () inherited from *Phalcon\Db\Adapter*

Returns the internal event manager

public **setDialect** (*Phalcon\Db\DialectInterface* \$dialect) inherited from *Phalcon\Db\Adapter*

Sets the dialect used to produce the SQL

public **getDialect** () inherited from *Phalcon\Db\Adapter*

Returns internal dialect instance

public **fetchOne** (*mixed* \$sqlQuery, [*mixed* \$fetchMode], [*mixed* \$bindParam], [*mixed* \$bindTypes]) inherited from *Phalcon\Db\Adapter*

Returns the first row in a SQL query result

```
<?php  
  
// Getting first robot  
$robot = $connection->fetchOne("SELECT * FROM robots");  
print_r($robot);  
  
// Getting first robot with associative indexes only  
$robot = $connection->fetchOne("SELECT * FROM robots", \Phalcon\Db::FETCH_ASSOC);  
print_r($robot);
```

public array **fetchAll** (*string* \$sqlQuery, [*int* \$fetchMode], [*array* \$bindParam], [*array* \$bindTypes]) inherited from *Phalcon\Db\Adapter*

Dumps the complete result of a query into an array

```
<?php  
  
// Getting all robots with associative indexes only  
$robots = $connection->fetchAll(  
    "SELECT * FROM robots",  
    \Phalcon\Db::FETCH_ASSOC
```

```
);

foreach ($robots as $robot) {
    print_r($robot);
}

// Getting all robots that contains word "robot" within the name
$robots = $connection->fetchAll(
    "SELECT * FROM robots WHERE name LIKE :name",
    \Phalcon\Db::FETCH_ASSOC,
    [
        "name" => "%robot%",
    ]
);
foreach($robots as $robot) {
    print_r($robot);
}
```

public *string* | **fetchColumn** (*string* \$sqlQuery, [*array* \$placeholders], [*int* | *string* \$column]) inherited from *Phalcon\Db\Adapter*

Returns the n'th field of first row in a SQL query result

```
<?php

// Getting count of robots
$robotsCount = $connection->fetchColumn("SELECT count(*) FROM robots");
print_r($robotsCount);

// Getting name of last edited robot
$robot = $connection->fetchColumn(
    "SELECT id, name FROM robots order by modified desc",
    1
);
print_r($robot);
```

public *boolean* **insert** (*string* | *array* \$table, *array* \$values, [*array* \$fields], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Inserts data into a table using custom RDBMS SQL syntax

```
<?php

// Inserting a new robot
$success = $connection->insert(
    "robots",
    ["Astro Boy", 1952],
    ["name", "year"]
);

// Next SQL sentence is sent to the database system
INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public *boolean* **insertAsDict** (*string* \$table, *array* \$data, [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Inserts data into a table using custom RBDM SQL syntax

```
<?php
```

```
// Inserting a new robot
$success = $connection->insertAsDict(
    "robots",
    [
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);

// Next SQL sentence is sent to the database system
INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public boolean **update** (string | array \$table, array \$fields, array \$values, [string | array \$whereCondition], [array \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Updates data on a table using custom RBDM SQL syntax

```
<?php

// Updating existing robot
$success = $connection->update(
    "robots",
    ["name"],
    ["New Astro Boy"],
    "id = 101"
);

// Next SQL sentence is sent to the database system
UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101

// Updating existing robot with array condition and $dataTypes
$success = $connection->update(
    "robots",
    ["name"],
    ["New Astro Boy"],
    [
        "conditions" => "id = ?",
        "bind"        => [$some_unsafe_id],
        "bindTypes"   => [PDO::PARAM_INT], // use only if you use $dataTypes param
    ],
    [
        PDO::PARAM_STR
    ]
);
```

Warning! If \$whereCondition is string it not escaped.

public boolean **updateAsDict** (string \$table, array \$data, [string \$whereCondition], [array \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Updates data on a table using custom RBDM SQL syntax Another, more convenient syntax

```
<?php

// Updating existing robot
$success = $connection->updateAsDict(
    "robots",
    [
        "name" => "New Astro Boy",
```

```

        ],
        "id = 101"
    );

    // Next SQL sentence is sent to the database system
    UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101

```

public **boolean delete** (*string* | *array* \$table, [*string* \$whereCondition], [*array* \$placeholders], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Deletes data from a table using custom RBDM SQL syntax

```

<?php

// Deleting existing robot
$success = $connection->delete(
    "robots",
    "id = 101"
);

// Next SQL sentence is generated
DELETE FROM `robots` WHERE `id` = 101

```

public **escapeIdentifier** (*array* | *string* \$identifier) inherited from *Phalcon\Db\Adapter*

Escapes a column/table/schema name

```

<?php

$escapedTable = $connection->escapeIdentifier(
    "robots"
);

$escapedTable = $connection->escapeIdentifier(
    [
        "store",
        "robots",
    ]
);

```

public **string getColumnList** (*array* \$columnList) inherited from *Phalcon\Db\Adapter*

Gets a list of columns

public **limit** (*mixed* \$sqlQuery, *mixed* \$number) inherited from *Phalcon\Db\Adapter*

Appends a LIMIT clause to \$sqlQuery argument

```

<?php

echo $connection->limit("SELECT * FROM robots", 5);

```

public **tableExists** (*mixed* \$tableName, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Generates SQL checking for the existence of a schema.table

```

<?php

var_dump(

```

```
$connection->tableExists("blog", "posts")
);
```

public **viewExists** (*mixed* \$viewName, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Generates SQL checking for the existence of a schema.view

```
<?php
var_dump (
    $connection->viewExists("active_users", "posts")
);
```

public **forUpdate** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Adapter*

Returns a SQL modified with a FOR UPDATE clause

public **sharedLock** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Adapter*

Returns a SQL modified with a LOCK IN SHARE MODE clause

public **dropTable** (*mixed* \$tableName, [*mixed* \$schemaName], [*mixed* \$ifExists]) inherited from *Phalcon\Db\Adapter*

Drops a table from a schema/database

public **createView** (*mixed* \$viewName, *array* \$definition, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Creates a view

public **dropView** (*mixed* \$viewName, [*mixed* \$schemaName], [*mixed* \$ifExists]) inherited from *Phalcon\Db\Adapter*

Drops a view

public **addColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column) inherited from *Phalcon\Db\Adapter*

Adds a column to a table

public **dropColumn** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$columnName) inherited from *Phalcon\Db\Adapter*

Drops a column from a table

public **addIndex** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index) inherited from *Phalcon\Db\Adapter*

Adds an index to a table

public **dropIndex** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$indexName) inherited from *Phalcon\Db\Adapter*

Drop an index from a table

public **addPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index) inherited from *Phalcon\Db\Adapter*

Adds a primary key to a table

public **dropPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName) inherited from *Phalcon\Db\Adapter*

Drops a table's primary key

public **addForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ReferenceInterface* \$reference) inherited from *Phalcon\Db\Adapter*

Adds a foreign key to a table

public **dropForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$referenceName) inherited from *Phalcon\Db\Adapter*

Drops a foreign key from a table

public **getColumnDefinition** (*Phalcon\Db\ColumnInterface* \$column) inherited from *Phalcon\Db\Adapter*

Returns the SQL column definition from a column

public **listTables** ([*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

List all tables on a database

```
<?php
print_r(
    $connection->listTables("blog")
);
```

public **listViews** ([*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

List all views on a database

```
<?php
print_r(
    $connection->listViews("blog")
);
```

public *Phalcon\Db\Index*[] **describeIndexes** (*string* \$table, [*string* \$schema]) inherited from *Phalcon\Db\Adapter*

Lists table indexes

```
<?php
print_r(
    $connection->describeIndexes("robots_parts")
);
```

public **describeReferences** (*mixed* \$table, [*mixed* \$schema]) inherited from *Phalcon\Db\Adapter*

Lists table references

```
<?php
print_r(
    $connection->describeReferences("robots_parts")
);
```

public **tableOptions** (*mixed* \$tableName, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Gets creation options from a table

```
<?php
print_r(
    $connection->tableOptions("robots")
);
```

public **createSavepoint** (*mixed* \$name) inherited from *Phalcon\Db\Adapter*

Creates a new savepoint

public **releaseSavepoint** (*mixed* \$name) inherited from *Phalcon\Db\Adapter*

Releases given savepoint

public **rollbackSavepoint** (*mixed* \$name) inherited from *Phalcon\Db\Adapter*

Rollbacks given savepoint

public **setNestedTransactionsWithSavepoints** (*mixed* \$nestedTransactionsWithSavepoints) inherited from *Phalcon\Db\Adapter*

Set if nested transactions should use savepoints

public **isNestedTransactionsWithSavepoints** () inherited from *Phalcon\Db\Adapter*

Returns if nested transactions should use savepoints

public **getNestedTransactionSavepointName** () inherited from *Phalcon\Db\Adapter*

Returns the savepoint name to use for nested transactions

public **getDefaultValue** () inherited from *Phalcon\Db\Adapter*

Returns the default value to make the RDBM use the default value declared in the table definition

```
<?php
// Inserting a new robot with a valid default value for the column 'year'
$success = $connection->insert (
    "robots",
    [
        "Astro Boy",
        $connection->getDefaultValue ()
    ],
    [
        "name",
        "year",
    ]
);
```

public **getDescriptor** () inherited from *Phalcon\Db\Adapter*

Return descriptor used to connect to the active database

public *string* **getConnectionId** () inherited from *Phalcon\Db\Adapter*

Gets the active connection unique identifier

public **getSQLStatement** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object

public **getRealSQLStatement** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object without replace bound parameters

public *array* **getSQLBindTypes** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object

Class *Phalcon\Db\Adapter\Pdo\Sqlite*

extends abstract class *Phalcon\Db\Adapter\Pdo*

implements *Phalcon\Db\AdapterInterface*, *Phalcon\Events\EventsAwareInterface*

Specific functions for the Sqlite database system

```
<?php

use Phalcon\Db\Adapter\Pdo\Sqlite;

$connection = new Sqlite(
    [
        "dbname" => "/tmp/test.sqlite",
    ]
);
```

Methods

public **connect** ([array \$descriptor])

This method is automatically called in Phalcon\Db\Adapter\Pdo constructor. Call it when you need to restore a database connection.

public **describeColumns** (mixed \$table, [mixed \$schema])

Returns an array of Phalcon\Db\Column objects describing a table

```
<?php

print_r(
    $connection->describeColumns("posts")
);
```

public *Phalcon\Db\IndexInterface*[] **describeIndexes** (string \$table, [string \$schema])

Lists table indexes

```
<?php

print_r(
    $connection->describeIndexes("robots_parts")
);
```

public *Phalcon\Db\ReferenceInterface*[] **describeReferences** (string \$table, [string \$schema])

Lists table references

public **useExplicitIdValue** ()

Check whether the database system requires an explicit value for identity columns

public **getDefaultValue** ()

Returns the default value to make the RDBM use the default value declared in the table definition

```
<?php

// Inserting a new robot with a valid default value for the column 'year'
$success = $connection->insert(
    "robots",
    [
        "Astro Boy",
        $connection->getDefaultValue(),
    ],
);
```

```
[  
    "name",  
    "year",  
]  
);
```

public **__construct** (array \$descriptor) inherited from *Phalcon\Db\Adapter\Pdo*

Constructor for Phalcon\Db\Adapter\Pdo

public **prepare** (mixed \$sqlStatement) inherited from *Phalcon\Db\Adapter\Pdo*

Returns a PDO prepared statement to be executed with 'executePrepared'

```
<?php  
  
use Phalcon\Db\Column;  
  
$statement = $db->prepare(  
    "SELECT * FROM robots WHERE name = :name"  
);  
  
$result = $connection->executePrepared(  
    $statement,  
    [  
        "name" => "Voltron",  
    ],  
    [  
        "name" => Column::BIND_PARAM_INT,  
    ]  
);
```

public **PDOSTatement executePrepared** (PDOSTatement \$statement, array \$placeholders, array \$dataTypes) inherited from *Phalcon\Db\Adapter\Pdo*

Executes a prepared statement binding. This function uses integer indexes starting from zero

```
<?php  
  
use Phalcon\Db\Column;  
  
$statement = $db->prepare(  
    "SELECT * FROM robots WHERE name = :name"  
);  
  
$result = $connection->executePrepared(  
    $statement,  
    [  
        "name" => "Voltron",  
    ],  
    [  
        "name" => Column::BIND_PARAM_INT,  
    ]  
);
```

public **query** (mixed \$sqlStatement, [mixed \$bindParams], [mixed \$bindTypes]) inherited from *Phalcon\Db\Adapter\Pdo*

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server is returning rows

```
<?php

// Querying data
$resultset = $connection->query(
    "SELECT * FROM robots WHERE type = 'mechanical'"
);

$resultset = $connection->query(
    "SELECT * FROM robots WHERE type = ?",
    [
        "mechanical",
    ]
);
```

public **execute** (*mixed* \$sqlStatement, [*mixed* \$bindParam], [*mixed* \$bindTypes]) inherited from *Phalcon\Db\Adapter\Pdo*

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server doesn't return any rows

```
<?php

// Inserting data
$success = $connection->execute(
    "INSERT INTO robots VALUES (1, 'Astro Boy')"
);

$success = $connection->execute(
    "INSERT INTO robots VALUES (?, ?)",
    [
        1,
        "Astro Boy",
    ]
);
```

public **affectedRows** () inherited from *Phalcon\Db\Adapter\Pdo*

Returns the number of affected rows by the latest INSERT/UPDATE/DELETE executed in the database system

```
<?php

$connection->execute(
    "DELETE FROM robots"
);

echo $connection->affectedRows(), " were deleted";
```

public **close** () inherited from *Phalcon\Db\Adapter\Pdo*

Closes the active connection returning success. Phalcon automatically closes and destroys active connections when the request ends

public **escapeString** (*mixed* \$str) inherited from *Phalcon\Db\Adapter\Pdo*

Escapes a value to avoid SQL injections according to the active charset in the connection

```
<?php

$escapedStr = $connection->escapeString("some dangerous value");
```

public **convertBoundParams** (*mixed* \$sql, [*array* \$params]) inherited from *Phalcon\Db\Adapter\Pdo*

Converts bound parameters such as :name: or ?1 into PDO bind params ?

```
<?php
print_r(
    $connection->convertBoundParams(
        "SELECT * FROM robots WHERE name = :name:",
        [
            "Bender",
        ]
    )
);
```

public *int | boolean* **lastInsertId** ([*string* \$sequenceName]) inherited from *Phalcon\Db\Adapter\Pdo*

Returns the insert id for the auto_increment/serial column inserted in the latest executed SQL statement

```
<?php
// Inserting a new robot
$success = $connection->insert(
    "robots",
    [
        "Astro Boy",
        1952,
    ],
    [
        "name",
        "year",
    ]
);

// Getting the generated id
$id = $connection->lastInsertId();
```

public **begin** ([*mixed* \$nesting]) inherited from *Phalcon\Db\Adapter\Pdo*

Starts a transaction in the connection

public **rollback** ([*mixed* \$nesting]) inherited from *Phalcon\Db\Adapter\Pdo*

Rollbacks the active transaction in the connection

public **commit** ([*mixed* \$nesting]) inherited from *Phalcon\Db\Adapter\Pdo*

Commits the active transaction in the connection

public **getTransactionLevel** () inherited from *Phalcon\Db\Adapter\Pdo*

Returns the current transaction nesting level

public **isUnderTransaction** () inherited from *Phalcon\Db\Adapter\Pdo*

Checks whether the connection is under a transaction

```
<?php
$connection->begin();

// true
var_dump(
```

```
$connection->isUnderTransaction()
);
```

public **getInternalHandler** () inherited from *Phalcon\Db\Adapter\Pdo*

Return internal PDO handler

public array **getErrorInfo** () inherited from *Phalcon\Db\Adapter\Pdo*

Return the error info, if any

public **getDialectType** () inherited from *Phalcon\Db\Adapter*

Name of the dialect used

public **getType** () inherited from *Phalcon\Db\Adapter*

Type of database system the adapter is used for

public **getSqlVariables** () inherited from *Phalcon\Db\Adapter*

Active SQL bound parameter variables

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Db\Adapter*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Db\Adapter*

Returns the internal event manager

public **setDialect** (*Phalcon\Db\DialectInterface* \$dialect) inherited from *Phalcon\Db\Adapter*

Sets the dialect used to produce the SQL

public **getDialect** () inherited from *Phalcon\Db\Adapter*

Returns internal dialect instance

public **fetchOne** (*mixed* \$sqlQuery, [*mixed* \$fetchMode], [*mixed* \$bindParam], [*mixed* \$bindTypes]) inherited from *Phalcon\Db\Adapter*

Returns the first row in a SQL query result

```
<?php

// Getting first robot
$robot = $connection->fetchOne("SELECT * FROM robots");
print_r($robot);

// Getting first robot with associative indexes only
$robot = $connection->fetchOne("SELECT * FROM robots", \Phalcon\Db::FETCH_ASSOC);
print_r($robot);
```

public array **fetchAll** (*string* \$sqlQuery, [*int* \$fetchMode], [*array* \$bindParam], [*array* \$bindTypes]) inherited from *Phalcon\Db\Adapter*

Dumps the complete result of a query into an array

```
<?php

// Getting all robots with associative indexes only
$robots = $connection->fetchAll(
    "SELECT * FROM robots",
    \Phalcon\Db::FETCH_ASSOC
```

```
);

foreach ($robots as $robot) {
    print_r($robot);
}

// Getting all robots that contains word "robot" withing the name
$robots = $connection->fetchAll(
    "SELECT * FROM robots WHERE name LIKE :name",
    \Phalcon\Db::FETCH_ASSOC,
    [
        "name" => "%robot%",
    ]
);
foreach($robots as $robot) {
    print_r($robot);
}
```

public *string* | **fetchColumn** (*string* \$sqlQuery, [*array* \$placeholders], [*int* | *string* \$column]) inherited from *Phalcon\Db\Adapter*

Returns the n'th field of first row in a SQL query result

```
<?php

// Getting count of robots
$robotsCount = $connection->fetchColumn("SELECT count(*) FROM robots");
print_r($robotsCount);

// Getting name of last edited robot
$robot = $connection->fetchColumn(
    "SELECT id, name FROM robots order by modified desc",
    1
);
print_r($robot);
```

public *boolean* **insert** (*string* | *array* \$table, *array* \$values, [*array* \$fields], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Inserts data into a table using custom RDBMS SQL syntax

```
<?php

// Inserting a new robot
$success = $connection->insert(
    "robots",
    ["Astro Boy", 1952],
    ["name", "year"]
);

// Next SQL sentence is sent to the database system
INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public *boolean* **insertAsDict** (*string* \$table, *array* \$data, [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Inserts data into a table using custom RBDM SQL syntax

```
<?php
```

```
// Inserting a new robot
$success = $connection->insertAsDict(
    "robots",
    [
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);

// Next SQL sentence is sent to the database system
INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public **boolean update** (*string* | *array* \$table, *array* \$fields, *array* \$values, [*string* | *array* \$whereCondition], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Updates data on a table using custom RBDM SQL syntax

```
<?php

// Updating existing robot
$success = $connection->update(
    "robots",
    ["name"],
    ["New Astro Boy"],
    "id = 101"
);

// Next SQL sentence is sent to the database system
UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101

// Updating existing robot with array condition and $dataTypes
$success = $connection->update(
    "robots",
    ["name"],
    ["New Astro Boy"],
    [
        "conditions" => "id = ?",
        "bind"        => [$some_unsafe_id],
        "bindTypes"   => [PDO::PARAM_INT], // use only if you use $dataTypes param
    ],
    [
        PDO::PARAM_STR
    ]
);
```

Warning! If \$whereCondition is string it not escaped.

public **boolean updateAsDict** (*string* \$table, *array* \$data, [*string* \$whereCondition], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Updates data on a table using custom RBDM SQL syntax Another, more convenient syntax

```
<?php

// Updating existing robot
$success = $connection->updateAsDict(
    "robots",
    [
        "name" => "New Astro Boy",
```

```
        ],
        "id = 101"
    );

    // Next SQL sentence is sent to the database system
    UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101
```

public **boolean delete** (*string* | *array* \$table, [*string* \$whereCondition], [*array* \$placeholders], [*array* \$dataTypes]) inherited from *Phalcon\Db\Adapter*

Deletes data from a table using custom RBDM SQL syntax

```
<?php

// Deleting existing robot
$success = $connection->delete(
    "robots",
    "id = 101"
);

// Next SQL sentence is generated
DELETE FROM `robots` WHERE `id` = 101
```

public **escapeIdentifier** (*array* | *string* \$identifier) inherited from *Phalcon\Db\Adapter*

Escapes a column/table/schema name

```
<?php

$escapedTable = $connection->escapeIdentifier(
    "robots"
);

$escapedTable = $connection->escapeIdentifier(
    [
        "store",
        "robots",
    ]
);
```

public **string getColumnList** (*array* \$columnList) inherited from *Phalcon\Db\Adapter*

Gets a list of columns

public **limit** (*mixed* \$sqlQuery, *mixed* \$number) inherited from *Phalcon\Db\Adapter*

Appends a LIMIT clause to \$sqlQuery argument

```
<?php

echo $connection->limit("SELECT * FROM robots", 5);
```

public **tableExists** (*mixed* \$tableName, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Generates SQL checking for the existence of a schema.table

```
<?php

var_dump(
```



```
$connection->tableExists("blog", "posts")
);
```

public **viewExists** (*mixed* \$viewName, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Generates SQL checking for the existence of a schema.view

```
<?php

var_dump (
    $connection->viewExists("active_users", "posts")
);
```

public **forUpdate** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Adapter*

Returns a SQL modified with a FOR UPDATE clause

public **sharedLock** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Adapter*

Returns a SQL modified with a LOCK IN SHARE MODE clause

public **createTable** (*mixed* \$tableName, *mixed* \$schemaName, *array* \$definition) inherited from *Phalcon\Db\Adapter*

Creates a table

public **dropTable** (*mixed* \$tableName, [*mixed* \$schemaName], [*mixed* \$ifExists]) inherited from *Phalcon\Db\Adapter*

Drops a table from a schema/database

public **createView** (*mixed* \$viewName, *array* \$definition, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Creates a view

public **dropView** (*mixed* \$viewName, [*mixed* \$schemaName], [*mixed* \$ifExists]) inherited from *Phalcon\Db\Adapter*

Drops a view

public **addColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column) inherited from *Phalcon\Db\Adapter*

Adds a column to a table

public **modifyColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column, [*Phalcon\Db\ColumnInterface* \$currentColumn]) inherited from *Phalcon\Db\Adapter*

Modifies a table column based on a definition

public **dropColumn** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$columnName) inherited from *Phalcon\Db\Adapter*

Drops a column from a table

public **addIndex** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index) inherited from *Phalcon\Db\Adapter*

Adds an index to a table

public **dropIndex** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$indexName) inherited from *Phalcon\Db\Adapter*

Drop an index from a table

public **addPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index) inherited from *Phalcon\Db\Adapter*

Adds a primary key to a table

public **dropPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName) inherited from *Phalcon\Db\Adapter*

Drops a table's primary key

public **addForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ReferenceInterface* \$reference) inherited from *Phalcon\Db\Adapter*

Adds a foreign key to a table

public **dropForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$referenceName) inherited from *Phalcon\Db\Adapter*

Drops a foreign key from a table

public **getColumnDefinition** (*Phalcon\Db\ColumnInterface* \$column) inherited from *Phalcon\Db\Adapter*

Returns the SQL column definition from a column

public **listTables** ([*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

List all tables on a database

```
<?php
print_r(
    $connection->listTables("blog")
);
```

public **listViews** ([*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

List all views on a database

```
<?php
print_r(
    $connection->listViews("blog")
);
```

public **tableOptions** (*mixed* \$tableName, [*mixed* \$schemaName]) inherited from *Phalcon\Db\Adapter*

Gets creation options from a table

```
<?php
print_r(
    $connection->tableOptions("robots")
);
```

public **createSavepoint** (*mixed* \$name) inherited from *Phalcon\Db\Adapter*

Creates a new savepoint

public **releaseSavepoint** (*mixed* \$name) inherited from *Phalcon\Db\Adapter*

Releases given savepoint

public **rollbackSavepoint** (*mixed* \$name) inherited from *Phalcon\Db\Adapter*

Rollbacks given savepoint

public **setNestedTransactionsWithSavepoints** (*mixed* \$nestedTransactionsWithSavepoints) inherited from *Phalcon\Db\Adapter*

Set if nested transactions should use savepoints

public **isNestedTransactionsWithSavepoints** () inherited from *Phalcon\Db\Adapter*

Returns if nested transactions should use savepoints

public **getNestedTransactionSavepointName** () inherited from *Phalcon\Db\Adapter*

Returns the savepoint name to use for nested transactions

public **getDefaultIdValue** () inherited from *Phalcon\Db\Adapter*

Returns the default identity value to be inserted in an identity column

```
<?php

// Inserting a new robot with a valid default value for the column 'id'
$success = $connection->insert(
    "robots",
    [
        $connection->getDefaultIdValue(),
        "Astro Boy",
        1952,
    ],
    [
        "id",
        "name",
        "year",
    ]
);
```

public **supportSequences** () inherited from *Phalcon\Db\Adapter*

Check whether the database system requires a sequence to produce auto-numeric values

public **getDescriptor** () inherited from *Phalcon\Db\Adapter*

Return descriptor used to connect to the active database

public *string* **getConnectionId** () inherited from *Phalcon\Db\Adapter*

Gets the active connection unique identifier

public **getSQLStatement** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object

public **getRealSQLStatement** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object without replace bound parameters

public *array* **getSQLBindTypes** () inherited from *Phalcon\Db\Adapter*

Active SQL statement in the object

Class **Phalcon\Db\Column**

implements *Phalcon\Db\ColumnInterface*

Allows to define columns to be used on create or alter table operations

```
<?php

use Phalcon\Db\Column as Column;

// Column definition
```

```
$column = new Column(
    "id",
    [
        "type"           => Column::TYPE_INTEGER,
        "size"           => 10,
        "unsigned"       => true,
        "notNull"        => true,
        "autoIncrement"  => true,
        "first"          => true,
    ]
);

// Add column to existing table
$connection->addColumn("robots", null, $column);
```

Constants

integer **TYPE_INTEGER**

integer **TYPE_DATE**

integer **TYPE_VARCHAR**

integer **TYPE_DECIMAL**

integer **TYPE_DATETIME**

integer **TYPE_CHAR**

integer **TYPE_TEXT**

integer **TYPE_FLOAT**

integer **TYPE_BOOLEAN**

integer **TYPE_DOUBLE**

integer **TYPE_TINYBLOB**

integer **TYPE_BLOB**

integer **TYPE_MEDIUMBLOB**

integer **TYPE_LONGBLOB**

integer **TYPE_BIGINTEGER**

integer **TYPE_JSON**

integer **TYPE_JSONB**

integer **TYPE_TIMESTAMP**

integer **BIND_PARAM_NULL**

integer **BIND_PARAM_INT**

integer **BIND_PARAM_STR**

integer **BIND_PARAM_BLOB**

integer **BIND_PARAM_BOOL**

integer **BIND_PARAM_DECIMAL**

integer **BIND_SKIP**

Methods

public **getName** ()

Column's name

public **getSchemaName** ()

Schema which table related is

public **getType** ()

Column data type

public **getTypeReference** ()

Column data type reference

public **getTypeValues** ()

Column data type values

public **getSize** ()

Integer column size

public **getScale** ()

Integer column number scale

public **getDefault** ()

Default column value

public **__construct** (*mixed* \$name, *array* \$definition)

Phalcon\Db\Column constructor

public **isUnsigned** ()

Returns true if number column is unsigned

public **isNotNull** ()

Not null

public **isPrimary** ()

Column is part of the primary key?

public **isAutoIncrement** ()

Auto-Increment

public **isNumeric** ()

Check whether column have an numeric type

public **isFirst** ()

Check whether column have first position in table

public *string* **getAfterPosition** ()

Check whether field absolute to position in table

public **getBindType** ()

Returns the type of bind handling

public static **__set_state** (*array* \$data)

Restores the internal state of a Phalcon\Db\Column object

public **hasDefault** ()

Check whether column has default value

Abstract class Phalcon\Db\Dialect

implements Phalcon\Db\DialectInterface

This is the base class to each database dialect. This implements common methods to transform intermediate code into its RDBMS related syntax

Methods

public **registerCustomFunction** (*mixed* \$name, *mixed* \$customFunction)

Registers custom SQL functions

public **getCustomFunctions** ()

Returns registered functions

final public **escapeSchema** (*mixed* \$str, [*mixed* \$escapeChar])

Escape Schema

final public **escape** (*mixed* \$str, [*mixed* \$escapeChar])

Escape identifiers

public **limit** (*mixed* \$sqlQuery, *mixed* \$number)

Generates the SQL for LIMIT clause

```
<?php
$sql = $dialect->limit("SELECT * FROM robots", 10);
echo $sql; // SELECT * FROM robots LIMIT 10

$sql = $dialect->limit("SELECT * FROM robots", [10, 50]);
echo $sql; // SELECT * FROM robots LIMIT 10 OFFSET 50
```

public **forUpdate** (*mixed* \$sqlQuery)

Returns a SQL modified with a FOR UPDATE clause

```
<?php
$sql = $dialect->forUpdate("SELECT * FROM robots");
echo $sql; // SELECT * FROM robots FOR UPDATE
```

public **sharedLock** (*mixed* \$sqlQuery)

Returns a SQL modified with a LOCK IN SHARE MODE clause

```
<?php

$sql = $dialect->sharedLock("SELECT * FROM robots");
echo $sql; // SELECT * FROM robots LOCK IN SHARE MODE
```

final public **getColumnList** (array \$columnList, [mixed \$escapeChar], [mixed \$bindCounts])

Gets a list of columns with escaped identifiers

```
<?php

echo $dialect->getColumnList (
    [
        "column1",
        "column",
    ]
);
```

final public **getSqlColumn** (mixed \$column, [mixed \$escapeChar], [mixed \$bindCounts])

Resolve Column expressions

public **getSqlExpression** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts])

Transforms an intermediate representation for an expression into a database system valid expression

final public **getSqlTable** (mixed \$table, [mixed \$escapeChar])

Transform an intermediate representation of a schema/table into a database system valid expression

public **select** (array \$definition)

Builds a SELECT statement

public **supportsSavepoints** ()

Checks whether the platform supports savepoints

public **supportsReleaseSavepoints** ()

Checks whether the platform supports releasing savepoints.

public **createSavepoint** (mixed \$name)

Generate SQL to create a new savepoint

public **releaseSavepoint** (mixed \$name)

Generate SQL to release a savepoint

public **rollbackSavepoint** (mixed \$name)

Generate SQL to rollback a savepoint

final protected **getSqlExpressionScalar** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts])

Resolve Column expressions

final protected **getSqlExpressionObject** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts])

Resolve object expressions

final protected **getSqlExpressionQualified** (array \$expression, [mixed \$escapeChar])

Resolve qualified expressions

final protected **getSqlExpressionBinaryOperations** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts])

Resolve binary operations expressions

final protected **getSqlExpressionUnaryOperations** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])

Resolve unary operations expressions

final protected **getSqlExpressionFunctionCall** (*array* \$expression, *mixed* \$escapeChar, [*mixed* \$bindCounts])

Resolve function calls

final protected **getSqlExpressionList** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])

Resolve Lists

final protected **getSqlExpressionAll** (*array* \$expression, [*mixed* \$escapeChar])

Resolve *

final protected **getSqlExpressionCastValue** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])

Resolve CAST of values

final protected **getSqlExpressionConvertValue** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])

Resolve CONVERT of values encodings

final protected **getSqlExpressionCase** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])

Resolve CASE expressions

final protected **getSqlExpressionFrom** (*mixed* \$expression, [*mixed* \$escapeChar])

Resolve a FROM clause

final protected **getSqlExpressionJoins** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])

Resolve a JOINS clause

final protected **getSqlExpressionWhere** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])

Resolve a WHERE clause

final protected **getSqlExpressionGroupBy** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])

Resolve a GROUP BY clause

final protected **getSqlExpressionHaving** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])

Resolve a HAVING clause

final protected **getSqlExpressionOrderBy** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])

Resolve an ORDER BY clause

final protected **getSqlExpressionLimit** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])

Resolve a LIMIT clause

protected **prepareColumnAlias** (*mixed* \$qualified, [*mixed* \$alias], [*mixed* \$escapeChar])

Prepares column for this RDBMS

protected **prepareTable** (*mixed* \$table, [*mixed* \$schema], [*mixed* \$alias], [*mixed* \$escapeChar])

Prepares table for this RDBMS

protected **prepareQualified** (*mixed* \$column, [*mixed* \$domain], [*mixed* \$escapeChar])

Prepares qualified for this RDBMS

abstract public **getColumnDefinition** (*Phalcon\Db\ColumnInterface* \$column) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **addColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **modifyColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column, [*Phalcon\Db\ColumnInterface* \$currentColumn]) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **dropColumn** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$columnName) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **addIndex** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db/IndexInterface* \$index) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **dropIndex** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$indexName) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **addPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db/IndexInterface* \$index) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **dropPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **addForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db/ReferenceInterface* \$reference) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **dropForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$referenceName) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **createTable** (*mixed* \$tableName, *mixed* \$schemaName, *array* \$definition) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **createView** (*mixed* \$viewName, *array* \$definition, [*mixed* \$schemaName]) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **dropTable** (*mixed* \$tableName, *mixed* \$schemaName) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **dropView** (*mixed* \$viewName, [*mixed* \$schemaName], [*mixed* \$ifExists]) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **tableExists** (*mixed* \$tableName, [*mixed* \$schemaName]) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **viewExists** (*mixed* \$viewName, [*mixed* \$schemaName]) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **describeColumns** (*mixed* \$table, [*mixed* \$schema]) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **listTables** ([*mixed* \$schemaName]) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **describeIndexes** (*mixed* \$table, [*mixed* \$schema]) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **describeReferences** (*mixed* \$table, [*mixed* \$schema]) inherited from *Phalcon\Db\DialectInterface*

...

abstract public **tableOptions** (*mixed* \$table, [*mixed* \$schema]) inherited from *Phalcon\Db\DialectInterface*

...

Class *Phalcon\Db\Dialect\Mysql*

extends abstract class *Phalcon\Db\Dialect*

implements *Phalcon\Db\DialectInterface*

Generates database specific SQL for the MySQL RDBMS

Methods

public **getColumnDefinition** (*Phalcon\Db\ColumnInterface* \$column)

Gets the column name in MySQL

public **addColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column)

Generates SQL to add a column to a table

public **modifyColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column, [*Phalcon\Db\ColumnInterface* \$currentColumn])

Generates SQL to modify a column in a table

public **dropColumn** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$columnName)

Generates SQL to delete a column from a table

public **addIndex** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db/IndexInterface* \$index)

Generates SQL to add an index to a table

public **dropIndex** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$indexName)

Generates SQL to delete an index from a table

public **addPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db/IndexInterface* \$index)

Generates SQL to add the primary key to a table

public **dropPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName)

Generates SQL to delete primary key from a table

public **addForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ReferenceInterface* \$reference)

Generates SQL to add an index to a table

public **dropForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$referenceName)

Generates SQL to delete a foreign key from a table

public **createTable** (*mixed* \$tableName, *mixed* \$schemaName, *array* \$definition)

Generates SQL to create a table

public **truncateTable** (*mixed* \$tableName, *mixed* \$schemaName)

Generates SQL to truncate a table

public **dropTable** (*mixed* \$tableName, [*mixed* \$schemaName], [*mixed* \$ifExists])

Generates SQL to drop a table

public **createView** (*mixed* \$viewName, *array* \$definition, [*mixed* \$schemaName])

Generates SQL to create a view

public **dropView** (*mixed* \$viewName, [*mixed* \$schemaName], [*mixed* \$ifExists])

Generates SQL to drop a view

public **tableExists** (*mixed* \$tableName, [*mixed* \$schemaName])

Generates SQL checking for the existence of a schema.table

```
<?php
echo $dialect->tableExists("posts", "blog");
echo $dialect->tableExists("posts");
```

public **viewExists** (*mixed* \$viewName, [*mixed* \$schemaName])

Generates SQL checking for the existence of a schema.view

public **describeColumns** (*mixed* \$table, [*mixed* \$schema])

Generates SQL describing a table

```
<?php
print_r(
    $dialect->describeColumns("posts")
);
```

public **listTables** ([*mixed* \$schemaName])

List all tables in database

```
<?php
print_r(
    $dialect->listTables("blog")
);
```

public **listViews** ([*mixed* \$schemaName])

Generates the SQL to list all views of a schema or user

public **describeIndexes** (*mixed* \$table, [*mixed* \$schema])

Generates SQL to query indexes on a table

public **describeReferences** (*mixed* \$table, [*mixed* \$schema])

Generates SQL to query foreign keys on a table

public **tableOptions** (*mixed* \$table, [*mixed* \$schema])

Generates the SQL to describe the table creation options

protected **_getTableOptions** (array \$definition)

Generates SQL to add the table creation options

public **registerCustomFunction** (*mixed* \$name, *mixed* \$customFunction) inherited from *Phalcon\Db\Dialect*

Registers custom SQL functions

public **getCustomFunctions** () inherited from *Phalcon\Db\Dialect*

Returns registered functions

final public **escapeSchema** (*mixed* \$str, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Escape Schema

final public **escape** (*mixed* \$str, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Escape identifiers

public **limit** (*mixed* \$sqlQuery, *mixed* \$number) inherited from *Phalcon\Db\Dialect*

Generates the SQL for LIMIT clause

```
<?php

$sql = $dialect->limit("SELECT * FROM robots", 10);
echo $sql; // SELECT * FROM robots LIMIT 10

$sql = $dialect->limit("SELECT * FROM robots", [10, 50]);
echo $sql; // SELECT * FROM robots LIMIT 10 OFFSET 50
```

public **forUpdate** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Dialect*

Returns a SQL modified with a FOR UPDATE clause

```
<?php

$sql = $dialect->forUpdate("SELECT * FROM robots");
echo $sql; // SELECT * FROM robots FOR UPDATE
```

public **sharedLock** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Dialect*

Returns a SQL modified with a LOCK IN SHARE MODE clause

```
<?php

$sql = $dialect->sharedLock("SELECT * FROM robots");
echo $sql; // SELECT * FROM robots LOCK IN SHARE MODE
```

final public **getColumnList** (array \$columnList, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Gets a list of columns with escaped identifiers

```
<?php

echo $dialect->getColumnList (
    [
        "column1",
        "column",
    ]
);
```

final public **getSqlColumn** (mixed \$column, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve Column expressions

public **getSqlExpression** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Transforms an intermediate representation for an expression into a database system valid expression

final public **getSqlTable** (mixed \$table, [mixed \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Transform an intermediate representation of a schema/table into a database system valid expression

public **select** (array \$definition) inherited from *Phalcon\Db\Dialect*

Builds a SELECT statement

public **supportsSavepoints** () inherited from *Phalcon\Db\Dialect*

Checks whether the platform supports savepoints

public **supportsReleaseSavepoints** () inherited from *Phalcon\Db\Dialect*

Checks whether the platform supports releasing savepoints.

public **createSavepoint** (mixed \$name) inherited from *Phalcon\Db\Dialect*

Generate SQL to create a new savepoint

public **releaseSavepoint** (mixed \$name) inherited from *Phalcon\Db\Dialect*

Generate SQL to release a savepoint

public **rollbackSavepoint** (mixed \$name) inherited from *Phalcon\Db\Dialect*

Generate SQL to rollback a savepoint

final protected **getSqlExpressionScalar** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve Column expressions

final protected **getSqlExpressionObject** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve object expressions

final protected **getSqlExpressionQualified** (array \$expression, [mixed \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Resolve qualified expressions

final protected **getSqlExpressionBinaryOperations** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve binary operations expressions

final protected **getSqlExpressionUnaryOperations** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve unary operations expressions

final protected **getSqlExpressionFunctionCall** (*array* \$expression, *mixed* \$escapeChar, [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve function calls

final protected **getSqlExpressionList** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve Lists

final protected **getSqlExpressionAll** (*array* \$expression, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Resolve *

final protected **getSqlExpressionCastValue** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve CAST of values

final protected **getSqlExpressionConvertValue** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve CONVERT of values encodings

final protected **getSqlExpressionCase** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve CASE expressions

final protected **getSqlExpressionFrom** (*mixed* \$expression, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Resolve a FROM clause

final protected **getSqlExpressionJoins** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a JOINS clause

final protected **getSqlExpressionWhere** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a WHERE clause

final protected **getSqlExpressionGroupBy** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a GROUP BY clause

final protected **getSqlExpressionHaving** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a HAVING clause

final protected **getSqlExpressionOrderBy** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve an ORDER BY clause

final protected **getSqlExpressionLimit** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a LIMIT clause

protected **prepareColumnAlias** (*mixed* \$qualified, [*mixed* \$alias], [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Prepares column for this RDBMS

protected **prepareTable** (*mixed* \$table, [*mixed* \$schema], [*mixed* \$alias], [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Prepares table for this RDBMS

protected **prepareQualified** (*mixed* \$column, [*mixed* \$domain], [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Prepares qualified for this RDBMS

Class *Phalcon\Db\Dialect\Postgresql*

extends abstract class *Phalcon\Db\Dialect*

implements *Phalcon\Db\DialectInterface*

Generates database specific SQL for the PostgreSQL RDBMS

Methods

public **getColumnDefinition** (*Phalcon\Db\ColumnInterface* \$column)

Gets the column name in PostgreSQL

public **addColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column)

Generates SQL to add a column to a table

public **modifyColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column, [*Phalcon\Db\ColumnInterface* \$currentColumn])

Generates SQL to modify a column in a table

public **dropColumn** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$columnName)

Generates SQL to delete a column from a table

public **addIndex** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db/IndexInterface* \$index)

Generates SQL to add an index to a table

public **dropIndex** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$indexName)

Generates SQL to delete an index from a table

public **addPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db/IndexInterface* \$index)

Generates SQL to add the primary key to a table

public **dropPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName)

Generates SQL to delete primary key from a table

public **addForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ReferenceInterface* \$reference)

Generates SQL to add an index to a table

public **dropForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$referenceName)

Generates SQL to delete a foreign key from a table

public **createTable** (*mixed* \$tableName, *mixed* \$schemaName, *array* \$definition)

Generates SQL to create a table

public **truncateTable** (*mixed* \$tableName, *mixed* \$schemaName)

Generates SQL to truncate a table

public **dropTable** (*mixed* \$tableName, [*mixed* \$schemaName], [*mixed* \$ifExists])

Generates SQL to drop a table

public **createView** (*mixed* \$viewName, *array* \$definition, [*mixed* \$schemaName])

Generates SQL to create a view

public **dropView** (*mixed* \$viewName, [*mixed* \$schemaName], [*mixed* \$ifExists])

Generates SQL to drop a view

public **tableExists** (*mixed* \$tableName, [*mixed* \$schemaName])

Generates SQL checking for the existence of a schema.table

```
<?php
echo $dialect->tableExists("posts", "blog");

echo $dialect->tableExists("posts");
```

public **viewExists** (*mixed* \$viewName, [*mixed* \$schemaName])

Generates SQL checking for the existence of a schema.view

public **describeColumns** (*mixed* \$table, [*mixed* \$schema])

Generates SQL describing a table

```
<?php
print_r(
    $dialect->describeColumns("posts")
);
```

public **listTables** ([*mixed* \$schemaName])

List all tables in database

```
<?php
print_r(
    $dialect->listTables("blog")
);
```

public *string* **listViews** ([*string* \$schemaName])

Generates the SQL to list all views of a schema or user

public **describeIndexes** (*mixed* \$table, [*mixed* \$schema])

Generates SQL to query indexes on a table

public **describeReferences** (*mixed* \$table, [*mixed* \$schema])

Generates SQL to query foreign keys on a table

public **tableOptions** (*mixed* \$table, [*mixed* \$schema])

Generates the SQL to describe the table creation options

protected **_castDefault** (*Phalcon\Db\ColumnInterface* \$column)

...

protected **_getTableOptions** (*array* \$definition)

...

public **registerCustomFunction** (*mixed* \$name, *mixed* \$customFunction) inherited from *Phalcon\Db\Dialect*

Registers custom SQL functions

public **getCustomFunctions** () inherited from *Phalcon\Db\Dialect*

Returns registered functions

final public **escapeSchema** (*mixed* \$str, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Escape Schema

final public **escape** (*mixed* \$str, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Escape identifiers

public **limit** (*mixed* \$sqlQuery, *mixed* \$number) inherited from *Phalcon\Db\Dialect*

Generates the SQL for LIMIT clause

```
<?php
$sql = $dialect->limit("SELECT * FROM robots", 10);
echo $sql; // SELECT * FROM robots LIMIT 10

$sql = $dialect->limit("SELECT * FROM robots", [10, 50]);
echo $sql; // SELECT * FROM robots LIMIT 10 OFFSET 50
```

public **forUpdate** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Dialect*

Returns a SQL modified with a FOR UPDATE clause

```
<?php
$sql = $dialect->forUpdate("SELECT * FROM robots");
echo $sql; // SELECT * FROM robots FOR UPDATE
```

public **sharedLock** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Dialect*

Returns a SQL modified with a LOCK IN SHARE MODE clause

```
<?php
$sql = $dialect->sharedLock("SELECT * FROM robots");
echo $sql; // SELECT * FROM robots LOCK IN SHARE MODE
```

final public **getColumnList** (array \$columnList, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Gets a list of columns with escaped identifiers

```
<?php
echo $dialect->getColumnList (
    [
        "column1",
        "column",
    ]
);
```

final public **getSqlColumn** (mixed \$column, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve Column expressions

public **getSqlExpression** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Transforms an intermediate representation for an expression into a database system valid expression

final public **getSqlTable** (mixed \$table, [mixed \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Transform an intermediate representation of a schema/table into a database system valid expression

public **select** (array \$definition) inherited from *Phalcon\Db\Dialect*

Builds a SELECT statement

public **supportsSavepoints** () inherited from *Phalcon\Db\Dialect*

Checks whether the platform supports savepoints

public **supportsReleaseSavepoints** () inherited from *Phalcon\Db\Dialect*

Checks whether the platform supports releasing savepoints.

public **createSavepoint** (mixed \$name) inherited from *Phalcon\Db\Dialect*

Generate SQL to create a new savepoint

public **releaseSavepoint** (mixed \$name) inherited from *Phalcon\Db\Dialect*

Generate SQL to release a savepoint

public **rollbackSavepoint** (mixed \$name) inherited from *Phalcon\Db\Dialect*

Generate SQL to rollback a savepoint

final protected **getSqlExpressionScalar** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve Column expressions

final protected **getSqlExpressionObject** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve object expressions

final protected **getSqlExpressionQualified** (array \$expression, [mixed \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Resolve qualified expressions

final protected **getSqlExpressionBinaryOperations** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])
inherited from *Phalcon\Db\Dialect*

Resolve binary operations expressions

final protected **getSqlExpressionUnaryOperations** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts])
inherited from *Phalcon\Db\Dialect*

Resolve unary operations expressions

final protected **getSqlExpressionFunctionCall** (*array* \$expression, *mixed* \$escapeChar, [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve function calls

final protected **getSqlExpressionList** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve Lists

final protected **getSqlExpressionAll** (*array* \$expression, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Resolve *

final protected **getSqlExpressionCastValue** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve CAST of values

final protected **getSqlExpressionConvertValue** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve CONVERT of values encodings

final protected **getSqlExpressionCase** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve CASE expressions

final protected **getSqlExpressionFrom** (*mixed* \$expression, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Resolve a FROM clause

final protected **getSqlExpressionJoins** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a JOINS clause

final protected **getSqlExpressionWhere** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a WHERE clause

final protected **getSqlExpressionGroupBy** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a GROUP BY clause

final protected **getSqlExpressionHaving** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a HAVING clause

final protected **getSqlExpressionOrderBy** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve an ORDER BY clause

final protected **getSqlExpressionLimit** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a LIMIT clause

protected **prepareColumnAlias** (*mixed* \$qualified, [*mixed* \$alias], [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Prepares column for this RDBMS

protected **prepareTable** (*mixed* \$table, [*mixed* \$schema], [*mixed* \$alias], [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Prepares table for this RDBMS

protected **prepareQualified** (*mixed* \$column, [*mixed* \$domain], [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Prepares qualified for this RDBMS

Class *Phalcon\Db\Dialect\Sqlite*

extends abstract class *Phalcon\Db\Dialect*

implements *Phalcon\Db\DialectInterface*

Generates database specific SQL for the Sqlite RDBMS

Methods

public **getColumnDefinition** (*Phalcon\Db\ColumnInterface* \$column)

Gets the column name in SQLite

public **addColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column)

Generates SQL to add a column to a table

public **modifyColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column, [*Phalcon\Db\ColumnInterface* \$currentColumn])

Generates SQL to modify a column in a table

public **dropColumn** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$columnName)

Generates SQL to delete a column from a table

public **addIndex** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index)

Generates SQL to add an index to a table

public **dropIndex** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$indexName)

Generates SQL to delete an index from a table

public **addPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\IndexInterface* \$index)

Generates SQL to add the primary key to a table

public **dropPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName)

Generates SQL to delete primary key from a table

public **addForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ReferenceInterface* \$reference)

Generates SQL to add an index to a table

public **dropForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$referenceName)

Generates SQL to delete a foreign key from a table

public **createTable** (*mixed* \$tableName, *mixed* \$schemaName, *array* \$definition)

Generates SQL to create a table

public **truncateTable** (*mixed* \$tableName, *mixed* \$schemaName)

Generates SQL to truncate a table

public **dropTable** (*mixed* \$tableName, [*mixed* \$schemaName], [*mixed* \$ifExists])

Generates SQL to drop a table

public **createView** (*mixed* \$viewName, *array* \$definition, [*mixed* \$schemaName])

Generates SQL to create a view

public **dropView** (*mixed* \$viewName, [*mixed* \$schemaName], [*mixed* \$ifExists])

Generates SQL to drop a view

public **tableExists** (*mixed* \$tableName, [*mixed* \$schemaName])

Generates SQL checking for the existence of a schema.table

```
<?php
echo $dialect->tableExists("posts", "blog");
echo $dialect->tableExists("posts");
```

public **viewExists** (*mixed* \$viewName, [*mixed* \$schemaName])

Generates SQL checking for the existence of a schema.view

public **describeColumns** (*mixed* \$table, [*mixed* \$schema])

Generates SQL describing a table

```
<?php
print_r(
    $dialect->describeColumns("posts")
);
```

public **listTables** ([*mixed* \$schemaName])

List all tables in database

```
<?php
print_r(
    $dialect->listTables("blog")
);
```

public **listViews** ([*mixed* \$schemaName])

Generates the SQL to list all views of a schema or user

public **listIndexesSql** (*mixed* \$table, [*mixed* \$schema], [*mixed* \$keyName])

Generates the SQL to get query list of indexes

```
<?php

print_r(
    $dialect->listIndexesSql("blog")
);
```

public **describeIndexes** (*mixed* \$table, [*mixed* \$schema])

Generates SQL to query indexes on a table

public **describeIndex** (*mixed* \$index)

Generates SQL to query indexes detail on a table

public **describeReferences** (*mixed* \$table, [*mixed* \$schema])

Generates SQL to query foreign keys on a table

public **tableOptions** (*mixed* \$table, [*mixed* \$schema])

Generates the SQL to describe the table creation options

public **registerCustomFunction** (*mixed* \$name, *mixed* \$customFunction) inherited from *Phalcon\Db\Dialect*

Registers custom SQL functions

public **getCustomFunctions** () inherited from *Phalcon\Db\Dialect*

Returns registered functions

final public **escapeSchema** (*mixed* \$str, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Escape Schema

final public **escape** (*mixed* \$str, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Escape identifiers

public **limit** (*mixed* \$sqlQuery, *mixed* \$number) inherited from *Phalcon\Db\Dialect*

Generates the SQL for LIMIT clause

```
<?php

$sql = $dialect->limit("SELECT * FROM robots", 10);
echo $sql; // SELECT * FROM robots LIMIT 10

$sql = $dialect->limit("SELECT * FROM robots", [10, 50]);
echo $sql; // SELECT * FROM robots LIMIT 10 OFFSET 50
```

public **forUpdate** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Dialect*

Returns a SQL modified with a FOR UPDATE clause

```
<?php

$sql = $dialect->forUpdate("SELECT * FROM robots");
echo $sql; // SELECT * FROM robots FOR UPDATE
```

public **sharedLock** (*mixed* \$sqlQuery) inherited from *Phalcon\Db\Dialect*

Returns a SQL modified with a LOCK IN SHARE MODE clause

```
<?php

$sql = $dialect->sharedLock("SELECT * FROM robots");
echo $sql; // SELECT * FROM robots LOCK IN SHARE MODE
```

final public **getColumnList** (array \$columnList, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Gets a list of columns with escaped identifiers

```
<?php

echo $dialect->getColumnList (
    [
        "column1",
        "column",
    ]
);
```

final public **getSqlColumn** (mixed \$column, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve Column expressions

public **getSqlExpression** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Transforms an intermediate representation for an expression into a database system valid expression

final public **getSqlTable** (mixed \$table, [mixed \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Transform an intermediate representation of a schema/table into a database system valid expression

public **select** (array \$definition) inherited from *Phalcon\Db\Dialect*

Builds a SELECT statement

public **supportsSavepoints** () inherited from *Phalcon\Db\Dialect*

Checks whether the platform supports savepoints

public **supportsReleaseSavepoints** () inherited from *Phalcon\Db\Dialect*

Checks whether the platform supports releasing savepoints.

public **createSavepoint** (mixed \$name) inherited from *Phalcon\Db\Dialect*

Generate SQL to create a new savepoint

public **releaseSavepoint** (mixed \$name) inherited from *Phalcon\Db\Dialect*

Generate SQL to release a savepoint

public **rollbackSavepoint** (mixed \$name) inherited from *Phalcon\Db\Dialect*

Generate SQL to rollback a savepoint

final protected **getSqlExpressionScalar** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve Column expressions

final protected **getSqlExpressionObject** (array \$expression, [mixed \$escapeChar], [mixed \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve object expressions

final protected **getSqlExpressionQualified** (*array* \$expression, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Resolve qualified expressions

final protected **getSqlExpressionBinaryOperations** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve binary operations expressions

final protected **getSqlExpressionUnaryOperations** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve unary operations expressions

final protected **getSqlExpressionFunctionCall** (*array* \$expression, *mixed* \$escapeChar, [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve function calls

final protected **getSqlExpressionList** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve Lists

final protected **getSqlExpressionAll** (*array* \$expression, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Resolve *

final protected **getSqlExpressionCastValue** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve CAST of values

final protected **getSqlExpressionConvertValue** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve CONVERT of values encodings

final protected **getSqlExpressionCase** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve CASE expressions

final protected **getSqlExpressionFrom** (*mixed* \$expression, [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Resolve a FROM clause

final protected **getSqlExpressionJoins** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a JOINS clause

final protected **getSqlExpressionWhere** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a WHERE clause

final protected **getSqlExpressionGroupBy** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a GROUP BY clause

final protected **getSqlExpressionHaving** (*array* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a HAVING clause

final protected **getSqlExpressionOrderBy** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve an ORDER BY clause

final protected **getSqlExpressionLimit** (*mixed* \$expression, [*mixed* \$escapeChar], [*mixed* \$bindCounts]) inherited from *Phalcon\Db\Dialect*

Resolve a LIMIT clause

protected **prepareColumnAlias** (*mixed* \$qualified, [*mixed* \$alias], [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Prepares column for this RDBMS

protected **prepareTable** (*mixed* \$table, [*mixed* \$schema], [*mixed* \$alias], [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Prepares table for this RDBMS

protected **prepareQualified** (*mixed* \$column, [*mixed* \$domain], [*mixed* \$escapeChar]) inherited from *Phalcon\Db\Dialect*

Prepares qualified for this RDBMS

Class *Phalcon\Db\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class Phalcon\Db\Index

implements [Phalcon\Db\IndexInterface](#)

Allows to define indexes to be used on tables. Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index

```
<?php

// Define new unique index
$index_unique = new \Phalcon\Db\Index (
    'column_UNIQUE',
    [
        'column',
        'column'
    ],
    'UNIQUE'
);

// Define new primary index
$index_primary = new \Phalcon\Db\Index (
    'PRIMARY',
    [
        'column'
    ]
);

// Add index to existing table
$connection->addIndex("robots", null, $index_unique);
$connection->addIndex("robots", null, $index_primary);
```

Methods

public **getName** ()

Index name

public **getColumns** ()

Index columns

public **getType** ()

Index type

public **__construct** (*mixed* \$name, *array* \$columns, [*mixed* \$type])

Phalcon\Db\Index constructor

public static **__set_state** (array \$data)

Restore a Phalcon\Db\Index object from export

Class Phalcon\Db\Profiler

Instances of Phalcon\Db can generate execution profiles on SQL statements sent to the relational database. Profiled information includes execution time in milliseconds. This helps you to identify bottlenecks in your applications.

```
<?php

$profiler = new \Phalcon\Db\Profiler();

// Set the connection profiler
$connection->setProfiler($profiler);

$sql = "SELECT buyer_name, quantity, product_name
FROM buyers LEFT JOIN products ON
buyers.pid=products.id";

// Execute a SQL statement
$connection->query($sql);

// Get the last profile in the profiler
$profile = $profiler->getLastProfile();

echo "SQL Statement: ", $profile->getSQLStatement(), "\n";
echo "Start Time: ", $profile->getInitialTime(), "\n";
echo "Final Time: ", $profile->getFinalTime(), "\n";
echo "Total Elapsed Time: ", $profile->getTotalElapsedSeconds(), "\n";
```

Methods

public *Phalcon\Db\Profiler* **startProfile** (string \$sqlStatement, [mixed \$sqlVariables], [mixed \$sqlBindTypes])

Starts the profile of a SQL sentence

public **stopProfile** ()

Stops the active profile

public **getNumberTotalStatements** ()

Returns the total number of SQL statements processed

public **getTotalElapsedSeconds** ()

Returns the total time in seconds spent by the profiles

public **getProfiles** ()

Returns all the processed profiles

public **reset** ()

Resets the profiler, cleaning up all the profiles

public **getLastProfile** ()

Returns the last profile executed in the profiler

Class Phalcon\Db\Profiler\Item

This class identifies each profile in a Phalcon\Db\Profiler

Methods

public **setSqlStatement** (*mixed* \$sqlStatement)

SQL statement related to the profile

public **getSqlStatement** ()

SQL statement related to the profile

public **setSqlVariables** (*array* \$sqlVariables)

SQL variables related to the profile

public **getSqlVariables** ()

SQL variables related to the profile

public **setSqlBindTypes** (*array* \$sqlBindTypes)

SQL bind types related to the profile

public **getSqlBindTypes** ()

SQL bind types related to the profile

public **setInitialTime** (*mixed* \$initialTime)

Timestamp when the profile started

public **getInitialTime** ()

Timestamp when the profile started

public **setFinalTime** (*mixed* \$finalTime)

Timestamp when the profile ended

public **getFinalTime** ()

Timestamp when the profile ended

public **getTotalElapsedSeconds** ()

Returns the total time in seconds spent by the profile

Class Phalcon\Db\RawValue

This class allows to insert/update raw data without quoting or formatting.

The next example shows how to use the MySQL now() function as a field value.

```
<?php
$subscriber = new Subscribers();

$subscriber->email      = "andres@phalconphp.com";
$subscriber->createdAt = new \Phalcon\Db\RawValue("now()");

$subscriber->save();
```

Methods

public **getValue** ()

Raw value without quoting or formatting

public **__toString** ()

Raw value without quoting or formatting

public **__construct** (*mixed* \$value)

Phalcon\Db\RawValue constructor

Class Phalcon\Db\Reference

implements *Phalcon\Db\ReferenceInterface*

Allows to define reference constraints on tables

```
<?php

$reference = new \Phalcon\Db\Reference (
    "field_fk",
    [
        "referencedSchema" => "invoicing",
        "referencedTable"  => "products",
        "columns"           => [
            "product_type",
            "product_code",
        ],
        "referencedColumns" => [
            "type",
            "code",
        ],
    ]
);
```

Methods

public **getName** ()

Constraint name

public **getSchemaName** ()

...

public **getReferencedSchema** ()

...

public **getReferencedTable** ()

Referenced Table

public **getColumns** ()

Local reference columns

public **getReferencedColumns** ()

Referenced Columns

public **getOnDelete** ()

ON DELETE

public **getOnUpdate** ()

ON UPDATE

public **__construct** (*mixed* \$name, *array* \$definition)

Phalcon\Db\Reference constructor

public static **__set_state** (*array* \$data)

Restore a Phalcon\Db\Reference object from export

Class Phalcon\Db\Result\Pdo

implements *Phalcon\Db\ResultInterface*

Encapsulates the resultset internals

```
<?php

$result = $connection->query("SELECT * FROM robots ORDER BY name");

$result->setFetchMode(
    \Phalcon\Db::FETCH_NUM
);

while ($robot = $result->fetchArray()) {
    print_r($robot);
}
```

Methods

public **__construct** (*Phalcon\Db\AdapterInterface* \$connection, *PDOStatement* \$result, [*string* \$sqlStatement], [*array* \$bindParams], [*array* \$bindTypes])

Phalcon\Db\Result\Pdo constructor

public **execute** ()

Allows to execute the statement again. Some database systems don't support scrollable cursors, So, as cursors are forward only, we need to execute the cursor again to fetch rows from the beginning

public **fetch** ([*mixed* \$fetchStyle], [*mixed* \$cursorOrientation], [*mixed* \$cursorOffset])

Fetches an array/object of strings that corresponds to the fetched row, or FALSE if there are no more rows. This method is affected by the active fetch flag set using Phalcon\Db\Result\Pdo::setFetchMode

```
<?php

$result = $connection->query("SELECT * FROM robots ORDER BY name");

$result->setFetchMode(
    \Phalcon\Db::FETCH_OBJ
);
```

```
while ($robot = $result->fetch()) {
    echo $robot->name;
}
```

public **fetchArray** ()

Returns an array of strings that corresponds to the fetched row, or FALSE if there are no more rows. This method is affected by the active fetch flag set using `Phalcon\Db\Result\Pdo::setFetchMode`

```
<?php

$result = $connection->query("SELECT * FROM robots ORDER BY name");

$result->setFetchMode(
    \Phalcon\Db::FETCH_NUM
);

while ($robot = $result->fetchArray()) {
    print_r($robot);
}
```

public **fetchAll** ([*mixed* \$fetchStyle], [*mixed* \$fetchArgument], [*mixed* \$ctorArgs])

Returns an array of arrays containing all the records in the result This method is affected by the active fetch flag set using `Phalcon\Db\Result\Pdo::setFetchMode`

```
<?php

$result = $connection->query(
    "SELECT * FROM robots ORDER BY name"
);

$robots = $result->fetchAll();
```

public **numRows** ()

Gets number of rows returned by a resultset

```
<?php

$result = $connection->query(
    "SELECT * FROM robots ORDER BY name"
);

echo "There are ", $result->numRows(), " rows in the resultset";
```

public **dataSeek** (*mixed* \$number)

Moves internal resultset cursor to another position letting us to fetch a certain row

```
<?php

$result = $connection->query(
    "SELECT * FROM robots ORDER BY name"
);

// Move to third row on result
$result->dataSeek(2);
```

```
// Fetch third row
$row = $result->fetch();
```

public **setFetchMode** (*mixed* \$fetchMode, [*mixed* \$colNoOrClassNameOrObject], [*mixed* \$ctorargs])

Changes the fetching mode affecting Phalcon\Db\Result\Pdo::fetch()

```
<?php

// Return array with integer indexes
$result->setFetchMode(
    \Phalcon\Db::FETCH_NUM
);

// Return associative array without integer indexes
$result->setFetchMode(
    \Phalcon\Db::FETCH_ASSOC
);

// Return associative array together with integer indexes
$result->setFetchMode(
    \Phalcon\Db::FETCH_BOTH
);

// Return an object
$result->setFetchMode(
    \Phalcon\Db::FETCH_OBJ
);
```

public **getInternalResult** ()

Gets the internal PDO result object

Class Phalcon\Debug

Provides debug capabilities to Phalcon applications

Methods

public **setUri** (*mixed* \$uri)

Change the base URI for static resources

public **setShowBackTrace** (*mixed* \$showBackTrace)

Sets if files the exception's backtrace must be showed

public **setShowFiles** (*mixed* \$showFiles)

Set if files part of the backtrace must be shown in the output

public **setShowFileFragment** (*mixed* \$showFileFragment)

Sets if files must be completely opened and showed in the output or just the fragment related to the exception

public **listen** ([*mixed* \$exceptions], [*mixed* \$lowSeverity])

Listen for uncaught exceptions and unsilent notices or warnings

public **listenExceptions** ()

Listen for uncaught exceptions

public **listenLowSeverity** ()

Listen for unsilent notices or warnings

public **halt** ()

Halts the request showing a backtrace

public **debugVar** (*mixed* \$varz, [*mixed* \$key])

Adds a variable to the debug output

public **clearVars** ()

Clears are variables added previously

protected **_escapeString** (*mixed* \$value)

Escapes a string with htmlentities

protected **_getArrayDump** (*array* \$argument, [*mixed* \$n])

Produces a recursive representation of an array

protected **_getVarDump** (*mixed* \$variable)

Produces an string representation of a variable

public **getMajorVersion** ()

Returns the major framework's version

public **getVersion** ()

Generates a link to the current version documentation

public **getCssSources** ()

Returns the css sources

public **getJsSources** ()

Returns the javascript sources

final protected **showTraceItem** (*mixed* \$n, *array* \$trace)

Shows a backtrace item

public **onUncaughtLowSeverity** (*mixed* \$severity, *mixed* \$message, *mixed* \$file, *mixed* \$line, *mixed* \$context)

Throws an exception when a notice or warning is raised

public **onUncaughtException** (*Exception* \$exception)

Handles uncaught exceptions

Class Phalcon\Debug\Dump

Dumps information about a variable(s)

```
<?php
$foo = 123;
```

```
echo (new \Phalcon\Debug\Dump())->variable($foo, "foo");
```

```
<?php

$foo = "string";
$bar = ["key" => "value"];
$baz = new stdClass();

echo (new \Phalcon\Debug\Dump())->variables($foo, $bar, $baz);
```

Methods

public **getDetailed** ()

...

public **setDetailed** (*mixed* \$detailed)

...

public **__construct** ([*array* \$styles], [*mixed* \$detailed])

Phalcon\Debug\Dump constructor

public **all** ()

Alias of variables() method

protected **getStyle** (*mixed* \$type)

Get style for type

public **setStyles** ([*array* \$styles])

Set styles for vars type

public **one** (*mixed* \$variable, [*mixed* \$name])

Alias of variable() method

protected **output** (*mixed* \$variable, [*mixed* \$name], [*mixed* \$tab])

Prepare an HTML string of information about a single variable.

public **variable** (*mixed* \$variable, [*mixed* \$name])

Returns an HTML string of information about a single variable.

```
<?php

echo (new \Phalcon\Debug\Dump())->variable($foo, "foo");
```

public **variables** ()

Returns an HTML string of debugging information about any number of variables, each wrapped in a “pre” tag.

```
<?php

$foo = "string";
$bar = ["key" => "value"];
$baz = new stdClass();
```

```
echo (new \Phalcon\Debug\Dump())->variables($foo, $bar, $baz);
```

public **toJson** (*mixed* \$variable)

Returns an JSON string of information about a single variable.

```
<?php

$foo = [
    "key" => "value",
];

echo (new \Phalcon\Debug\Dump())->toJson($foo);

$foo = new stdClass();
$foo->bar = "buz";

echo (new \Phalcon\Debug\Dump())->toJson($foo);
```

Class **Phalcon\Debug\Exception**

extends class *Phalcon\Exception*

implements **Throwable**

Methods

final private **Exception** **__clone** () inherited from **Exception**

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [**Exception** \$previous]) inherited from **Exception**

Exception constructor

public **__wakeup** () inherited from **Exception**

...

final public *string* **getMessage** () inherited from **Exception**

Gets the Exception message

final public *int* **getCode** () inherited from **Exception**

Gets the Exception code

final public *string* **getFile** () inherited from **Exception**

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from **Exception**

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from **Exception**

Gets the stack trace

final public **Exception** **getPrevious** () inherited from **Exception**

Returns previous Exception

final public **Exception** **getTraceAsString** () inherited from **Exception**

Gets the stack trace as a string

public *string* **__toString** () inherited from **Exception**

String representation of the exception

Class Phalcon\Di

implements *Phalcon\DiInterface*, *ArrayAccess*

Phalcon\Di is a component that implements Dependency Injection/Service Location of services and it's itself a container for them.

Since Phalcon is highly decoupled, Phalcon\Di is essential to integrate the different components of the framework. The developer can also use this component to inject dependencies and manage global instances of the different classes used in the application.

Basically, this component implements the *Inversion of Control* pattern. Applying this, the objects do not receive their dependencies using setters or constructors, but requesting a service dependency injector. This reduces the overall complexity, since there is only one way to get the required dependencies within a component.

Additionally, this pattern increases testability in the code, thus making it less prone to errors.

```
<?php

use Phalcon\Di;
use Phalcon\Http\Request;

$di = new Di();

// Using a string definition
$di->set("request", Request::class, true);

// Using an anonymous function
$di->setShared(
    "request",
    function () {
        return new Request();
    }
);

$request = $di->getRequest();
```

Methods

public **__construct** ()

Phalcon\Di constructor

public **setInternalEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager)

Sets the internal event manager

public **getInternalEventManager** ()

Returns the internal event manager

public **set** (*mixed* \$name, *mixed* \$definition, [*mixed* \$shared])

Registers a service in the services container

public **setShared** (*mixed* \$name, *mixed* \$definition)

Registers an “always shared” service in the services container

public **remove** (*mixed* \$name)

Removes a service in the services container It also removes any shared instance created for the service

public **attempt** (*mixed* \$name, *mixed* \$definition, [*mixed* \$shared])

Attempts to register a service in the services container Only is successful if a service hasn’t been registered previously with the same name

public **setRaw** (*mixed* \$name, *Phalcon\Di\ServiceInterface* \$rawDefinition)

Sets a service using a raw *Phalcon\Di\Service* definition

public **getRaw** (*mixed* \$name)

Returns a service definition without resolving

public **getService** (*mixed* \$name)

Returns a *Phalcon\Di\Service* instance

public **get** (*mixed* \$name, [*mixed* \$parameters])

Resolves the service based on its configuration

public *mixed* **getShared** (*string* \$name, [*array* \$parameters])

Resolves a service, the resolved service is stored in the DI, subsequent requests for this service will return the same instance

public **has** (*mixed* \$name)

Check whether the DI contains a service by a name

public **wasFreshInstance** ()

Check whether the last service obtained via *getShared* produced a fresh instance or an existing one

public **getServices** ()

Return the services registered in the DI

public **offsetExists** (*mixed* \$name)

Check if a service is registered using the array syntax

public **offsetSet** (*mixed* \$name, *mixed* \$definition)

Allows to register a shared service using the array syntax

```
<?php
$di["request"] = new \Phalcon\Http\Request();
```

public **offsetGet** (*mixed* \$name)

Allows to obtain a shared service using the array syntax

```
<?php
var_dump($di["request"]);
```

public **offsetUnset** (*mixed* \$name)

Removes a service from the services container using the array syntax

public **__call** (*mixed* \$method, [*mixed* \$arguments])

Magic method to get or set services using setters/getters

public static **setDefault** (*Phalcon\DiInterface* \$dependencyInjector)

Set a default dependency injection container to be obtained into static methods

public static **getDefault** ()

Return the latest DI created

public static **reset** ()

Resets the internal default DI

Class **Phalcon\Di\Exception**

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Di\FactoryDefault](#)

extends class [Phalcon\Di](#)

implements [ArrayAccess](#), [Phalcon\DiInterface](#)

This is a variant of the standard [Phalcon\Di](#). By default it automatically registers all the services provided by the framework. Thanks to this, the developer does not need to register each service individually providing a full stack framework

Methods

public **__construct** ()

[Phalcon\Di\FactoryDefault](#) constructor

public **setInternalEventManager** ([Phalcon\Events\ManagerInterface](#) \$eventsManager) inherited from [Phalcon\Di](#)

Sets the internal event manager

public **getInternalEventManager** () inherited from [Phalcon\Di](#)

Returns the internal event manager

public **set** (*mixed* \$name, *mixed* \$definition, [*mixed* \$shared]) inherited from [Phalcon\Di](#)

Registers a service in the services container

public **setShared** (*mixed* \$name, *mixed* \$definition) inherited from [Phalcon\Di](#)

Registers an “always shared” service in the services container

public **remove** (*mixed* \$name) inherited from [Phalcon\Di](#)

Removes a service in the services container It also removes any shared instance created for the service

public **attempt** (*mixed* \$name, *mixed* \$definition, [*mixed* \$shared]) inherited from [Phalcon\Di](#)

Attempts to register a service in the services container Only is successful if a service hasn’t been registered previously with the same name

public **setRaw** (*mixed* \$name, [Phalcon\Di\ServiceInterface](#) \$rawDefinition) inherited from [Phalcon\Di](#)

Sets a service using a raw [Phalcon\Di\Service](#) definition

public **getRaw** (*mixed* \$name) inherited from [Phalcon\Di](#)

Returns a service definition without resolving

public **getService** (*mixed* \$name) inherited from [Phalcon\Di](#)

Returns a [Phalcon\Di\Service](#) instance

public **get** (*mixed* \$name, [*mixed* \$parameters]) inherited from [Phalcon\Di](#)

Resolves the service based on its configuration

public *mixed* **getShared** (*string* \$name, [*array* \$parameters]) inherited from [Phalcon\Di](#)

Resolves a service, the resolved service is stored in the DI, subsequent requests for this service will return the same instance

public **has** (*mixed* \$name) inherited from *Phalcon\Di*

Check whether the DI contains a service by a name

public **wasFreshInstance** () inherited from *Phalcon\Di*

Check whether the last service obtained via `getShared` produced a fresh instance or an existing one

public **getServices** () inherited from *Phalcon\Di*

Return the services registered in the DI

public **offsetExists** (*mixed* \$name) inherited from *Phalcon\Di*

Check if a service is registered using the array syntax

public **offsetSet** (*mixed* \$name, *mixed* \$definition) inherited from *Phalcon\Di*

Allows to register a shared service using the array syntax

```
<?php
$di["request"] = new \Phalcon\Http\Request();
```

public **offsetGet** (*mixed* \$name) inherited from *Phalcon\Di*

Allows to obtain a shared service using the array syntax

```
<?php
var_dump($di["request"]);
```

public **offsetUnset** (*mixed* \$name) inherited from *Phalcon\Di*

Removes a service from the services container using the array syntax

public **__call** (*mixed* \$method, [*mixed* \$arguments]) inherited from *Phalcon\Di*

Magic method to get or set services using setters/getters

public static **setDefault** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di*

Set a default dependency injection container to be obtained into static methods

public static **getDefault** () inherited from *Phalcon\Di*

Return the latest DI created

public static **reset** () inherited from *Phalcon\Di*

Resets the internal default DI

Class *Phalcon\Di\FactoryDefault\Cli*

extends class *Phalcon\Di\FactoryDefault*

implements *Phalcon\DiInterface*, *ArrayAccess*

This is a variant of the standard *Phalcon\Di*. By default it automatically registers all the services provided by the framework. Thanks to this, the developer does not need to register each service individually. This class is specially suitable for CLI applications

Methods

public **__construct** ()

Phalcon\Di\FactoryDefault\Cli constructor

public **setInternalEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Di*

Sets the internal event manager

public **getInternalEventManager** () inherited from *Phalcon\Di*

Returns the internal event manager

public **set** (*mixed* \$name, *mixed* \$definition, [*mixed* \$shared]) inherited from *Phalcon\Di*

Registers a service in the services container

public **setShared** (*mixed* \$name, *mixed* \$definition) inherited from *Phalcon\Di*

Registers an “always shared” service in the services container

public **remove** (*mixed* \$name) inherited from *Phalcon\Di*

Removes a service in the services container It also removes any shared instance created for the service

public **attempt** (*mixed* \$name, *mixed* \$definition, [*mixed* \$shared]) inherited from *Phalcon\Di*

Attempts to register a service in the services container Only is successful if a service hasn’t been registered previously with the same name

public **setRaw** (*mixed* \$name, *Phalcon\Di\ServiceInterface* \$rawDefinition) inherited from *Phalcon\Di*

Sets a service using a raw Phalcon\Di\Service definition

public **getRaw** (*mixed* \$name) inherited from *Phalcon\Di*

Returns a service definition without resolving

public **getService** (*mixed* \$name) inherited from *Phalcon\Di*

Returns a Phalcon\Di\Service instance

public **get** (*mixed* \$name, [*mixed* \$parameters]) inherited from *Phalcon\Di*

Resolves the service based on its configuration

public *mixed* **getShared** (*string* \$name, [*array* \$parameters]) inherited from *Phalcon\Di*

Resolves a service, the resolved service is stored in the DI, subsequent requests for this service will return the same instance

public **has** (*mixed* \$name) inherited from *Phalcon\Di*

Check whether the DI contains a service by a name

public **wasFreshInstance** () inherited from *Phalcon\Di*

Check whether the last service obtained via getShared produced a fresh instance or an existing one

public **getServices** () inherited from *Phalcon\Di*

Return the services registered in the DI

public **offsetExists** (*mixed* \$name) inherited from *Phalcon\Di*

Check if a service is registered using the array syntax

public **offsetSet** (*mixed* \$name, *mixed* \$definition) inherited from *Phalcon\Di*

Allows to register a shared service using the array syntax

```
<?php
$di["request"] = new \Phalcon\Http\Request();
```

public **offsetGet** (*mixed* \$name) inherited from *Phalcon\Di*

Allows to obtain a shared service using the array syntax

```
<?php
var_dump($di["request"]);
```

public **offsetUnset** (*mixed* \$name) inherited from *Phalcon\Di*

Removes a service from the services container using the array syntax

public **__call** (*mixed* \$method, [*mixed* \$arguments]) inherited from *Phalcon\Di*

Magic method to get or set services using setters/getters

public static **setDefault** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di*

Set a default dependency injection container to be obtained into static methods

public static **getDefault** () inherited from *Phalcon\Di*

Return the latest DI created

public static **reset** () inherited from *Phalcon\Di*

Resets the internal default DI

Abstract class *Phalcon\Di\Injectable*

implements Phalcon\Di\InjectionAwareInterface, Phalcon\Events\EventsAwareInterface

This class allows to access services in the services container by just only accessing a public property with the same name of a registered service

Methods

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injector

public **getDI** ()

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager)

Sets the event manager

public **getEventManager** ()

Returns the internal event manager

public **__get** (*mixed* \$propertyName)

Magic method **__get**

Class Phalcon\Di\Service

implements Phalcon\Di\ServiceInterface

Represents individually a service in the services container

```
<?php

$service = new \Phalcon\Di\Service(
    "request",
    "Phalcon\Http\Request"
);

$request = service->resolve();
```

```
<?php
```

Methods

final public **__construct** (*string* \$name, *mixed* \$definition, [*boolean* \$shared])

public **getName** ()

Returns the service's name

public **setShared** (*mixed* \$shared)

Sets if the service is shared or not

public **isShared** ()

Check whether the service is shared or not

public **setSharedInstance** (*mixed* \$sharedInstance)

Sets/Resets the shared instance related to the service

public **setDefinition** (*mixed* \$definition)

Set the service definition

public *mixed* **getDefinition** ()

Returns the service definition

public *mixed* **resolve** ([*array* \$parameters], [*Phalcon\DiInterface* \$dependencyInjector])

Resolves the service

public **setParameter** (*mixed* \$position, *array* \$parameter)

Changes a parameter in the definition without resolve the service

public *array* **getParameter** (*int* \$position)

Returns a parameter in a specific position

public **isResolved** ()

Returns true if the service was resolved

public static **__set_state** (*array* \$attributes)

Restore the internal state of a service

Class Phalcon\Di\Service\Builder

This class builds instances based on complex definitions

Methods

private *mixed* **_buildParameter** (*Phalcon\DiInterface* \$dependencyInjector, *int* \$position, *array* \$argument)

Resolves a constructor/call parameter

private **_buildParameters** (*Phalcon\DiInterface* \$dependencyInjector, *array* \$arguments)

Resolves an array of parameters

public *mixed* **build** (*Phalcon\DiInterface* \$dependencyInjector, *array* \$definition, [*array* \$parameters])

Builds a service using a complex service definition

Abstract class Phalcon\Dispatcher

implements *Phalcon\DispatcherInterface*, *Phalcon\Di\InjectionAwareInterface*, *Phalcon\Events\EventsAwareInterface*

This is the base class for Phalcon\Mvc\Dispatcher and Phalcon\Cli\Dispatcher. This class can't be instantiated directly, you can use it to create your own dispatchers.

Constants

integer **EXCEPTION_NO_DI**

integer **EXCEPTION_CYCLIC_ROUTING**

integer **EXCEPTION_HANDLER_NOT_FOUND**

integer **EXCEPTION_INVALID_HANDLER**

integer **EXCEPTION_INVALID_PARAMS**

integer **EXCEPTION_ACTION_NOT_FOUND**

Methods

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injector

public **getDI** ()

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager)

Sets the events manager

public **getEventManager** ()

Returns the internal event manager

public **setActionSuffix** (*mixed* \$actionSuffix)

Sets the default action suffix

public **getActionSuffix** ()

Gets the default action suffix

public **setModuleName** (*mixed* \$moduleName)

Sets the module where the controller is (only informative)

public **getModuleName** ()

Gets the module where the controller class is

public **setNamespaceName** (*mixed* \$namespaceName)

Sets the namespace where the controller class is

public **getNamespaceName** ()

Gets a namespace to be prepended to the current handler name

public **setDefaultNamespace** (*mixed* \$namespaceName)

Sets the default namespace

public **getDefaultNamespace** ()

Returns the default namespace

public **setDefaultAction** (*mixed* \$actionName)

Sets the default action name

public **setActionName** (*mixed* \$actionName)

Sets the action name to be dispatched

public **getActionName** ()

Gets the latest dispatched action name

public **setParams** (*array* \$params)

Sets action params to be dispatched

public **getParams** ()

Gets action params

public **setParam** (*mixed* \$param, *mixed* \$value)

Set a param by its name or numeric index

public *mixed* **getParam** (*mixed* \$param, [*string* | *array* \$filters], [*mixed* \$defaultValue])

Gets a param by its name or numeric index

public *boolean* **hasParam** (*mixed* \$param)

Check if a param exists

public **getActiveMethod** ()

Returns the current method to be/executed in the dispatcher

public **isFinished** ()

Checks if the dispatch loop is finished or has more pendent controllers/tasks to dispatch

public **setReturnedValue** (*mixed* \$value)

Sets the latest returned value by an action manually

public *mixed* **getReturnedValue** ()

Returns value returned by the latest dispatched action

public **setModelBinding** (*mixed* \$value, [*mixed* \$cache])

Enable/Disable model binding during dispatch

```
<?php

$di->set('dispatcher', function() {
    $dispatcher = new Dispatcher();

    $dispatcher->setModelBinding(true, 'cache');
    return $dispatcher;
});
```

public **setModelBinder** (Phalcon\Mvc\Model\BinderInterface \$modelBinder, [*mixed* \$cache])

Enable model binding during dispatch

```
<?php

$di->set('dispatcher', function() {
    $dispatcher = new Dispatcher();

    $dispatcher->setModelBinder(new Binder(), 'cache');
    return $dispatcher;
});
```

public **getModelBinder** ()

Gets model binder

public *object* **dispatch** ()

Dispatches a handle action taking into account the routing parameters

protected *object* **_dispatch** ()

Dispatches a handle action taking into account the routing parameters

public **forward** (array \$forward)

Forwards the execution flow to another controller/action Dispatchers are unique per module. Forwarding between modules is not allowed

```
<?php

$this->dispatcher->forward(
    [
        "controller" => "posts",
        "action"      => "index",
    ]
);
```

public **wasForwarded** ()

Check if the current executed action was forwarded by another one

public **getHandlerClass** ()

Possible class name that will be located to dispatch the request

public **callActionMethod** (*mixed* \$handler, *mixed* \$actionMethod, [array \$params])

...

public **getBoundModels** ()

Returns bound models from binder instance

```
<?php

class UserController extends Controller
{
    public function showAction(User $user)
    {
        $boundModels = $this->dispatcher->getBoundModels(); // return array with $user
    }
}
```

protected **_resolveEmptyProperties** ()

Set empty properties to their defaults (where defaults are available)

Class Phalcon\Escaper

implements Phalcon\EscaperInterface

Escapes different kinds of text securing them. By using this component you may prevent XSS attacks.

This component only works with UTF-8. The PREG extension needs to be compiled with UTF-8 support.

```
<?php

$escaper = new \Phalcon\Escaper();

$escaped = $escaper->escapeCss("font-family: <Verdana>");

echo $escaped; // font\2D family\3A \20 \3C Verdana\3E
```

Methods

public **setEncoding** (*mixed* \$encoding)

Sets the encoding to be used by the escaper

```
<?php

$escaper->setEncoding("utf-8");
```

public **getEncoding** ()

Returns the internal encoding used by the escaper

public **setHtmlQuoteType** (*mixed* \$quoteType)

Sets the HTML quoting type for htmlspecialchars

```
<?php
$escaper->setHtmlQuoteType (ENT_XHTML) ;
```

public **setDoubleEncode** (*mixed* \$doubleEncode)

Sets the double_encode to be used by the escaper

```
<?php
$escaper->setDoubleEncode (false) ;
```

final public **detectEncoding** (*mixed* \$str)

Detect the character encoding of a string to be handled by an encoder Special-handling for chr(172) and chr(128) to chr(159) which fail to be detected by mb_detect_encoding()

final public **normalizeEncoding** (*mixed* \$str)

Utility to normalize a string's encoding to UTF-32.

public **escapeHtml** (*mixed* \$text)

Escapes a HTML string. Internally uses htmlspecialchars

public **escapeHtmlAttr** (*mixed* \$attribute)

Escapes a HTML attribute string

public **escapeCss** (*mixed* \$css)

Escape CSS strings by replacing non-alphanumeric chars by their hexadecimal escaped representation

public **escapeJs** (*mixed* \$js)

Escape javascript strings by replacing non-alphanumeric chars by their hexadecimal escaped representation

public **escapeUrl** (*mixed* \$url)

Escapes a URL. Internally uses rawurlencode

Class **Phalcon\Escaper\Exception**

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from [Exception](#)

Gets the Exception code

final public *string* **getFile** () inherited from [Exception](#)

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from [Exception](#)

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Events\Event](#)

implements [Phalcon\Events\EventInterface](#)

This class offers contextual information of a fired event in the EventsManager

Methods

public **getType** ()

Event type

public **getSource** ()

Event source

public **getData** ()

Event data

public **__construct** (*string* \$type, *object* \$source, [*mixed* \$data], [*boolean* \$cancelable])

[Phalcon\Events\Event](#) constructor

public **setData** ([*mixed* \$data])

Sets event data.

public **setType** (*mixed* \$type)

Sets event type.

public **stop** ()

Stops the event preventing propagation.

```
<?php

if ($event->isCancelable()) {
    $event->stop();
}
```

public **isStopped** ()

Check whether the event is currently stopped.

public **isCancelable** ()

Check whether the event is cancelable.

```
<?php

if ($event->isCancelable()) {
    $event->stop();
}
```

Class `Phalcon\Events\Exception`

extends class `Phalcon\Exception`

implements `Throwable`

Methods

final private `Exception` **__clone** () inherited from `Exception`

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [`Exception` \$previous]) inherited from `Exception`

Exception constructor

public **__wakeup** () inherited from `Exception`

...

final public *string* **getMessage** () inherited from `Exception`

Gets the Exception message

final public *int* **getCode** () inherited from `Exception`

Gets the Exception code

final public *string* **getFile** () inherited from `Exception`

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from `Exception`

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from `Exception`

Gets the stack trace

final public `Exception` **getPrevious** () inherited from `Exception`

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Events\Manager](#)

implements [Phalcon\Events\ManagerInterface](#)

Phalcon Events Manager, offers an easy way to intercept and manipulate, if needed, the normal flow of operation. With the EventsManager the developer can create hooks or plugins that will offer monitoring of data, manipulation, conditional execution and much more.

Methods

public **attach** (*string* \$eventType, *object* | *callable* \$handler, [*int* \$priority])

Attach a listener to the events manager

public **detach** (*string* \$eventType, *object* \$handler)

Detach the listener from the events manager

public **enablePriorities** (*mixed* \$enablePriorities)

Set if priorities are enabled in the EventsManager

public **arePrioritiesEnabled** ()

Returns if priorities are enabled

public **collectResponses** (*mixed* \$collect)

Tells the event manager if it needs to collect all the responses returned by every registered listener in a single fire

public **isCollecting** ()

Check if the events manager is collecting all all the responses returned by every registered listener in a single fire

public *array* **getResponses** ()

Returns all the responses returned by every handler executed by the last 'fire' executed

public **detachAll** ([*mixed* \$type])

Removes all events from the EventsManager

final public *mixed* **fireQueue** ([SplPriorityQueue](#) | *array* \$queue, [Phalcon\Events\Event](#) \$event)

Internal handler to call a queue of events

public *mixed* **fire** (*string* \$eventType, *object* \$source, [*mixed* \$data], [*boolean* \$cancelable])

Fires an event in the events manager causing the active listeners to be notified about it

```
<?php
$eventsManager->fire("db", $connection);
```

public **hasListeners** (*mixed* \$type)

Check whether certain type of event has listeners

public *array* **getListeners** (*string* \$type)

Returns all the attached listeners of a certain type

Class Phalcon\Exception

extends class [Exception](#)

implements [Throwable](#)

Methods

final private [Exception](#) **__clone** () inherited from [Exception](#)

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [[Exception](#) \$previous]) inherited from [Exception](#)

Exception constructor

public **__wakeup** () inherited from [Exception](#)

...

final public *string* **getMessage** () inherited from [Exception](#)

Gets the Exception message

final public *int* **getCode** () inherited from [Exception](#)

Gets the Exception code

final public *string* **getFile** () inherited from [Exception](#)

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from [Exception](#)

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class Phalcon\Filter

implements *Phalcon\FilterInterface*

The Phalcon\Filter component provides a set of commonly needed data filters. It provides object oriented wrappers to the php filter extension. Also allows the developer to define his/her own filters

```
<?php

$filter = new \Phalcon\Filter();

$filter->sanitize("some(one)@exa\mple.com", "email"); // returns "someone@example.com"
↳
$filter->sanitize("hello<<", "string"); // returns "hello"
$filter->sanitize("!100a019", "int"); // returns "100019"
$filter->sanitize("!100a019.01a", "float"); // returns "100019.01"
```

Constants

string **FILTER_EMAIL**

string **FILTER_ABSINT**

string **FILTER_INT**

string **FILTER_INT_CAST**

string **FILTER_STRING**

string **FILTER_FLOAT**

string **FILTER_FLOAT_CAST**

string **FILTER_ALPHANUM**

string **FILTER_TRIM**

string **FILTER_STRIPTAGS**

string **FILTER_LOWER**

string **FILTER_UPPER**

Methods

public **add** (*mixed* \$name, *mixed* \$handler)

Adds a user-defined filter

public **sanitize** (*mixed* \$value, *mixed* \$filters, [*mixed* \$noRecursive])

Sanitizes a value with a specified single or set of filters

protected **_sanitize** (*mixed* \$value, *mixed* \$filter)

Internal sanitize wrapper to filter_var

public **getFilters** ()

Return the user-defined filters in the instance

Class Phalcon\Filter\Exception

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Abstract class Phalcon\Flash

implements *Phalcon\FlashInterface*, *Phalcon\DI\InjectionAwareInterface*

Shows HTML notifications related to different circumstances. Classes can be stylized using CSS

```
<?php
$flash->success("The record was successfully deleted");
$flash->error("Cannot open the file");
```

Methods

public **__construct** ([*mixed* \$cssClasses])

Phalcon\Flash constructor

public **getAutoescape** ()

Returns the autoescape mode in generated html

public **setAutoescape** (*mixed* \$autoescape)

Set the autoescape mode in generated html

public **getEscaperService** ()

Returns the Escaper Service

public **setEscaperService** (*Phalcon\EscaperInterface* \$escaperService)

Sets the Escaper Service

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injector

public **getDI** ()

Returns the internal dependency injector

public **setImplicitFlush** (*mixed* \$implicitFlush)

Set whether the output must be implicitly flushed to the output or returned as string

public **setAutomaticHtml** (*mixed* \$automaticHtml)

Set if the output must be implicitly formatted with HTML

public **setCssClasses** (*array* \$cssClasses)

Set an array with CSS classes to format the messages

public **error** (*mixed* \$message)

Shows a HTML error message

```
<?php
$flash->error("This is an error");
```

public **notice** (*mixed* \$message)

Shows a HTML notice/information message

```
<?php
$flash->notice("This is an information");
```

public **success** (*mixed* \$message)

Shows a HTML success message

```
<?php
$flash->success("The process was finished successfully");
```

public **warning** (*mixed* \$message)

Shows a HTML warning message

```
<?php
$flash->warning("Hey, this is important");
```

public *string* | void **outputMessage** (*mixed* \$type, *string* | *array* \$message)

Outputs a message formatting it with HTML

```
<?php
$flash->outputMessage("error", $message);
```

public **clear** ()

Clears accumulated messages when implicit flush is disabled

abstract public **message** (*mixed* \$type, *mixed* \$message) inherited from *Phalcon\FlashInterface*

...

Class *Phalcon\Flash\Direct*

extends abstract class *Phalcon\Flash*

implements *Phalcon\Di\InjectionAwareInterface*, *Phalcon\FlashInterface*

This is a variant of the *Phalcon\Flash* that immediately outputs any message passed to it

Methods

public **message** (*mixed* \$type, *mixed* \$message)

Outputs a message

public **output** ([*mixed* \$remove])

Prints the messages accumulated in the flasher

public **__construct** ([*mixed* \$cssClasses]) inherited from *Phalcon\Flash*

Phalcon\Flash constructor

public **getAutoescape** () inherited from *Phalcon\Flash*

Returns the autoescape mode in generated html

public **setAutoescape** (*mixed* \$autoescape) inherited from *Phalcon\Flash*

Set the autoescape mode in generated html

public **getEscaperService** () inherited from *Phalcon\Flash*

Returns the Escaper Service

public **setEscaperService** (*Phalcon\EscaperInterface* \$escaperService) inherited from *Phalcon\Flash*

Sets the Escaper Service

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Flash*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Flash*

Returns the internal dependency injector

public **setImplicitFlush** (*mixed* \$implicitFlush) inherited from *Phalcon\Flash*

Set whether the output must be implicitly flushed to the output or returned as string

public **setAutomaticHtml** (*mixed* \$automaticHtml) inherited from *Phalcon\Flash*

Set if the output must be implicitly formatted with HTML

public **setCssClasses** (*array* \$cssClasses) inherited from *Phalcon\Flash*

Set an array with CSS classes to format the messages

public **error** (*mixed* \$message) inherited from *Phalcon\Flash*

Shows a HTML error message

```
<?php
$flash->error("This is an error");
```

public **notice** (*mixed* \$message) inherited from *Phalcon\Flash*

Shows a HTML notice/information message

```
<?php
$flash->notice("This is an information");
```

public **success** (*mixed* \$message) inherited from *Phalcon\Flash*

Shows a HTML success message

```
<?php
$flash->success("The process was finished successfully");
```

public **warning** (*mixed* \$message) inherited from *Phalcon\Flash*

Shows a HTML warning message

```
<?php
$flash->warning("Hey, this is important");
```

public *string* | *void* **outputMessage** (*mixed* \$type, *string* | *array* \$message) inherited from *Phalcon\Flash*

Outputs a message formatting it with HTML

```
<?php
$flash->outputMessage("error", $message);
```

public **clear** () inherited from *Phalcon\Flash*

Clears accumulated messages when implicit flush is disabled

Class Phalcon\Flash\Exception

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Class Phalcon\Flash\Session

extends abstract class *Phalcon\Flash*

implements *Phalcon\DI\InjectionAwareInterface*, *Phalcon\FlashInterface*

Temporarily stores the messages in session, then messages can be printed in the next request

Methods

protected **_getSessionMessages** (*mixed* \$remove, [*mixed* \$type])

Returns the messages stored in session

protected **_setSessionMessages** (*array* \$messages)

Stores the messages in session

public **message** (*mixed* \$type, *mixed* \$message)

Adds a message to the session flasher

public **has** ([*mixed* \$type])

Checks whether there are messages

public **getMessages** ([*mixed* \$type], [*mixed* \$remove])

Returns the messages in the session flasher

public **output** ([*mixed* \$remove])

Prints the messages in the session flasher

public **clear** ()

Clear messages in the session messenger

public **__construct** ([*mixed* \$cssClasses]) inherited from *Phalcon\Flash*

Phalcon\Flash constructor

public **getAutoescape** () inherited from *Phalcon\Flash*

Returns the autoescape mode in generated html

public **setAutoescape** (*mixed* \$autoescape) inherited from *Phalcon\Flash*

Set the autoescape mode in generated html

public **getEscaperService** () inherited from *Phalcon\Flash*

Returns the Escaper Service

public **setEscaperService** (*Phalcon\EscaperInterface* \$escaperService) inherited from *Phalcon\Flash*

Sets the Escaper Service

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Flash*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Flash*

Returns the internal dependency injector

public **setImplicitFlush** (*mixed* \$implicitFlush) inherited from *Phalcon\Flash*

Set whether the output must be implicitly flushed to the output or returned as string

public **setAutomaticHtml** (*mixed* \$automaticHtml) inherited from *Phalcon\Flash*

Set if the output must be implicitly formatted with HTML

public **setCssClasses** (*array* \$cssClasses) inherited from *Phalcon\Flash*

Set an array with CSS classes to format the messages

public **error** (*mixed* \$message) inherited from *Phalcon\Flash*

Shows a HTML error message

```
<?php
$flash->error("This is an error");
```

public **notice** (*mixed* \$message) inherited from *Phalcon\Flash*

Shows a HTML notice/information message

```
<?php
$flash->notice("This is an information");
```

public **success** (*mixed* \$message) inherited from *Phalcon\Flash*

Shows a HTML success message

```
<?php
$flash->success("The process was finished successfully");
```

public **warning** (*mixed* \$message) inherited from *Phalcon\Flash*

Shows a HTML warning message

```
<?php
$flash->warning("Hey, this is important");
```

public *string* | void **outputMessage** (*mixed* \$type, *string* | *array* \$message) inherited from *Phalcon\Flash*

Outputs a message formatting it with HTML

```
<?php
$flash->outputMessage("error", $message);
```

Abstract class **Phalcon\Forms\Element**

implements *Phalcon\Forms\ElementInterface*

This is a base class for form elements

Methods

public **__construct** (*string* \$name, [*array* \$attributes])

Phalcon\Forms\Element constructor

public **setForm** (*Phalcon\Forms\Form* \$form)

Sets the parent form to the element

public **getForm** ()

Returns the parent form to the element

public **setName** (*mixed* \$name)

Sets the element name

```
public getName ()
```

Returns the element name

```
public Phalcon\Forms\ElementInterface setFilters (array | string $filters)
```

Sets the element filters

```
public addFilter (mixed $filter)
```

Adds a filter to current list of filters

```
public mixed getFilters ()
```

Returns the element filters

```
public Phalcon\Forms\ElementInterface addValidators (array $validators, [mixed $merge])
```

Adds a group of validators

```
public addValidator (Phalcon\Validation\ValidatorInterface $validator)
```

Adds a validator to the element

```
public getValidators ()
```

Returns the validators registered for the element

```
public prepareAttributes ([array $attributes], [mixed $useChecked])
```

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

```
public Phalcon\Forms\ElementInterface setAttribute (string $attribute, mixed $value)
```

Sets a default attribute for the element

```
public mixed getAttribute (string $attribute, [mixed $defaultValue])
```

Returns the value of an attribute if present

```
public setAttributes (array $attributes)
```

Sets default attributes for the element

```
public getAttributes ()
```

Returns the default attributes for the element

```
public Phalcon\Forms\ElementInterface setUserOption (string $option, mixed $value)
```

Sets an option for the element

```
public mixed getUserOption (string $option, [mixed $defaultValue])
```

Returns the value of an option if present

```
public setUserOptions (array $options)
```

Sets options for the element

```
public getUserOptions ()
```

Returns the options for the element

```
public setLabel (mixed $label)
```

Sets the element label

```
public getLabel ()
```

Returns the element label

public **label** ([array \$attributes])

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (mixed \$value)

Sets a default value in case the form does not use an entity or there is no value available for the element in `_POST`

public **getDefault** ()

Returns the default value assigned to the element

public **getValue** ()

Returns the element value

public **getMessages** ()

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** ()

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group)

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message)

Appends a message to the internal message list

public **clear** ()

Clears every element in the form to its default value

public **__toString** ()

Magic method `__toString` renders the widget without attributes

abstract public **render** ([mixed \$attributes]) inherited from *Phalcon\Forms\ElementInterface*

...

Class *Phalcon\Forms\Element\Check*

extends abstract class *Phalcon\Forms\Element*

implements *Phalcon\Forms\ElementInterface*

Component `INPUT[type=check]` for forms

Methods

public **render** ([array \$attributes])

Renders the element widget returning html

public **__construct** (string \$name, [array \$attributes]) inherited from *Phalcon\Forms\Element*

Phalcon\Forms\Element constructor

public **setForm** (*Phalcon\Forms\Form* \$form) inherited from *Phalcon\Forms\Element*

Sets the parent form to the element

public **getForm** () inherited from *Phalcon\Forms\Element*

Returns the parent form to the element

public **setName** (*mixed* \$name) inherited from *Phalcon\Forms\Element*

Sets the element name

public **getName** () inherited from *Phalcon\Forms\Element*

Returns the element name

public *Phalcon\Forms\ElementInterface* **setFilters** (*array* | *string* \$filters) inherited from *Phalcon\Forms\Element*

Sets the element filters

public **addFilter** (*mixed* \$filter) inherited from *Phalcon\Forms\Element*

Adds a filter to current list of filters

public *mixed* **getFilters** () inherited from *Phalcon\Forms\Element*

Returns the element filters

public *Phalcon\Forms\ElementInterface* **addValidators** (*array* \$validators, [*mixed* \$merge]) inherited from *Phalcon\Forms\Element*

Adds a group of validators

public **addValidator** (*Phalcon\Validation\ValidatorInterface* \$validator) inherited from *Phalcon\Forms\Element*

Adds a validator to the element

public **getValidators** () inherited from *Phalcon\Forms\Element*

Returns the validators registered for the element

public **prepareAttributes** ([*array* \$attributes], [*mixed* \$useChecked]) inherited from *Phalcon\Forms\Element*

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* \$attribute, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default attribute for the element

public *mixed* **getAttribute** (*string* \$attribute, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (*array* \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (*string* \$option, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public *mixed* **getUserOption** (*string* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (*array* \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (*mixed* \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([*array* \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (*mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in `_POST`

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method `__toString` renders the widget without attributes

Class *Phalcon\Forms\Element\Date*

extends abstract class *Phalcon\Forms\Element*

implements *Phalcon\Forms\ElementInterface*

Component `INPUT[type=date]` for forms

Methods

public **render** ([*array* \$attributes])

Renders the element widget returning html

public **__construct** (*string* \$name, [*array* \$attributes]) inherited from *Phalcon\Forms\Element*

Phalcon\Forms\Element constructor

public **setForm** (*Phalcon\Forms\Form* \$form) inherited from *Phalcon\Forms\Element*

Sets the parent form to the element

public **getForm** () inherited from *Phalcon\Forms\Element*

Returns the parent form to the element

public **setName** (*mixed* \$name) inherited from *Phalcon\Forms\Element*

Sets the element name

public **getName** () inherited from *Phalcon\Forms\Element*

Returns the element name

public *Phalcon\Forms\ElementInterface* **setFilters** (*array* | *string* \$filters) inherited from *Phalcon\Forms\Element*

Sets the element filters

public **addFilter** (*mixed* \$filter) inherited from *Phalcon\Forms\Element*

Adds a filter to current list of filters

public *mixed* **getFilters** () inherited from *Phalcon\Forms\Element*

Returns the element filters

public *Phalcon\Forms\ElementInterface* **addValidators** (*array* \$validators, [*mixed* \$merge]) inherited from *Phalcon\Forms\Element*

Adds a group of validators

public **addValidator** (*Phalcon\Validation\ValidatorInterface* \$validator) inherited from *Phalcon\Forms\Element*

Adds a validator to the element

public **getValidators** () inherited from *Phalcon\Forms\Element*

Returns the validators registered for the element

public **prepareAttributes** ([*array* \$attributes], [*mixed* \$useChecked]) inherited from *Phalcon\Forms\Element*

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* \$attribute, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default attribute for the element

public *mixed* **getAttribute** (*string* \$attribute, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (*array* \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (*string* \$option, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public *mixed* **getUserOption** (*string* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (array \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (mixed \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([array \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (mixed \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in `_POST`

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method `__toString` renders the widget without attributes

Class *Phalcon\Forms\Element\Email*

extends abstract class *Phalcon\Forms\Element*

implements *Phalcon\Forms\ElementInterface*

Component `INPUT[type=email]` for forms

Methods

public **render** ([array \$attributes])

Renders the element widget returning html

public **__construct** (string \$name, [array \$attributes]) inherited from *Phalcon\Forms\Element*

Phalcon\Forms\Element constructor

public **setForm** (*Phalcon\Forms\Form* \$form) inherited from *Phalcon\Forms\Element*

Sets the parent form to the element

public **getForm** () inherited from *Phalcon\Forms\Element*

Returns the parent form to the element

public **setName** (mixed \$name) inherited from *Phalcon\Forms\Element*

Sets the element name

public **getName** () inherited from *Phalcon\Forms\Element*

Returns the element name

public *Phalcon\Forms\ElementInterface* **setFilters** (array | string \$filters) inherited from *Phalcon\Forms\Element*

Sets the element filters

public **addFilter** (mixed \$filter) inherited from *Phalcon\Forms\Element*

Adds a filter to current list of filters

public mixed **getFilters** () inherited from *Phalcon\Forms\Element*

Returns the element filters

public *Phalcon\Forms\ElementInterface* **addValidators** (array \$validators, [mixed \$merge]) inherited from *Phalcon\Forms\Element*

Adds a group of validators

public **addValidator** (*Phalcon\Validation\ValidatorInterface* \$validator) inherited from *Phalcon\Forms\Element*

Adds a validator to the element

public **getValidators** () inherited from *Phalcon\Forms\Element*

Returns the validators registered for the element

public **prepareAttributes** ([array \$attributes], [mixed \$useChecked]) inherited from *Phalcon\Forms\Element*

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (string \$attribute, mixed \$value) inherited from *Phalcon\Forms\Element*

Sets a default attribute for the element

public mixed **getAttribute** (string \$attribute, [mixed \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (array \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (*string* \$option, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public *mixed* **getUserOption** (*string* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (*array* \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (*mixed* \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([*array* \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (*mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in `_POST`

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method `__toString` renders the widget without attributes

Class `Phalcon\Forms\Element\File`

extends abstract class `Phalcon\Forms\Element`

implements `Phalcon\Forms\ElementInterface`

Component INPUT[type=file] for forms

Methods

public **render** ([array \$attributes])

Renders the element widget returning html

public **__construct** (string \$name, [array \$attributes]) inherited from `Phalcon\Forms\Element`

Phalcon\Forms\Element constructor

public **setForm** (`Phalcon\Forms\Form` \$form) inherited from `Phalcon\Forms\Element`

Sets the parent form to the element

public **getForm** () inherited from `Phalcon\Forms\Element`

Returns the parent form to the element

public **setName** (mixed \$name) inherited from `Phalcon\Forms\Element`

Sets the element name

public **getName** () inherited from `Phalcon\Forms\Element`

Returns the element name

public `Phalcon\Forms\ElementInterface` **setFilters** (array | string \$filters) inherited from `Phalcon\Forms\Element`

Sets the element filters

public **addFilter** (mixed \$filter) inherited from `Phalcon\Forms\Element`

Adds a filter to current list of filters

public mixed **getFilters** () inherited from `Phalcon\Forms\Element`

Returns the element filters

public `Phalcon\Forms\ElementInterface` **addValidators** (array \$validators, [mixed \$merge]) inherited from `Phalcon\Forms\Element`

Adds a group of validators

public **addValidator** (`Phalcon\Validation\ValidatorInterface` \$validator) inherited from `Phalcon\Forms\Element`

Adds a validator to the element

public **getValidators** () inherited from `Phalcon\Forms\Element`

Returns the validators registered for the element

public **prepareAttributes** ([array \$attributes], [mixed \$useChecked]) inherited from `Phalcon\Forms\Element`

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public `Phalcon\Forms\ElementInterface` **setAttribute** (string \$attribute, mixed \$value) inherited from `Phalcon\Forms\Element`

Sets a default attribute for the element

public *mixed* **getAttribute** (*string* \$attribute, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (*array* \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (*string* \$option, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public *mixed* **getUserOption** (*string* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (*array* \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (*mixed* \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([*array* \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (*mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in `_POST`

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method **__toString** renders the widget without attributes

Class *Phalcon\Forms\Element\Hidden*

extends abstract class *Phalcon\Forms\Element*

implements *Phalcon\Forms\ElementInterface*

Component INPUT[type=hidden] for forms

Methods

public **render** ([array \$attributes])

Renders the element widget returning html

public **__construct** (string \$name, [array \$attributes]) inherited from *Phalcon\Forms\Element*

Phalcon\Forms\Element constructor

public **setForm** (*Phalcon\Forms\Form* \$form) inherited from *Phalcon\Forms\Element*

Sets the parent form to the element

public **getForm** () inherited from *Phalcon\Forms\Element*

Returns the parent form to the element

public **setName** (mixed \$name) inherited from *Phalcon\Forms\Element*

Sets the element name

public **getName** () inherited from *Phalcon\Forms\Element*

Returns the element name

public *Phalcon\Forms\ElementInterface* **setFilters** (array | string \$filters) inherited from *Phalcon\Forms\Element*

Sets the element filters

public **addFilter** (mixed \$filter) inherited from *Phalcon\Forms\Element*

Adds a filter to current list of filters

public mixed **getFilters** () inherited from *Phalcon\Forms\Element*

Returns the element filters

public *Phalcon\Forms\ElementInterface* **addValidators** (array \$validators, [mixed \$merge]) inherited from *Phalcon\Forms\Element*

Adds a group of validators

public **addValidator** (*Phalcon\Validation\ValidatorInterface* \$validator) inherited from *Phalcon\Forms\Element*

Adds a validator to the element

public **getValidators** () inherited from *Phalcon\Forms\Element*

Returns the validators registered for the element

public **prepareAttributes** ([array \$attributes], [mixed \$useChecked]) inherited from *Phalcon\Forms\Element*

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* \$attribute, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default attribute for the element

public *mixed* **getAttribute** (*string* \$attribute, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (*array* \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (*string* \$option, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public *mixed* **getUserOption** (*string* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (*array* \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (*mixed* \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([*array* \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (*mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in _POST

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method **__toString** renders the widget without attributes

Class *Phalcon\Forms\Element\Numeric*

extends abstract class *Phalcon\Forms\Element*

implements *Phalcon\Forms\ElementInterface*

Component INPUT[type=number] for forms

Methods

public **render** ([array \$attributes])

Renders the element widget returning html

public **__construct** (string \$name, [array \$attributes]) inherited from *Phalcon\Forms\Element*

Phalcon\Forms\Element constructor

public **setForm** (*Phalcon\Forms\Form* \$form) inherited from *Phalcon\Forms\Element*

Sets the parent form to the element

public **getForm** () inherited from *Phalcon\Forms\Element*

Returns the parent form to the element

public **setName** (mixed \$name) inherited from *Phalcon\Forms\Element*

Sets the element name

public **getName** () inherited from *Phalcon\Forms\Element*

Returns the element name

public *Phalcon\Forms\ElementInterface* **setFilters** (array | string \$filters) inherited from *Phalcon\Forms\Element*

Sets the element filters

public **addFilter** (mixed \$filter) inherited from *Phalcon\Forms\Element*

Adds a filter to current list of filters

public mixed **getFilters** () inherited from *Phalcon\Forms\Element*

Returns the element filters

public *Phalcon\Forms\ElementInterface* **addValidators** (array \$validators, [mixed \$merge]) inherited from *Phalcon\Forms\Element*

Adds a group of validators

public **addValidator** (*Phalcon\Validation\ValidatorInterface* \$validator) inherited from *Phalcon\Forms\Element*

Adds a validator to the element

public **getValidators** () inherited from *Phalcon\Forms\Element*

Returns the validators registered for the element

public **prepareAttributes** ([array \$attributes], [mixed \$useChecked]) inherited from *Phalcon\Forms\Element*

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (string \$attribute, mixed \$value) inherited from *Phalcon\Forms\Element*

Sets a default attribute for the element

public mixed **getAttribute** (string \$attribute, [mixed \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (array \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (string \$option, mixed \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public mixed **getUserOption** (string \$option, [mixed \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (array \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (mixed \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([array \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (mixed \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in _POST

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method **__toString** renders the widget without attributes

Class *Phalcon\Forms\Element\Password*

extends abstract class *Phalcon\Forms\Element*

implements *Phalcon\Forms\ElementInterface*

Component INPUT[type=password] for forms

Methods

public **render** ([array \$attributes])

Renders the element widget returning html

public **__construct** (string \$name, [array \$attributes]) inherited from *Phalcon\Forms\Element*

Phalcon\Forms\Element constructor

public **setForm** (*Phalcon\Forms\Form* \$form) inherited from *Phalcon\Forms\Element*

Sets the parent form to the element

public **getForm** () inherited from *Phalcon\Forms\Element*

Returns the parent form to the element

public **setName** (mixed \$name) inherited from *Phalcon\Forms\Element*

Sets the element name

public **getName** () inherited from *Phalcon\Forms\Element*

Returns the element name

public *Phalcon\Forms\ElementInterface* **setFilters** (array | string \$filters) inherited from *Phalcon\Forms\Element*

Sets the element filters

public **addFilter** (mixed \$filter) inherited from *Phalcon\Forms\Element*

Adds a filter to current list of filters

public mixed **getFilters** () inherited from *Phalcon\Forms\Element*

Returns the element filters

public *Phalcon\Forms\ElementInterface* **addValidators** (array \$validators, [mixed \$merge]) inherited from *Phalcon\Forms\Element*

Adds a group of validators

public **addValidator** (*Phalcon\Validation\ValidatorInterface* \$validator) inherited from *Phalcon\Forms\Element*

Adds a validator to the element

public **getValidators** () inherited from *Phalcon\Forms\Element*

Returns the validators registered for the element

public **prepareAttributes** ([array \$attributes], [mixed \$useChecked]) inherited from *Phalcon\Forms\Element*

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (string \$attribute, mixed \$value) inherited from *Phalcon\Forms\Element*

Sets a default attribute for the element

public mixed **getAttribute** (string \$attribute, [mixed \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (array \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (string \$option, mixed \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public mixed **getUserOption** (string \$option, [mixed \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (array \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (mixed \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([array \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (mixed \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in _POST

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method __toString renders the widget without attributes

Class *Phalcon\Forms\Element\Radio*

extends abstract class *Phalcon\Forms\Element*

implements *Phalcon\Forms\ElementInterface*

Component INPUT[type=radio] for forms

Methods

public **render** ([array \$attributes])

Renders the element widget returning html

public **__construct** (*string* \$name, [array \$attributes]) inherited from *Phalcon\Forms\Element*

Phalcon\Forms\Element constructor

public **setForm** (*Phalcon\Forms\Form* \$form) inherited from *Phalcon\Forms\Element*

Sets the parent form to the element

public **getForm** () inherited from *Phalcon\Forms\Element*

Returns the parent form to the element

public **setName** (*mixed* \$name) inherited from *Phalcon\Forms\Element*

Sets the element name

public **getName** () inherited from *Phalcon\Forms\Element*

Returns the element name

public *Phalcon\Forms\ElementInterface* **setFilters** (array | *string* \$filters) inherited from *Phalcon\Forms\Element*

Sets the element filters

public **addFilter** (*mixed* \$filter) inherited from *Phalcon\Forms\Element*

Adds a filter to current list of filters

public *mixed* **getFilters** () inherited from *Phalcon\Forms\Element*

Returns the element filters

public *Phalcon\Forms\ElementInterface* **addValidators** (array \$validators, [mixed \$merge]) inherited from *Phalcon\Forms\Element*

Adds a group of validators

public **addValidator** (*Phalcon\Validation\ValidatorInterface* \$validator) inherited from *Phalcon\Forms\Element*

Adds a validator to the element

public **getValidators** () inherited from *Phalcon\Forms\Element*

Returns the validators registered for the element

public **prepareAttributes** ([array \$attributes], [mixed \$useChecked]) inherited from *Phalcon\Forms\Element*

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (string \$attribute, mixed \$value) inherited from *Phalcon\Forms\Element*

Sets a default attribute for the element

public mixed **getAttribute** (string \$attribute, [mixed \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (array \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (string \$option, mixed \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public mixed **getUserOption** (string \$option, [mixed \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (array \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (mixed \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([array \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (mixed \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in _POST

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method **__toString** renders the widget without attributes

Class *Phalcon\Forms\Element\Select*

extends abstract class *Phalcon\Forms\Element*

implements *Phalcon\Forms\ElementInterface*

Component SELECT (choice) for forms

Methods

public **__construct** (*string* \$name, [*object* | *array* \$options], [*array* \$attributes])

Phalcon\Forms\Element constructor

public *Phalcon\Forms\Element* **setOptions** (*array* | *object* \$options)

Set the choice's options

public *array* | *object* **getOptions** ()

Returns the choices' options

public *this* **addOption** (*array* \$option)

Adds an option to the current options

public **render** ([*array* \$attributes])

Renders the element widget returning html

public **setForm** (*Phalcon\Forms\Form* \$form) inherited from *Phalcon\Forms\Element*

Sets the parent form to the element

public **getForm** () inherited from *Phalcon\Forms\Element*

Returns the parent form to the element

public **setName** (*mixed* \$name) inherited from *Phalcon\Forms\Element*

Sets the element name

public **getName** () inherited from *Phalcon\Forms\Element*

Returns the element name

public *Phalcon\Forms\ElementInterface* **setFilters** (*array* | *string* \$filters) inherited from *Phalcon\Forms\Element*

Sets the element filters

public **addFilter** (*mixed* \$filter) inherited from *Phalcon\Forms\Element*

Adds a filter to current list of filters

public *mixed* **getFilters** () inherited from *Phalcon\Forms\Element*

Returns the element filters

public *Phalcon\Forms\ElementInterface* **addValidators** (*array* \$validators, [*mixed* \$merge]) inherited from *Phalcon\Forms\Element*

Adds a group of validators

public **addValidator** (*Phalcon\Validation\ValidatorInterface* \$validator) inherited from *Phalcon\Forms\Element*

Adds a validator to the element

public **getValidators** () inherited from *Phalcon\Forms\Element*

Returns the validators registered for the element

public **prepareAttributes** ([*array* \$attributes], [*mixed* \$useChecked]) inherited from *Phalcon\Forms\Element*

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* \$attribute, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default attribute for the element

public *mixed* **getAttribute** (*string* \$attribute, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (*array* \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (*string* \$option, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public *mixed* **getUserOption** (*string* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (*array* \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (*mixed* \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([*array* \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (*mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in `_POST`

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method `__toString` renders the widget without attributes

Class *Phalcon\Forms\Element\Submit*

extends abstract class *Phalcon\Forms\Element*

implements *Phalcon\Forms\ElementInterface*

Component `INPUT[type=submit]` for forms

Methods

public **render** ([*array* \$attributes])

Renders the element widget

public **__construct** (*string* \$name, [*array* \$attributes]) inherited from *Phalcon\Forms\Element*

Phalcon\Forms\Element constructor

public **setForm** (*Phalcon\Forms\Form* \$form) inherited from *Phalcon\Forms\Element*

Sets the parent form to the element

public **getForm** () inherited from *Phalcon\Forms\Element*

Returns the parent form to the element

public **setName** (*mixed* \$name) inherited from *Phalcon\Forms\Element*

Sets the element name

public **getName** () inherited from *Phalcon\Forms\Element*

Returns the element name

public *Phalcon\Forms\ElementInterface* **setFilters** (*array* | *string* \$filters) inherited from *Phalcon\Forms\Element*

Sets the element filters

public **addFilter** (*mixed* \$filter) inherited from *Phalcon\Forms\Element*

Adds a filter to current list of filters

public *mixed* **getFilters** () inherited from *Phalcon\Forms\Element*

Returns the element filters

public *Phalcon\Forms\ElementInterface* **addValidators** (*array* \$validators, [*mixed* \$merge]) inherited from *Phalcon\Forms\Element*

Adds a group of validators

public **addValidator** (*Phalcon\Validation\ValidatorInterface* \$validator) inherited from *Phalcon\Forms\Element*

Adds a validator to the element

public **getValidators** () inherited from *Phalcon\Forms\Element*

Returns the validators registered for the element

public **prepareAttributes** (*array* \$attributes, [*mixed* \$useChecked]) inherited from *Phalcon\Forms\Element*

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* \$attribute, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default attribute for the element

public *mixed* **getAttribute** (*string* \$attribute, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (*array* \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (*string* \$option, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public *mixed* **getUserOption** (*string* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (*array* \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (*mixed* \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([*array* \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (*mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in `_POST`

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method `__toString` renders the widget without attributes

Class *Phalcon\Forms\Element\Text*

extends abstract class *Phalcon\Forms\Element*

implements *Phalcon\Forms\ElementInterface*

Component `INPUT[type=text]` for forms

Methods

public **render** ([array \$attributes])

Renders the element widget

public **__construct** (string \$name, [array \$attributes]) inherited from *Phalcon\Forms\Element*

Phalcon\Forms\Element constructor

public **setForm** (*Phalcon\Forms\Form* \$form) inherited from *Phalcon\Forms\Element*

Sets the parent form to the element

public **getForm** () inherited from *Phalcon\Forms\Element*

Returns the parent form to the element

public **setName** (mixed \$name) inherited from *Phalcon\Forms\Element*

Sets the element name

public **getName** () inherited from *Phalcon\Forms\Element*

Returns the element name

public *Phalcon\Forms\ElementInterface* **setFilters** (array | string \$filters) inherited from *Phalcon\Forms\Element*

Sets the element filters

public **addFilter** (mixed \$filter) inherited from *Phalcon\Forms\Element*

Adds a filter to current list of filters

public mixed **getFilters** () inherited from *Phalcon\Forms\Element*

Returns the element filters

public *Phalcon\Forms\ElementInterface* **addValidators** (array \$validators, [mixed \$merge]) inherited from *Phalcon\Forms\Element*

Adds a group of validators

public **addValidator** (*Phalcon\Validation\ValidatorInterface* \$validator) inherited from *Phalcon\Forms\Element*

Adds a validator to the element

public **getValidators** () inherited from *Phalcon\Forms\Element*

Returns the validators registered for the element

public **prepareAttributes** ([array \$attributes], [mixed \$useChecked]) inherited from *Phalcon\Forms\Element*

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (string \$attribute, mixed \$value) inherited from *Phalcon\Forms\Element*

Sets a default attribute for the element

public mixed **getAttribute** (string \$attribute, [mixed \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (array \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (*string* \$option, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public *mixed* **getUserOption** (*string* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (*array* \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (*mixed* \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([*array* \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (*mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in `_POST`

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method `__toString` renders the widget without attributes

Class `Phalcon\Forms\Element\TextArea`

extends abstract class `Phalcon\Forms\Element`

implements `Phalcon\Forms\ElementInterface`

Component TEXTAREA for forms

Methods

public **render** ([array \$attributes])

Renders the element widget

public **__construct** (string \$name, [array \$attributes]) inherited from `Phalcon\Forms\Element`

Phalcon\Forms\Element constructor

public **setForm** (`Phalcon\Forms\Form` \$form) inherited from `Phalcon\Forms\Element`

Sets the parent form to the element

public **getForm** () inherited from `Phalcon\Forms\Element`

Returns the parent form to the element

public **setName** (mixed \$name) inherited from `Phalcon\Forms\Element`

Sets the element name

public **getName** () inherited from `Phalcon\Forms\Element`

Returns the element name

public `Phalcon\Forms\ElementInterface` **setFilters** (array | string \$filters) inherited from `Phalcon\Forms\Element`

Sets the element filters

public **addFilter** (mixed \$filter) inherited from `Phalcon\Forms\Element`

Adds a filter to current list of filters

public mixed **getFilters** () inherited from `Phalcon\Forms\Element`

Returns the element filters

public `Phalcon\Forms\ElementInterface` **addValidators** (array \$validators, [mixed \$merge]) inherited from `Phalcon\Forms\Element`

Adds a group of validators

public **addValidator** (`Phalcon\Validation\ValidatorInterface` \$validator) inherited from `Phalcon\Forms\Element`

Adds a validator to the element

public **getValidators** () inherited from `Phalcon\Forms\Element`

Returns the validators registered for the element

public **prepareAttributes** ([array \$attributes], [mixed \$useChecked]) inherited from `Phalcon\Forms\Element`

Returns an array of prepared attributes for Phalcon\Tag helpers according to the element parameters

public `Phalcon\Forms\ElementInterface` **setAttribute** (string \$attribute, mixed \$value) inherited from `Phalcon\Forms\Element`

Sets a default attribute for the element

public *mixed* **getAttribute** (*string* \$attribute, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an attribute if present

public **setAttributes** (*array* \$attributes) inherited from *Phalcon\Forms\Element*

Sets default attributes for the element

public **getAttributes** () inherited from *Phalcon\Forms\Element*

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setUserOption** (*string* \$option, *mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets an option for the element

public *mixed* **getUserOption** (*string* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Forms\Element*

Returns the value of an option if present

public **setUserOptions** (*array* \$options) inherited from *Phalcon\Forms\Element*

Sets options for the element

public **getUserOptions** () inherited from *Phalcon\Forms\Element*

Returns the options for the element

public **setLabel** (*mixed* \$label) inherited from *Phalcon\Forms\Element*

Sets the element label

public **getLabel** () inherited from *Phalcon\Forms\Element*

Returns the element label

public **label** ([*array* \$attributes]) inherited from *Phalcon\Forms\Element*

Generate the HTML to label the element

public *Phalcon\Forms\ElementInterface* **setDefault** (*mixed* \$value) inherited from *Phalcon\Forms\Element*

Sets a default value in case the form does not use an entity or there is no value available for the element in `_POST`

public **getDefault** () inherited from *Phalcon\Forms\Element*

Returns the default value assigned to the element

public **getValue** () inherited from *Phalcon\Forms\Element*

Returns the element value

public **getMessages** () inherited from *Phalcon\Forms\Element*

Returns the messages that belongs to the element The element needs to be attached to a form

public **hasMessages** () inherited from *Phalcon\Forms\Element*

Checks whether there are messages attached to the element

public **setMessages** (*Phalcon\Validation\Message\Group* \$group) inherited from *Phalcon\Forms\Element*

Sets the validation messages related to the element

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message) inherited from *Phalcon\Forms\Element*

Appends a message to the internal message list

public **clear** () inherited from *Phalcon\Forms\Element*

Clears every element in the form to its default value

public **__toString** () inherited from *Phalcon\Forms\Element*

Magic method **__toString** renders the widget without attributes

Class **Phalcon\Forms\Exception**

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Class Phalcon\Forms\Form

extends abstract class *Phalcon\DI\Injectable*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\DI\InjectionAwareInterface*, *Countable*, *Iterator*, *Traversable*

This component allows to build forms using an object-oriented interface

Methods

public **setValidation** (*mixed* \$validation)

...

public **getValidation** ()

...

public **__construct** ([*object* \$entity], [*array* \$userOptions])

Phalcon\Forms\Form constructor

public **setAction** (*mixed* \$action)

Sets the form's action

public **getAction** ()

Returns the form's action

public **setUserOption** (*string* \$option, *mixed* \$value)

Sets an option for the form

public **getUserOption** (*string* \$option, [*mixed* \$defaultValue])

Returns the value of an option if present

public **setUserOptions** (*array* \$options)

Sets options for the element

public **getUserOptions** ()

Returns the options for the element

public **setEntity** (*object* \$entity)

Sets the entity related to the model

public *object* **getEntity** ()

Returns the entity related to the model

public **getElements** ()

Returns the form elements added to the form

public **bind** (*array* \$data, *object* \$entity, [*array* \$whitelist])

Binds data to the entity

public **isValid** ([*array* \$data], [*object* \$entity])

Validates the form

public **getMessages** ([*mixed* \$byItemName])

Returns the messages generated in the validation

public **getMessagesFor** (*mixed* \$name)

Returns the messages generated for a specific element

public **hasMessagesFor** (*mixed* \$name)

Check if messages were generated for a specific element

public **add** (*Phalcon\Forms\ElementInterface* \$element, [*mixed* \$position], [*mixed* \$type])

Adds an element to the form

public **render** (*string* \$name, [*array* \$attributes])

Renders a specific item in the form

public **get** (*mixed* \$name)

Returns an element added to the form by its name

public **label** (*mixed* \$name, [*array* \$attributes])

Generate the label of an element added to the form including HTML

public **getLabel** (*mixed* \$name)

Returns a label for an element

public **getValue** (*mixed* \$name)

Gets a value from the internal related entity or from the default value

public **has** (*mixed* \$name)

Check if the form contains an element

public **remove** (*mixed* \$name)

Removes an element from the form

public **clear** ([*array* \$fields])

Clears every element in the form to its default value

public **count** ()

Returns the number of elements in the form

public **rewind** ()

Rewinds the internal iterator

public **current** ()

Returns the current element in the iterator

public **key** ()

Returns the current position/key in the iterator

public **next** ()

Moves the internal iteration pointer to the next position

public **valid** ()

Check if the current element in the iterator is valid

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\DiInjectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Di\Injectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Di\Injectable*

Returns the internal event manager

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method **__get**

Class *Phalcon\Forms\Manager*

Methods

public **create** (*string* \$name, [*object* \$entity])

Creates a form registering it in the forms manager

public **get** (*mixed* \$name)

Returns a form by its name

public **has** (*mixed* \$name)

Checks if a form is registered in the forms manager

public **set** (*mixed* \$name, *Phalcon\Forms\Form* \$form)

Registers a form in the Forms Manager

Class *Phalcon\Http\Cookie*

implements Phalcon\Http\CookieInterface, Phalcon\Di\InjectionAwareInterface

Provide OO wrappers to manage a HTTP cookie

Methods

public **__construct** (*string* \$name, [*mixed* \$value], [*int* \$expire], [*string* \$path], [*boolean* \$secure], [*string* \$domain], [*boolean* \$httpOnly])

Phalcon\Http\Cookie constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injector

public **getDI** ()

Returns the internal dependency injector

public *Phalcon\Http\Cookie* **setValue** (*string* \$value)

Sets the cookie's value

public *mixed* **getValue** ([*string* | *array* \$filters], [*string* \$defaultValue])

Returns the cookie's value

public **send** ()

Sends the cookie to the HTTP client Stores the cookie definition in session

public **restore** ()

Reads the cookie-related info from the SESSION to restore the cookie as it was set This method is automatically called internally so normally you don't need to call it

public **delete** ()

Deletes the cookie by setting an expire time in the past

public **useEncryption** (*mixed* \$useEncryption)

Sets if the cookie must be encrypted/decrypted automatically

public **isUsingEncryption** ()

Check if the cookie is using implicit encryption

public **setExpiration** (*mixed* \$expire)

Sets the cookie's expiration time

public **getExpiration** ()

Returns the current expiration time

public **setPath** (*mixed* \$path)

Sets the cookie's expiration time

public **getName** ()

Returns the current cookie's name

public **getPath** ()

Returns the current cookie's path

public **setDomain** (*mixed* \$domain)

Sets the domain that the cookie is available to

public **getDomain** ()

Returns the domain that the cookie is available to

public **setSecure** (*mixed* \$secure)

Sets if the cookie must only be sent when the connection is secure (HTTPS)

public **getSecure** ()

Returns whether the cookie must only be sent when the connection is secure (HTTPS)

public **setHttpOnly** (*mixed* \$httpOnly)

Sets if the cookie is accessible only through the HTTP protocol

public **getHttpOnly** ()

Returns if the cookie is accessible only through the HTTP protocol

public **__toString** ()

Magic `__toString` method converts the cookie's value to string

Class `Phalcon\Http\Cookie\Exception`

extends class `Phalcon\Exception`

implements `Throwable`

Methods

final private `Exception __clone ()` inherited from `Exception`

Clone the exception

public `__construct ([string $message], [int $code], [Exception $previous])` inherited from `Exception`

Exception constructor

public `__wakeup ()` inherited from `Exception`

...

final public `string getMessage ()` inherited from `Exception`

Gets the Exception message

final public `int getCode ()` inherited from `Exception`

Gets the Exception code

final public `string getFile ()` inherited from `Exception`

Gets the file in which the exception occurred

final public `int getLine ()` inherited from `Exception`

Gets the line in which the exception occurred

final public `array getTrace ()` inherited from `Exception`

Gets the stack trace

final public `Exception getPrevious ()` inherited from `Exception`

Returns previous Exception

final public `Exception getTraceAsString ()` inherited from `Exception`

Gets the stack trace as a string

public `string __toString ()` inherited from `Exception`

String representation of the exception

Class `Phalcon\Http\Request`

implements `Phalcon\Http\RequestInterface`, `Phalcon\Di\InjectionAwareInterface`

Encapsulates request information for easy and secure access from application controllers.

The request object is a simple value object that is passed between the dispatcher and controller classes. It packages the HTTP request environment.

```
<?php

use Phalcon\Http\Request;

$request = new Request();

if ($request->isPost() && $request->isAjax()) {
    echo "Request was made using POST and AJAX";
}

$request->getServer("HTTP_HOST"); // Retrieve SERVER variables
$request->getMethod();           // GET, POST, PUT, DELETE, HEAD, OPTIONS, PATCH, PURGE, TRACE, CONNECT
$request->getLanguages();        // An array of languages the client accepts
```

Methods

public **getHttpRequestMethodParameterOverride** ()

...

public **setHttpRequestMethodParameterOverride** (*mixed* \$httpMethodParameterOverride)

...

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injector

public **getDI** ()

Returns the internal dependency injector

public **get** (*mixed* \$name), (*mixed* \$filters), (*mixed* \$defaultValue), (*mixed* \$notAllowEmpty), (*mixed* \$noRecursive)

Gets a variable from the \$_REQUEST superglobal applying filters if needed. If no parameters are given the \$_REQUEST superglobal is returned

```
<?php

// Returns value from $_REQUEST["user_email"] without sanitizing
$userEmail = $request->get("user_email");

// Returns value from $_REQUEST["user_email"] with sanitizing
$userEmail = $request->get("user_email", "email");
```

public **getPost** (*mixed* \$name), (*mixed* \$filters), (*mixed* \$defaultValue), (*mixed* \$notAllowEmpty), (*mixed* \$noRecursive)

Gets a variable from the \$_POST superglobal applying filters if needed. If no parameters are given the \$_POST superglobal is returned

```
<?php

// Returns value from $_POST["user_email"] without sanitizing
$userEmail = $request->getPost("user_email");

// Returns value from $_POST["user_email"] with sanitizing
$userEmail = $request->getPost("user_email", "email");
```

public **getPut** ([mixed \$name], [mixed \$filters], [mixed \$defaultValue], [mixed \$notAllowEmpty], [mixed \$noRecursive])

Gets a variable from put request

```
<?php

// Returns value from $_PUT["user_email"] without sanitizing
$userEmail = $request->getPut("user_email");

// Returns value from $_PUT["user_email"] with sanitizing
$userEmail = $request->getPut("user_email", "email");
```

public **getQuery** ([mixed \$name], [mixed \$filters], [mixed \$defaultValue], [mixed \$notAllowEmpty], [mixed \$noRecursive])

Gets variable from \$_GET superglobal applying filters if needed If no parameters are given the \$_GET superglobal is returned

```
<?php

// Returns value from $_GET["id"] without sanitizing
$id = $request->getQuery("id");

// Returns value from $_GET["id"] with sanitizing
$id = $request->getQuery("id", "int");

// Returns value from $_GET["id"] with a default value
$id = $request->getQuery("id", null, 150);
```

final protected **getHelper** (array \$source, [mixed \$name], [mixed \$filters], [mixed \$defaultValue], [mixed \$notAllowEmpty], [mixed \$noRecursive])

Helper to get data from superglobals, applying filters if needed. If no parameters are given the superglobal is returned.

public **getServer** (mixed \$name)

Gets variable from \$_SERVER superglobal

public **has** (mixed \$name)

Checks whether \$_REQUEST superglobal has certain index

public **hasPost** (mixed \$name)

Checks whether \$_POST superglobal has certain index

public **hasPut** (mixed \$name)

Checks whether the PUT data has certain index

public **hasQuery** (mixed \$name)

Checks whether \$_GET superglobal has certain index

final public **hasServer** (mixed \$name)

Checks whether \$_SERVER superglobal has certain index

final public **getHeader** (mixed \$header)

Gets HTTP header from request data

public **getScheme** ()

Gets HTTP schema (http/https)

public **isAjax** ()

Checks whether request has been made using ajax

public **isSoap** ()

Checks whether request has been made using SOAP

public **isSoapRequested** ()

Alias of isSoap(). It will be deprecated in future versions

public **isSecure** ()

Checks whether request has been made using any secure layer

public **isSecureRequest** ()

Alias of isSecure(). It will be deprecated in future versions

public **getRawBody** ()

Gets HTTP raw request body

public **getJsonRawBody** ([mixed \$associative])

Gets decoded JSON HTTP raw request body

public **getServerAddress** ()

Gets active server address IP

public **getServerName** ()

Gets active server name

public **getHttpHost** ()

Gets host name used by the request. *Request::getHttpHost* trying to find host name in following order: - `$_SERVER["HTTP_HOST"]` - `$_SERVER["SERVER_NAME"]` - `$_SERVER["SERVER_ADDR"]` Optionally *Request::getHttpHost* validates and clean host name. The *Request::\$_strictHostCheck* can be used to validate host name. Note: validation and cleaning have a negative performance impact because they use regular expressions.

```
<?php
use Phalcon\Http\Request;

$request = new Request;

$_SERVER["HTTP_HOST"] = "example.com";
$request->getHttpHost(); // example.com

$_SERVER["HTTP_HOST"] = "example.com:8080";
$request->getHttpHost(); // example.com:8080

$request->setStrictHostCheck(true);
$_SERVER["HTTP_HOST"] = "ex=am~ple.com";
$request->getHttpHost(); // UnexpectedValueException

$_SERVER["HTTP_HOST"] = "ExAmPlE.com";
$request->getHttpHost(); // example.com
```


public **setStrictHostCheck** ([*mixed* \$flag])

Sets if the *Request::getHttpHost* method must be use strict validation of host name or not

public **isStrictHostCheck** ()

Checks if the *Request::getHttpHost* method will be use strict validation of host name or not

public **getPort** ()

Gets information about the port on which the request is made.

final public **getURI** ()

Gets HTTP URI which request has been made

public **getClientAddress** ([*mixed* \$trustForwardedHeader])

Gets most possible client IPv4 Address. This method searches in \$_SERVER["REMOTE_ADDR"] and optionally in \$_SERVER["HTTP_X_FORWARDED_FOR"]

final public **getMethod** ()

Gets HTTP method which request has been made If the X-HTTP-Method-Override header is set, and if the method is a POST, then it is used to determine the “real” intended HTTP method. The *_method* request parameter can also be used to determine the HTTP method, but only if *setHttpMethodParameterOverride(true)* has been called. The method is always an uppercased string.

public **getUserAgent** ()

Gets HTTP user agent used to made the request

public **isValidHttpMethod** (*mixed* \$method)

Checks if a method is a valid HTTP method

public **isMethod** (*mixed* \$methods, [*mixed* \$strict])

Check if HTTP method match any of the passed methods When strict is true it checks if validated methods are real HTTP methods

public **isPost** ()

Checks whether HTTP method is POST. if \$_SERVER["REQUEST_METHOD"]==="POST"

public **isGet** ()

Checks whether HTTP method is GET. if \$_SERVER["REQUEST_METHOD"]==="GET"

public **isPut** ()

Checks whether HTTP method is PUT. if \$_SERVER["REQUEST_METHOD"]==="PUT"

public **isPatch** ()

Checks whether HTTP method is PATCH. if \$_SERVER["REQUEST_METHOD"]==="PATCH"

public **isHead** ()

Checks whether HTTP method is HEAD. if \$_SERVER["REQUEST_METHOD"]==="HEAD"

public **isDelete** ()

Checks whether HTTP method is DELETE. if \$_SERVER["REQUEST_METHOD"]==="DELETE"

public **isOptions** ()

Checks whether HTTP method is OPTIONS. if \$_SERVER["REQUEST_METHOD"]==="OPTIONS"

public **isPurge** ()

Checks whether HTTP method is PURGE (Squid and Varnish support). if
_SERVER["REQUEST_METHOD"]=="PURGE"

public **isTrace** ()

Checks whether HTTP method is TRACE. if _SERVER["REQUEST_METHOD"]=="TRACE"

public **isConnect** ()

Checks whether HTTP method is CONNECT. if _SERVER["REQUEST_METHOD"]=="CONNECT"

public **hasFiles** ([*mixed* \$onlySuccessful])

Checks whether request include attached files

final protected **hasFileHelper** (*mixed* \$data, *mixed* \$onlySuccessful)

Recursively counts file in an array of files

public **getUploadedFiles** ([*mixed* \$onlySuccessful])

Gets attached files as Phalcon\Http\Request\File instances

final protected **smoothFiles** (array \$names, array \$types, array \$tmp_names, array \$sizes, array \$errors, *mixed* \$prefix)

Smooth out \$_FILES to have plain array with all files uploaded

public **getHeaders** ()

Returns the available headers in the request

```
<?php

$_SERVER = [
    "PHP_AUTH_USER" => "phalcon",
    "PHP_AUTH_PW"   => "secret",
];

$headers = $request->getHeaders();

echo $headers["Authorization"]; // Basic cGhhbGNvbjpwZWNyZXQ=
```

public **getHTTPReferer** ()

Gets web page that refers active request. ie: <http://www.google.com>

final protected **getBestQuality** (array \$qualityParts, *mixed* \$name)

Process a request header and return the one with best quality

public **getContentType** ()

Gets content type which request has been made

public **getAcceptableContent** ()

Gets an array with mime/types and their quality accepted by the browser/client from _SERVER["HTTP_ACCEPT"]

public **getBestAccept** ()

Gets best mime/type accepted by the browser/client from _SERVER["HTTP_ACCEPT"]

public **getClientCharsets** ()

Gets a charsets array and their quality accepted by the browser/client from _SERVER["HTTP_ACCEPT_CHARSET"]

public **getBestCharset** ()

Gets best charset accepted by the browser/client from `_SERVER["HTTP_ACCEPT_CHARSET"]`

public **getLanguages** ()

Gets languages array and their quality accepted by the browser/client from `_SERVER["HTTP_ACCEPT_LANGUAGE"]`

public **getBestLanguage** ()

Gets best language accepted by the browser/client from `_SERVER["HTTP_ACCEPT_LANGUAGE"]`

public **getBasicAuth** ()

Gets auth info accepted by the browser/client from `$_SERVER["PHP_AUTH_USER"]`

public **getDigestAuth** ()

Gets auth info accepted by the browser/client from `$_SERVER["PHP_AUTH_DIGEST"]`

final protected **__getQualityHeader** (*mixed* \$serverIndex, *mixed* \$name)

Process a request header and return an array of values with their qualities

Class `Phalcon\Http\Request\Exception`

extends class `Phalcon\Exception`

implements `Throwable`

Methods

final private `Exception` **__clone** () inherited from `Exception`

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [`Exception` \$previous]) inherited from `Exception`

Exception constructor

public **__wakeup** () inherited from `Exception`

...

final public *string* **getMessage** () inherited from `Exception`

Gets the Exception message

final public *int* **getCode** () inherited from `Exception`

Gets the Exception code

final public *string* **getFile** () inherited from `Exception`

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from `Exception`

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from `Exception`

Gets the stack trace

final public `Exception` **getPrevious** () inherited from `Exception`

Returns previous Exception

final public [Exception](#) **getTraceAsString()** inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString()** inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Http\Request\File](#)

implements [Phalcon\Http\Request\FileInterface](#)

Provides OO wrappers to the \$_FILES superglobal

```
<?php

use Phalcon\Mvc\Controller;

class PostsController extends Controller
{
    public function uploadAction()
    {
        // Check if the user has uploaded files
        if ($this->request->hasFiles() == true) {
            // Print the real file names and their sizes
            foreach ($this->request->getUploadedFiles() as $file) {
                echo $file->getName(), " ", $file->getSize(), "\n";
            }
        }
    }
}
```

Methods

public **getError()**

public **getKey()**

public **getExtension()**

public **__construct** (array \$file, [mixed \$key])

[Phalcon\Http\Request\File](#) constructor

public **getSize()**

Returns the file size of the uploaded file

public **getName()**

Returns the real name of the uploaded file

public **getTempName()**

Returns the temporary name of the uploaded file

public **getType()**

Returns the mime type reported by the browser This mime type is not completely secure, use [getRealType\(\)](#) instead

public **getRealType()**

Gets the real mime type of the upload file using finfo

public **isUploadedFile** ()

Checks whether the file has been uploaded via Post.

public **moveTo** (*mixed* \$destination)

Moves the temporary file to a destination within the application

Class Phalcon\Http\Response

implements Phalcon\Http\ResponseInterface, Phalcon\Di\InjectionAwareInterface

Part of the HTTP cycle is return responses to the clients. Phalcon\Http\Response is the Phalcon component responsible to achieve this task. HTTP responses are usually composed by headers and body.

```
<?php

$response = new \Phalcon\Http\Response();

$response->setStatusCode(200, "OK");
$response->setContent("<html><body>Hello</body></html>");

$response->send();
```

Methods

public **__construct** ([*mixed* \$content], [*mixed* \$code], [*mixed* \$status])

Phalcon\Http\Response constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injector

public **getDI** ()

Returns the internal dependency injector

public **setStatusCode** (*mixed* \$code, [*mixed* \$message])

Sets the HTTP response code

```
<?php

$response->setStatusCode(404, "Not Found");
```

public **getStatusCode** ()

Returns the status code

```
<?php

print_r(
    $response->getStatusCode()
);
```

public **setHeaders** (*Phalcon\Http\Response\HeadersInterface* \$headers)

Sets a headers bag for the response externally

public **getHeaders** ()

Returns headers set by the user

public **setCookies** (*Phalcon\Http\Response\CookiesInterface* \$cookies)

Sets a cookies bag for the response externally

public *Phalcon\Http\Response\CookiesInterface* **getCookies** ()

Returns cookies set by the user

public **setHeader** (*mixed* \$name, *mixed* \$value)

Overwrites a header in the response

```
<?php
$response->setHeader("Content-Type", "text/plain");
```

public **setRawHeader** (*mixed* \$header)

Send a raw header to the response

```
<?php
$response->setRawHeader("HTTP/1.1 404 Not Found");
```

public **resetHeaders** ()

Resets all the established headers

public **setExpires** (*DateTime* \$datetime)

Sets an Expires header in the response that allows to use the HTTP cache

```
<?php
$this->response->setExpires (
    new DateTime ()
);
```

public **setLastModified** (*DateTime* \$datetime)

Sets Last-Modified header

```
<?php
$this->response->setLastModified (
    new DateTime ()
);
```

public **setCache** (*mixed* \$minutes)

Sets Cache headers to use HTTP cache

```
<?php
$this->response->setCache (60);
```

public **setNotModified** ()

Sends a Not-Modified response

public **setContentType** (*mixed* \$contentType, [*mixed* \$charset])

Sets the response content-type mime, optionally the charset

```
<?php
$response->setContentType("application/pdf");
$response->setContentType("text/plain", "UTF-8");
```

public **setContentLength** (*mixed* \$contentLength)

Sets the response content-length

```
<?php
$response->setContentLength(2048);
```

public **setEtag** (*mixed* \$etag)

Set a custom ETag

```
<?php
$response->setEtag(md5(time()));
```

public **redirect** ([*mixed* \$location], [*mixed* \$externalRedirect], [*mixed* \$statusCode])

Redirect by HTTP to another action or URL

```
<?php
// Using a string redirect (internal/external)
$response->redirect("posts/index");
$response->redirect("http://en.wikipedia.org", true);
$response->redirect("http://www.example.com/new-location", true, 301);

// Making a redirection based on a named route
$response->redirect(
    [
        "for"          => "index-lang",
        "lang"         => "jp",
        "controller" => "index",
    ]
);
```

public **setContent** (*mixed* \$content)

Sets HTTP response body

```
<?php
$response->setContent("<h1>Hello!</h1>");
```

public **setJsonContent** (*mixed* \$content, [*mixed* \$jsonOptions], [*mixed* \$depth])

Sets HTTP response body. The parameter is automatically converted to JSON and also sets default header: Content-Type: “application/json; charset=UTF-8”

```
<?php

$response->setJsonContent (
    [
        "status" => "OK",
    ]
);
```

public **appendContent** (*mixed* \$content)

Appends a string to the HTTP response body

public **getContent** ()

Gets the HTTP response body

public **isSent** ()

Check if the response is already sent

public **sendHeaders** ()

Sends headers to the client

public **sendCookies** ()

Sends cookies to the client

public **send** ()

Prints out HTTP response to the client

public **setFileToSend** (*mixed* \$filePath, [*mixed* \$attachmentName], [*mixed* \$attachment])

Sets an attached file to be sent at the end of the request

Class **Phalcon\Http\Response\Cookies**

implements [Phalcon\Http\Response\CookiesInterface](#), [Phalcon\Di\InjectionAwareInterface](#)

This class is a bag to manage the cookies A cookies bag is automatically registered as part of the ‘response’ service in the DI

Methods

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injector

public **getDI** ()

Returns the internal dependency injector

public **useEncryption** (*mixed* \$useEncryption)

Set if cookies in the bag must be automatically encrypted/decrypted

public **isUsingEncryption** ()

Returns if the bag is automatically encrypting/decrypting cookies

public **set** (*mixed* \$name, [*mixed* \$value], [*mixed* \$expire], [*mixed* \$path], [*mixed* \$secure], [*mixed* \$domain], [*mixed* \$httpOnly])

Sets a cookie to be sent at the end of the request This method overrides any cookie set before with the same name

public **get** (*mixed* \$name)

Gets a cookie from the bag

public **has** (*mixed* \$name)

Check if a cookie is defined in the bag or exists in the `_COOKIE` superglobal

public **delete** (*mixed* \$name)

Deletes a cookie by its name This method does not removes cookies from the `_COOKIE` superglobal

public **send** ()

Sends the cookies to the client Cookies aren't sent if headers are sent in the current request

public **reset** ()

Reset set cookies

Class `Phalcon\Http\Response\Exception`

extends class `Phalcon\Exception`

implements `Throwable`

Methods

final private `Exception` **__clone** () inherited from `Exception`

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [`Exception` \$previous]) inherited from `Exception`

Exception constructor

public **__wakeup** () inherited from `Exception`

...

final public *string* **getMessage** () inherited from `Exception`

Gets the Exception message

final public *int* **getCode** () inherited from `Exception`

Gets the Exception code

final public *string* **getFile** () inherited from `Exception`

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from `Exception`

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from `Exception`

Gets the stack trace

final public `Exception` **getPrevious** () inherited from `Exception`

Returns previous Exception

final public `Exception` **getTraceAsString** () inherited from `Exception`

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Http\Response\Headers](#)

implements [Phalcon\Http\Response\HeadersInterface](#)

This class is a bag to manage the response headers

Methods

public **set** (*mixed* \$name, *mixed* \$value)

Sets a header to be sent at the end of the request

public **get** (*mixed* \$name)

Gets a header value from the internal bag

public **setRaw** (*mixed* \$header)

Sets a raw header to be sent at the end of the request

public **remove** (*mixed* \$header)

Removes a header to be sent at the end of the request

public **send** ()

Sends the headers to the client

public **reset** ()

Reset set headers

public **toArray** ()

Returns the current headers as an array

public static **__set_state** (*array* \$data)

Restore a [\Phalcon\Http\Response\Headers](#) object

Class [Phalcon\Image](#)

Constants

integer **NONE**

integer **WIDTH**

integer **HEIGHT**

integer **AUTO**

integer **INVERSE**

integer **PRECISE**

integer **TENSILE**

integer **HORIZONTAL**

integer **VERTICAL**

Abstract class **Phalcon\Image\Adapter**

implements **Phalcon\Image\AdapterInterface**

All image adapters must use this class

Methods

public **getImage** ()

...

public **getRealpath** ()

...

public **getWidth** ()

Image width

public **getHeight** ()

Image height

public **getType** ()

Image type Driver dependent

public **getMime** ()

Image mime type

public **resize** ([*mixed* \$width], [*mixed* \$height], [*mixed* \$master])

Resize the image to the given size

public **liquidRescale** (*mixed* \$width, *mixed* \$height, [*mixed* \$deltaX], [*mixed* \$rigidity])

This method scales the images using liquid rescaling method. Only support Imagick

public **crop** (*mixed* \$width, *mixed* \$height, [*mixed* \$offsetX], [*mixed* \$offsetY])

Crop an image to the given size

public **rotate** (*mixed* \$degrees)

Rotate the image by a given amount

public **flip** (*mixed* \$direction)

Flip the image along the horizontal or vertical axis

public **sharpen** (*mixed* \$amount)

Sharpen the image by a given amount

public **reflection** (*mixed* \$height, [*mixed* \$opacity], [*mixed* \$fadeIn])

Add a reflection to an image

public **watermark** (**Phalcon\Image\Adapter** \$watermark, [*mixed* \$offsetX], [*mixed* \$offsetY], [*mixed* \$opacity])

Add a watermark to an image with the specified opacity

public **text** (*mixed* \$text, [*mixed* \$offsetX], [*mixed* \$offsetY], [*mixed* \$opacity], [*mixed* \$color], [*mixed* \$size], [*mixed* \$fontfile])

Add a text to an image with a specified opacity

public **mask** (*Phalcon\Image\Adapter* \$watermark)

Composite one image onto another

public **background** (*mixed* \$color, [*mixed* \$opacity])

Set the background color of an image

public **blur** (*mixed* \$radius)

Blur image

public **pixelate** (*mixed* \$amount)

Pixelate image

public **save** ([*mixed* \$file], [*mixed* \$quality])

Save the image

public **render** ([*mixed* \$ext], [*mixed* \$quality])

Render the image and return the binary string

Class *Phalcon\Image\Adapter\Gd*

extends abstract class *Phalcon\Image\Adapter*

implements *Phalcon\Image\AdapterInterface*

Methods

public static **check** ()

...

public **__construct** (*mixed* \$file, [*mixed* \$width], [*mixed* \$height])

...

protected **_resize** (*mixed* \$width, *mixed* \$height)

...

protected **_crop** (*mixed* \$width, *mixed* \$height, *mixed* \$offsetX, *mixed* \$offsetY)

...

protected **_rotate** (*mixed* \$degrees)

...

protected **_flip** (*mixed* \$direction)

...

protected **_sharpen** (*mixed* \$amount)

...

protected **_reflection** (*mixed* \$height, *mixed* \$opacity, *mixed* \$fadeIn)

```

...
protected _watermark (Phalcon\Image\Adapter $watermark, mixed $offsetX, mixed $offsetY, mixed $opacity)
...
protected _text (mixed $text, mixed $offsetX, mixed $offsetY, mixed $opacity, mixed $r, mixed $g, mixed $b, mixed
$size, mixed $fontfile)
...
protected _mask (Phalcon\Image\Adapter $mask)
...
protected _background (mixed $r, mixed $g, mixed $b, mixed $opacity)
...
protected _blur (mixed $radius)
...
protected _pixelate (mixed $amount)
...
protected _save (mixed $file, mixed $quality)
...
protected _render (mixed $ext, mixed $quality)
...
protected _create (mixed $width, mixed $height)
...
public __destruct ()
...
public getImage () inherited from Phalcon\Image\Adapter
...
public getRealpath () inherited from Phalcon\Image\Adapter
...
public getWidth () inherited from Phalcon\Image\Adapter
Image width
public getHeight () inherited from Phalcon\Image\Adapter
Image height
public getType () inherited from Phalcon\Image\Adapter
Image type Driver dependent
public getMime () inherited from Phalcon\Image\Adapter
Image mime type
public resize ([mixed $width], [mixed $height], [mixed $master]) inherited from Phalcon\Image\Adapter
Resize the image to the given size

```

public **liquidRescale** (*mixed* \$width, *mixed* \$height, [*mixed* \$deltaX], [*mixed* \$rigidity]) inherited from *Phalcon\Image\Adapter*

This method scales the images using liquid rescaling method. Only support Imagick

public **crop** (*mixed* \$width, *mixed* \$height, [*mixed* \$offsetX], [*mixed* \$offsetY]) inherited from *Phalcon\Image\Adapter*

Crop an image to the given size

public **rotate** (*mixed* \$degrees) inherited from *Phalcon\Image\Adapter*

Rotate the image by a given amount

public **flip** (*mixed* \$direction) inherited from *Phalcon\Image\Adapter*

Flip the image along the horizontal or vertical axis

public **sharpen** (*mixed* \$amount) inherited from *Phalcon\Image\Adapter*

Sharpen the image by a given amount

public **reflection** (*mixed* \$height, [*mixed* \$opacity], [*mixed* \$fadeIn]) inherited from *Phalcon\Image\Adapter*

Add a reflection to an image

public **watermark** (*Phalcon\Image\Adapter* \$watermark, [*mixed* \$offsetX], [*mixed* \$offsetY], [*mixed* \$opacity]) inherited from *Phalcon\Image\Adapter*

Add a watermark to an image with the specified opacity

public **text** (*mixed* \$text, [*mixed* \$offsetX], [*mixed* \$offsetY], [*mixed* \$opacity], [*mixed* \$color], [*mixed* \$size], [*mixed* \$fontfile]) inherited from *Phalcon\Image\Adapter*

Add a text to an image with a specified opacity

public **mask** (*Phalcon\Image\Adapter* \$watermark) inherited from *Phalcon\Image\Adapter*

Composite one image onto another

public **background** (*mixed* \$color, [*mixed* \$opacity]) inherited from *Phalcon\Image\Adapter*

Set the background color of an image

public **blur** (*mixed* \$radius) inherited from *Phalcon\Image\Adapter*

Blur image

public **pixelate** (*mixed* \$amount) inherited from *Phalcon\Image\Adapter*

Pixelate image

public **save** ([*mixed* \$file], [*mixed* \$quality]) inherited from *Phalcon\Image\Adapter*

Save the image

public **render** ([*mixed* \$ext], [*mixed* \$quality]) inherited from *Phalcon\Image\Adapter*

Render the image and return the binary string

Class *Phalcon\Image\Adapter\Imagick*

extends abstract class *Phalcon\Image\Adapter*

implements *Phalcon\Image\AdapterInterface*

Image manipulation support. Allows images to be resized, cropped, etc.

```
<?php

$image = new \Phalcon\Image\Adapter\Imagick("upload/test.jpg");

$image->resize(200, 200)->rotate(90)->crop(100, 100);

if ($image->save()) {
    echo "success";
}
```

Methods

public static **check** ()

Checks if Imagick is enabled

public **__construct** (*mixed* \$file, [*mixed* \$width], [*mixed* \$height])

\Phalcon\Image\Adapter\Imagick constructor

protected **_resize** (*mixed* \$width, *mixed* \$height)

Execute a resize.

protected **_liquidRescale** (*mixed* \$width, *mixed* \$height, *mixed* \$deltaX, *mixed* \$rigidity)

This method scales the images using liquid rescaling method. Only support Imagick

protected **_crop** (*mixed* \$width, *mixed* \$height, *mixed* \$offsetX, *mixed* \$offsetY)

Execute a crop.

protected **_rotate** (*mixed* \$degrees)

Execute a rotation.

protected **_flip** (*mixed* \$direction)

Execute a flip.

protected **_sharpen** (*mixed* \$amount)

Execute a sharpen.

protected **_reflection** (*mixed* \$height, *mixed* \$opacity, *mixed* \$fadeIn)

Execute a reflection.

protected **_watermark** (*Phalcon\Image\Adapter* \$image, *mixed* \$offsetX, *mixed* \$offsetY, *mixed* \$opacity)

Execute a watermarking.

protected **_text** (*mixed* \$text, *mixed* \$offsetX, *mixed* \$offsetY, *mixed* \$opacity, *mixed* \$r, *mixed* \$g, *mixed* \$b, *mixed* \$size, *mixed* \$fontfile)

Execute a text

protected **_mask** (*Phalcon\Image\Adapter* \$image)

Composite one image onto another

protected **_background** (*mixed* \$r, *mixed* \$g, *mixed* \$b, *mixed* \$opacity)

Execute a background.

protected **_blur** (*mixed* \$radius)

Blur image

protected **_pixelate** (*mixed* \$amount)

Pixelate image

protected **_save** (*mixed* \$file, *mixed* \$quality)

Execute a save.

protected **_render** (*mixed* \$extension, *mixed* \$quality)

Execute a render.

public **__destruct** ()

Destroys the loaded image to free up resources.

public **getInternalImInstance** ()

Get instance

public **setResourceLimit** (*mixed* \$type, *mixed* \$limit)

Sets the limit for a particular resource in megabytes

public **getImage** () inherited from *Phalcon\Image\Adapter*

...

public **getRealpath** () inherited from *Phalcon\Image\Adapter*

...

public **getWidth** () inherited from *Phalcon\Image\Adapter*

Image width

public **getHeight** () inherited from *Phalcon\Image\Adapter*

Image height

public **getType** () inherited from *Phalcon\Image\Adapter*

Image type Driver dependent

public **getMime** () inherited from *Phalcon\Image\Adapter*

Image mime type

public **resize** ([*mixed* \$width], [*mixed* \$height], [*mixed* \$master]) inherited from *Phalcon\Image\Adapter*

Resize the image to the given size

public **liquidRescale** (*mixed* \$width, *mixed* \$height, [*mixed* \$deltaX], [*mixed* \$rigidity]) inherited from *Phalcon\Image\Adapter*

This method scales the images using liquid rescaling method. Only support Imagick

public **crop** (*mixed* \$width, *mixed* \$height, [*mixed* \$offsetX], [*mixed* \$offsetY]) inherited from *Phalcon\Image\Adapter*

Crop an image to the given size

public **rotate** (*mixed* \$degrees) inherited from *Phalcon\Image\Adapter*

Rotate the image by a given amount

public **flip** (*mixed* \$direction) inherited from *Phalcon\Image\Adapter*

Flip the image along the horizontal or vertical axis

public **sharpen** (*mixed* \$amount) inherited from *Phalcon\Image\Adapter*

Sharpen the image by a given amount

public **reflection** (*mixed* \$height, [*mixed* \$opacity], [*mixed* \$fadeIn]) inherited from *Phalcon\Image\Adapter*

Add a reflection to an image

public **watermark** (*Phalcon\Image\Adapter* \$watermark, [*mixed* \$offsetX], [*mixed* \$offsetY], [*mixed* \$opacity]) inherited from *Phalcon\Image\Adapter*

Add a watermark to an image with the specified opacity

public **text** (*mixed* \$text, [*mixed* \$offsetX], [*mixed* \$offsetY], [*mixed* \$opacity], [*mixed* \$color], [*mixed* \$size], [*mixed* \$fontfile]) inherited from *Phalcon\Image\Adapter*

Add a text to an image with a specified opacity

public **mask** (*Phalcon\Image\Adapter* \$watermark) inherited from *Phalcon\Image\Adapter*

Composite one image onto another

public **background** (*mixed* \$color, [*mixed* \$opacity]) inherited from *Phalcon\Image\Adapter*

Set the background color of an image

public **blur** (*mixed* \$radius) inherited from *Phalcon\Image\Adapter*

Blur image

public **pixelate** (*mixed* \$amount) inherited from *Phalcon\Image\Adapter*

Pixelate image

public **save** ([*mixed* \$file], [*mixed* \$quality]) inherited from *Phalcon\Image\Adapter*

Save the image

public **render** ([*mixed* \$ext], [*mixed* \$quality]) inherited from *Phalcon\Image\Adapter*

Render the image and return the binary string

Class *Phalcon\Image\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from [Exception](#)

Gets the Exception code

final public *string* **getFile** () inherited from [Exception](#)

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from [Exception](#)

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class Phalcon\Kernel

Methods

public static **preComputeHashKey** (*mixed* \$key)

...

Class Phalcon\Loader

implements [Phalcon\Events\EventsAwareInterface](#)

This component helps to load your project classes automatically based on some conventions

```
<?php

use Phalcon\Loader;

// Creates the autoloader
$loader = new Loader();

// Register some namespaces
$loader->registerNamespaces(
    [
        "Example\\Base"      => "vendor/example/base/",
        "Example\\Adapter"   => "vendor/example/adapter/",
        "Example"            => "vendor/example/",
    ]
);

// Register autoloader
$loader->register();
```

```
// Requiring this class will automatically include file vendor/example/adapter/Some.  
↪php  
$adapter = new \Example\Adapter\Some();
```

Methods

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager)

Sets the events manager

public **getEventManager** ()

Returns the internal event manager

public **setExtensions** (array \$extensions)

Sets an array of file extensions that the loader must try in each attempt to locate the file

public **getExtensions** ()

Returns the file extensions registered in the loader

public **registerNamespaces** (array \$namespaces, [*mixed* \$merge])

Register namespaces and their related directories

protected **prepareNamespace** (array \$namespace)

...

public **getNamespaces** ()

Returns the namespaces currently registered in the autoloader

public **registerDirs** (array \$directories, [*mixed* \$merge])

Register directories in which “not found” classes could be found

public **getDirs** ()

Returns the directories currently registered in the autoloader

public **registerFiles** (array \$files, [*mixed* \$merge])

Registers files that are “non-classes” hence need a “require”. This is very useful for including files that only have functions

public **getFiles** ()

Returns the files currently registered in the autoloader

public **registerClasses** (array \$classes, [*mixed* \$merge])

Register classes and their locations

public **getClasses** ()

Returns the class-map currently registered in the autoloader

public **register** ([*mixed* \$prepend])

Register the autoload method

public **unregister** ()

Unregister the autoload method

public **loadFiles** ()

Checks if a file exists and then adds the file by doing virtual require

public **autoLoad** (*mixed* \$className)

Autoloads the registered classes

public **getFoundPath** ()

Get the path when a class was found

public **getCheckedPath** ()

Get the path the loader is checking for a path

Class **Phalcon\Loader\Exception**

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Abstract class `Phalcon\Logger`

Constants

integer **SPECIAL**

integer **CUSTOM**

integer **DEBUG**

integer **INFO**

integer **NOTICE**

integer **WARNING**

integer **ERROR**

integer **ALERT**

integer **CRITICAL**

integer **EMERGENCE**

integer **EMERGENCY**

Abstract class `Phalcon\Logger\Adapter`

implements `Phalcon\Logger\AdapterInterface`

Base class for `Phalcon\Logger` adapters

Methods

public **setLogLevel** (*mixed* \$level)

Filters the logs sent to the handlers that are less or equal than a specific level

public **getLogLevel** ()

Returns the current log level

public **setFormatter** (`Phalcon\Logger\FormatterInterface` \$formatter)

Sets the message formatter

public **begin** ()

Starts a transaction

public **commit** ()

Commits the internal transaction

public **rollback** ()

Rollbacks the internal transaction

public **isTransaction** ()

Returns the whether the logger is currently in an active transaction or not

public **critical** (*mixed* \$message, [*array* \$context])

Sends/Writes a critical message to the log

public **emergency** (*mixed* \$message, [*array* \$context])

Sends/Writes an emergency message to the log

public **debug** (*mixed* \$message, [*array* \$context])

Sends/Writes a debug message to the log

public **error** (*mixed* \$message, [*array* \$context])

Sends/Writes an error message to the log

public **info** (*mixed* \$message, [*array* \$context])

Sends/Writes an info message to the log

public **notice** (*mixed* \$message, [*array* \$context])

Sends/Writes a notice message to the log

public **warning** (*mixed* \$message, [*array* \$context])

Sends/Writes a warning message to the log

public **alert** (*mixed* \$message, [*array* \$context])

Sends/Writes an alert message to the log

public **log** (*mixed* \$type, [*mixed* \$message], [*array* \$context])

Logs messages to the internal logger. Appends logs to the logger

abstract public **getFormatter** () inherited from *Phalcon\Logger\AdapterInterface*

...

abstract public **close** () inherited from *Phalcon\Logger\AdapterInterface*

...

Class *Phalcon\Logger\Adapter\File*

extends abstract class *Phalcon\Logger\Adapter*

implements *Phalcon\Logger\AdapterInterface*

Adapter to store logs in plain text files

```
<?php

$logger = new \Phalcon\Logger\Adapter\File("app/logs/test.log");

$logger->log("This is a message");
$logger->log(\Phalcon\Logger::ERROR, "This is an error");
$logger->error("This is another error");

$logger->close();
```

Methods

public **getPath** ()

File Path

public **__construct** (*string* \$name, [*array* \$options])

Phalcon\Logger\Adapter\File constructor

public **getFormatter** ()

Returns the internal formatter

public **logInternal** (*mixed* \$message, *mixed* \$type, *mixed* \$time, *array* \$context)

Writes the log to the file itself

public **close** ()

Closes the logger

public **__wakeup** ()

Opens the internal file handler after unserialization

public **setLogLevel** (*mixed* \$level) inherited from *Phalcon\Logger\Adapter*

Filters the logs sent to the handlers that are less or equal than a specific level

public **getLogLevel** () inherited from *Phalcon\Logger\Adapter*

Returns the current log level

public **setFormatter** (*Phalcon\Logger\FormatterInterface* \$formatter) inherited from *Phalcon\Logger\Adapter*

Sets the message formatter

public **begin** () inherited from *Phalcon\Logger\Adapter*

Starts a transaction

public **commit** () inherited from *Phalcon\Logger\Adapter*

Commits the internal transaction

public **rollback** () inherited from *Phalcon\Logger\Adapter*

Rollbacks the internal transaction

public **isTransaction** () inherited from *Phalcon\Logger\Adapter*

Returns the whether the logger is currently in an active transaction or not

public **critical** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a critical message to the log

public **emergency** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an emergency message to the log

public **debug** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a debug message to the log

public **error** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an error message to the log

public **info** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an info message to the log

public **notice** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a notice message to the log

public **warning** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a warning message to the log

public **alert** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an alert message to the log

public **log** (*mixed* \$type, [*mixed* \$message], [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Logs messages to the internal logger. Appends logs to the logger

Class **Phalcon\Logger\Adapter\Firephp**

extends abstract class *Phalcon\Logger\Adapter*

implements *Phalcon\Logger\AdapterInterface*

Sends logs to FirePHP

```
<?php

use Phalcon\Logger\Adapter\Firephp;
use Phalcon\Logger;

$logger = new Firephp();

$logger->log(Logger::ERROR, "This is an error");
$logger->error("This is another error");
```

Methods

public **getFormatter** ()

Returns the internal formatter

public **logInternal** (*mixed* \$message, *mixed* \$type, *mixed* \$time, *array* \$context)

Writes the log to the stream itself

public **close** ()

Closes the logger

public **setLogLevel** (*mixed* \$level) inherited from *Phalcon\Logger\Adapter*

Filters the logs sent to the handlers that are less or equal than a specific level

public **getLogLevel** () inherited from *Phalcon\Logger\Adapter*

Returns the current log level

public **setFormatter** (*Phalcon\Logger\FormatterInterface* \$formatter) inherited from *Phalcon\Logger\Adapter*

Sets the message formatter

public **begin** () inherited from *Phalcon\Logger\Adapter*

Starts a transaction

public **commit** () inherited from *Phalcon\Logger\Adapter*

Commits the internal transaction

public **rollback** () inherited from *Phalcon\Logger\Adapter*

Rollbacks the internal transaction

public **isTransaction** () inherited from *Phalcon\Logger\Adapter*

Returns the whether the logger is currently in an active transaction or not

public **critical** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a critical message to the log

public **emergency** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an emergency message to the log

public **debug** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a debug message to the log

public **error** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an error message to the log

public **info** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an info message to the log

public **notice** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a notice message to the log

public **warning** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a warning message to the log

public **alert** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an alert message to the log

public **log** (*mixed* \$type, [*mixed* \$message], [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Logs messages to the internal logger. Appends logs to the logger

Class *Phalcon\Logger\Adapter\Stream*

extends abstract class *Phalcon\Logger\Adapter*

implements *Phalcon\Logger\AdapterInterface*

Sends logs to a valid PHP stream

```
<?php

use Phalcon\Logger;
use Phalcon\Logger\Adapter\Stream;

$logger = new Stream("php://stderr");

$logger->log("This is a message");
$logger->log(Logger::ERROR, "This is an error");
$logger->error("This is another error");
```

Methods

public **__construct** (*string* \$name, [*array* \$options])

Phalcon\Logger\Adapter\Stream constructor

public **getFormatter** ()

Returns the internal formatter

public **logInternal** (*mixed* \$message, *mixed* \$type, *mixed* \$time, *array* \$context)

Writes the log to the stream itself

public **close** ()

Closes the logger

public **setLogLevel** (*mixed* \$level) inherited from *Phalcon\Logger\Adapter*

Filters the logs sent to the handlers that are less or equal than a specific level

public **getLogLevel** () inherited from *Phalcon\Logger\Adapter*

Returns the current log level

public **setFormatter** (*Phalcon\Logger\FormatterInterface* \$formatter) inherited from *Phalcon\Logger\Adapter*

Sets the message formatter

public **begin** () inherited from *Phalcon\Logger\Adapter*

Starts a transaction

public **commit** () inherited from *Phalcon\Logger\Adapter*

Commits the internal transaction

public **rollback** () inherited from *Phalcon\Logger\Adapter*

Rollbacks the internal transaction

public **isTransaction** () inherited from *Phalcon\Logger\Adapter*

Returns the whether the logger is currently in an active transaction or not

public **critical** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a critical message to the log

public **emergency** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an emergency message to the log

public **debug** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a debug message to the log

public **error** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an error message to the log

public **info** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an info message to the log

public **notice** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a notice message to the log

public **warning** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a warning message to the log

public **alert** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an alert message to the log

public **log** (*mixed* \$type, [*mixed* \$message], [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Logs messages to the internal logger. Appends logs to the logger

Class *Phalcon\Logger\Adapter\Syslog*

extends abstract class *Phalcon\Logger\Adapter*

implements *Phalcon\Logger\AdapterInterface*

Sends logs to the system logger

```
<?php

use Phalcon\Logger;
use Phalcon\Logger\Adapter\Syslog;

// LOG_USER is the only valid log type under Windows operating systems
$logger = new Syslog(
    "ident",
    [
        "option" => LOG_CONS | LOG_NDELAY | LOG_PID,
        "facility" => LOG_USER,
    ]
);

$logger->log("This is a message");
$logger->log(Logger::ERROR, "This is an error");
$logger->error("This is another error");
```

Methods

public **__construct** (*string* \$name, [*array* \$options])

Phalcon\Logger\Adapter\Syslog constructor

public **getFormatter** ()

Returns the internal formatter

public **logInternal** (*mixed* \$message, *mixed* \$type, *mixed* \$time, *array* \$context)

Writes the log to the stream itself

public **close** ()

Closes the logger

public **setLogLevel** (*mixed* \$level) inherited from *Phalcon\Logger\Adapter*

Filters the logs sent to the handlers that are less or equal than a specific level

public **getLogLevel** () inherited from *Phalcon\Logger\Adapter*

Returns the current log level

public **setFormatter** (*Phalcon\Logger\FormatterInterface* \$formatter) inherited from *Phalcon\Logger\Adapter*

Sets the message formatter

public **begin** () inherited from *Phalcon\Logger\Adapter*

Starts a transaction

public **commit** () inherited from *Phalcon\Logger\Adapter*

Commits the internal transaction

public **rollback** () inherited from *Phalcon\Logger\Adapter*

Rollbacks the internal transaction

public **isTransaction** () inherited from *Phalcon\Logger\Adapter*

Returns the whether the logger is currently in an active transaction or not

public **critical** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a critical message to the log

public **emergency** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an emergency message to the log

public **debug** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a debug message to the log

public **error** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an error message to the log

public **info** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an info message to the log

public **notice** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a notice message to the log

public **warning** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes a warning message to the log

public **alert** (*mixed* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Sends/Writes an alert message to the log

public **log** (*mixed* \$type, [*mixed* \$message], [*array* \$context]) inherited from *Phalcon\Logger\Adapter*

Logs messages to the internal logger. Appends logs to the logger

Class *Phalcon\Logger\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private [Exception](#) **__clone** () inherited from [Exception](#)

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [[Exception](#) \$previous]) inherited from [Exception](#)

Exception constructor

public **__wakeup** () inherited from [Exception](#)

...

final public *string* **getMessage** () inherited from [Exception](#)

Gets the Exception message

final public *int* **getCode** () inherited from [Exception](#)

Gets the Exception code

final public *string* **getFile** () inherited from [Exception](#)

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from [Exception](#)

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Abstract class [Phalcon\Logger\Formatter](#)

implements [Phalcon\Logger\FormatterInterface](#)

This is a base class for logger formatters

Methods

public **getTypeString** (*mixed* \$type)

Returns the string meaning of a logger constant

public **interpolate** (*string* \$message, [*array* \$context])

Interpolates context values into the message placeholders

abstract public **format** (*mixed* \$message, *mixed* \$type, *mixed* \$timestamp, [*mixed* \$context]) inherited from [Phalcon\Logger\FormatterInterface](#)

...

Class `Phalcon\Logger\Formatter\Firephp`

extends abstract class `Phalcon\Logger\Formatter`

implements `Phalcon\Logger\FormatterInterface`

Formats messages so that they can be sent to FirePHP

Methods

public **getTypeString** (*mixed* \$type)

Returns the string meaning of a logger constant

public **setShowBacktrace** ([*mixed* \$isShow])

Returns the string meaning of a logger constant

public **getShowBacktrace** ()

Returns the string meaning of a logger constant

public **enableLabels** ([*mixed* \$isEnabled])

Returns the string meaning of a logger constant

public **labelsEnabled** ()

Returns the labels enabled

public *string* **format** (*string* \$message, *int* \$type, *int* \$timestamp, [*array* \$context])

Applies a format to a message before sending it to the log

public **interpolate** (*string* \$message, [*array* \$context]) inherited from `Phalcon\Logger\Formatter`

Interpolates context values into the message placeholders

Class `Phalcon\Logger\Formatter\Json`

extends abstract class `Phalcon\Logger\Formatter`

implements `Phalcon\Logger\FormatterInterface`

Formats messages using JSON encoding

Methods

public *string* **format** (*string* \$message, *int* \$type, *int* \$timestamp, [*array* \$context])

Applies a format to a message before sent it to the internal log

public **getTypeString** (*mixed* \$type) inherited from `Phalcon\Logger\Formatter`

Returns the string meaning of a logger constant

public **interpolate** (*string* \$message, [*array* \$context]) inherited from `Phalcon\Logger\Formatter`

Interpolates context values into the message placeholders

Class `Phalcon\Logger\Formatter\Line`

extends abstract class *Phalcon\Logger\Formatter*

implements *Phalcon\Logger\FormatterInterface*

Formats messages using an one-line string

Methods

public **getDateFormat** ()

Default date format

public **setDateFormat** (*mixed* \$dateFormat)

Default date format

public **getFormat** ()

Format applied to each message

public **setFormat** (*mixed* \$format)

Format applied to each message

public **__construct** ([*string* \$format], [*string* \$dateFormat])

`Phalcon\Logger\Formatter\Line` construct

public *string* **format** (*string* \$message, *int* \$type, *int* \$timestamp, [*array* \$context])

Applies a format to a message before sent it to the internal log

public **getTypeString** (*mixed* \$type) inherited from *Phalcon\Logger\Formatter*

Returns the string meaning of a logger constant

public **interpolate** (*string* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Formatter*

Interpolates context values into the message placeholders

Class `Phalcon\Logger\Formatter\Syslog`

extends abstract class *Phalcon\Logger\Formatter*

implements *Phalcon\Logger\FormatterInterface*

Prepares a message to be used in a Syslog backend

Methods

public *array* **format** (*string* \$message, *int* \$type, *int* \$timestamp, [*array* \$context])

Applies a format to a message before sent it to the internal log

public **getTypeString** (*mixed* \$type) inherited from *Phalcon\Logger\Formatter*

Returns the string meaning of a logger constant

public **interpolate** (*string* \$message, [*array* \$context]) inherited from *Phalcon\Logger\Formatter*

Interpolates context values into the message placeholders

Class Phalcon\Logger\Item

Represents each item in a logging transaction

Methods

public **getType** ()

Log type

public **getMessage** ()

Log message

public **getTime** ()

Log timestamp

public **getContext** ()

...

public **__construct** (*string* \$message, *integer* \$type, [*integer* \$time], [*array* \$context])

Phalcon\Logger\Item constructor

Class Phalcon\Logger\Multiple

Handles multiples logger handlers

Methods

public **getLoggers** ()

...

public **getFormatter** ()

...

public **getLogLevel** ()

...

public **push** (*Phalcon\Logger\AdapterInterface* \$logger)

Pushes a logger to the logger tail

public **setFormatter** (*Phalcon\Logger\FormatterInterface* \$formatter)

Sets a global formatter

public **setLogLevel** (*mixed* \$level)

Sets a global level

public **log** (*mixed* \$type, [*mixed* \$message], [*array* \$context])

Sends a message to each registered logger

public **critical** (*mixed* \$message, [*array* \$context])

Sends/Writes an critical message to the log

public **emergency** (*mixed* \$message, [*array* \$context])

Sends/Writes an emergency message to the log

public **debug** (*mixed* \$message, [*array* \$context])

Sends/Writes a debug message to the log

public **error** (*mixed* \$message, [*array* \$context])

Sends/Writes an error message to the log

public **info** (*mixed* \$message, [*array* \$context])

Sends/Writes an info message to the log

public **notice** (*mixed* \$message, [*array* \$context])

Sends/Writes a notice message to the log

public **warning** (*mixed* \$message, [*array* \$context])

Sends/Writes a warning message to the log

public **alert** (*mixed* \$message, [*array* \$context])

Sends/Writes an alert message to the log

Class Phalcon\Mvc\Application

extends abstract class *Phalcon\Application*

implements *Phalcon\Di\InjectionAwareInterface*, *Phalcon\Events\EventsAwareInterface*

This component encapsulates all the complex operations behind instantiating every component needed and integrating it with the rest to allow the MVC pattern to operate as desired.

```
<?php

use Phalcon\Mvc\Application;

class MyApp extends Application
{
    /**
     * Register the services here to make them general or register
     * in the ModuleDefinition to make them module-specific
     */
    protected function registerServices()
    {

    }

    /**
     * This method registers all the modules in the application
     */
    public function main()
    {
        $this->registerModules(
            [
                "frontend" => [
                    "className" => "Multiple\\Frontend\\Module",
                    "path"       => "../apps/frontend/Module.php",
                ],
            ],
        );
    }
}
```

```
        "backend" => [
            "className" => "Multiple\\Backend\\Module",
            "path"      => "../apps/backend/Module.php",
        ],
    ],
);
}
}

$application = new MyApp();

$application->main();
```

Methods

public **useImplicitView** (*mixed* \$implicitView)

By default. The view is implicitly buffering all the output You can full disable the view component using this method

public **handle** (*mixed* \$uri)

Handles a MVC request

public **__construct** ([*Phalcon\DiInterface* \$dependencyInjector]) inherited from *Phalcon\Application*

Phalcon\Application

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Application*

Sets the events manager

public **getEventManager** () inherited from *Phalcon\Application*

Returns the internal event manager

public **registerModules** (array \$modules, [*mixed* \$merge]) inherited from *Phalcon\Application*

Register an array of modules present in the application

```
<?php

$this->registerModules(
[
    "frontend" => [
        "className" => "Multiple\\Frontend\\Module",
        "path"      => "../apps/frontend/Module.php",
    ],
    "backend" => [
        "className" => "Multiple\\Backend\\Module",
        "path"      => "../apps/backend/Module.php",
    ],
],
);
```

public **getModules** () inherited from *Phalcon\Application*

Return the modules registered in the application

public **getModule** (*mixed* \$name) inherited from *Phalcon\Application*

Gets the module definition registered in the application via module name

public **setDefaultModule** (*mixed* \$defaultModule) inherited from *Phalcon\Application*

Sets the module name to be used if the router doesn't return a valid module

public **getDefaultModule** () inherited from *Phalcon\Application*

Returns the default module name

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di\Injectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method __get

Class *Phalcon\Mvc\Application\Exception*

extends class *Phalcon\Application\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Abstract class **Phalcon\Mvc\Collection**

implements [Phalcon\Mvc\EntityInterface](#), [Phalcon\Mvc\CollectionInterface](#), [Phalcon\Di\InjectionAwareInterface](#), [Serializable](#)

This component implements a high level abstraction for NoSQL databases which works with documents

Constants

integer **OP_NONE**

integer **OP_CREATE**

integer **OP_UPDATE**

integer **OP_DELETE**

integer **DIRTY_STATE_PERSISTENT**

integer **DIRTY_STATE_TRANSIENT**

integer **DIRTY_STATE_DETACHED**

Methods

final public **__construct** ([Phalcon\DiInterface](#) \$dependencyInjector], [[Phalcon\Mvc\Collection\ManagerInterface](#) \$modelsManager])

Phalcon\Mvc\Collection constructor

public **setId** (*mixed* \$id)

Sets a value for the `_id` property, creates a `MongoId` object if needed

public *MongoId* **getId** ()

Returns the value of the `_id` property

public **setDI** ([Phalcon\DiInterface](#) \$dependencyInjector)

Sets the dependency injection container

public **getDI** ()

Returns the dependency injection container

protected **setEventManager** ([Phalcon\Mvc\Collection\ManagerInterface](#) \$eventsManager)

Sets a custom events manager

protected **getEventManager** ()

Returns the custom events manager

public **getCollectionManager** ()

Returns the models manager related to the entity instance

public **getReservedAttributes** ()

Returns an array with reserved properties that cannot be part of the insert/update

protected **useImplicitObjectIds** (*mixed* \$useImplicitObjectIds)

Sets if a model must use implicit objects ids

protected **setSource** (*mixed* \$source)

Sets collection name which model should be mapped

public **getSource** ()

Returns collection name mapped in the model

public **setConnectionService** (*mixed* \$connectionService)

Sets the DependencyInjection connection service name

public **getConnectionService** ()

Returns DependencyInjection connection service

public *MongoDb* **getConnection** ()

Retrieves a database connection

public *mixed* **readAttribute** (*string* \$attribute)

Reads an attribute value by its name

```
<?php
echo $robot->readAttribute ("name");
```

public **writeAttribute** (*string* \$attribute, *mixed* \$value)

Writes an attribute value by its name

```
<?php
$robot->writeAttribute ("name", "Rosey");
```

public static **cloneResult** (*Phalcon\Mvc\CollectionInterface* \$collection, *array* \$document)

Returns a cloned collection

protected static *array* **_getResultset** (*array* \$params, *Phalcon\Mvc\Collection* \$collection, *MongoDb* \$connection, *boolean* \$unique)

Returns a collection resultset

protected static *int* **_getGroupResultset** (*array* \$params, *Phalcon\Mvc\Collection* \$collection, *MongoDb* \$connection)

Perform a count over a resultset

final protected *boolean* **_preSave** (*Phalcon\DiInterface* \$dependencyInjector, *boolean* \$disableEvents, *boolean* \$exists)

Executes internal hooks before save a document

final protected **_postSave** (*mixed* \$disableEvents, *mixed* \$success, *mixed* \$exists)

Executes internal events after save a document

protected **validate** (*mixed* \$validator)

Executes validators on every validation call

```
<?php

use Phalcon\Mvc\Model\Validator\ExclusionIn as ExclusionIn;

class Subscriptors extends \Phalcon\Mvc\Collection
{
    public function validation()
    {
        // Old, deprecated syntax, use new one below
        $this->validate(
            new ExclusionIn(
                [
                    "field" => "status",
                    "domain" => ["A", "I"],
                ]
            )
        );

        if ($this->validationHasFailed() == true) {
            return false;
        }
    }
}
```

```
<?php

use Phalcon\Validation\Validator\ExclusionIn as ExclusionIn;
use Phalcon\Validation;

class Subscriptors extends \Phalcon\Mvc\Collection
{
    public function validation()
    {
        $validator = new Validation();
        $validator->add("status",
            new ExclusionIn(
                [
                    "domain" => ["A", "I"]
                ]
            )
        );

        return $this->validate($validator);
    }
}
```

public validationHasFailed ()

Check whether validation process has generated any messages

```
<?php

use Phalcon\Mvc\Model\Validator\ExclusionIn as ExclusionIn;

class Subscriptors extends \Phalcon\Mvc\Collection
{
    public function validation()
    {
```

```

        $this->validate(
            new ExclusionIn(
                [
                    "field" => "status",
                    "domain" => ["A", "I"],
                ]
            )
        );

        if ($this->validationHasFailed() == true) {
            return false;
        }
    }
}

```

public **fireEvent** (*mixed* \$eventName)

Fires an internal event

public **fireEventCancel** (*mixed* \$eventName)

Fires an internal event that cancels the operation

protected **_cancelOperation** (*mixed* \$disableEvents)

Cancel the current operation

protected *boolean* **_exists** (*MongoCollection* \$collection)

Checks if the document exists in the collection

public **getMessages** ()

Returns all the validation messages

```

<?php

$robot = new Robots();

$robot->type = "mechanical";
$robot->name = "Astro Boy";
$robot->year = 1952;

if ($robot->save() === false) {
    echo "Umh, We can't store robots right now ";

    $messages = $robot->getMessages();

    foreach ($messages as $message) {
        echo $message;
    }
} else {
    echo "Great, a new robot was saved successfully!";
}

```

public **appendMessage** (*Phalcon\Mvc\Model\MessageInterface* \$message)

Appends a customized message on the validation process

```

<?php

```

```
use \Phalcon\Mvc\Model\Message as Message;

class Robots extends \Phalcon\Mvc\Model
{
    public function beforeSave()
    {
        if ($this->name === "Peter") {
            $message = new Message(
                "Sorry, but a robot cannot be named Peter"
            );

            $this->appendMessage($message);
        }
    }
}
```

protected **prepareCU** ()

Shared Code for CU Operations Prepares Collection

public **save** ()

Creates/Updates a collection based on the values in the attributes

public **create** ()

Creates a collection based on the values in the attributes

public **createIfNotExist** (array \$criteria)

Creates a document based on the values in the attributes, if not found by criteria Preferred way to avoid duplication is to create index on attribute

```
<?php

$robot = new Robot();

$robot->name = "MyRobot";
$robot->type = "Droid";

// Create only if robot with same name and type does not exist
$robot->createIfNotExist(
    [
        "name",
        "type",
    ]
);
```

public **update** ()

Creates/Updates a collection based on the values in the attributes

public static **findById** (mixed \$id)

Find a document by its id (_id)

```
<?php

// Find user by using \MongoId object
$user = Users::findById(
    new \MongoId("545eb081631d16153a293a66")
);
```



```
);

// Find user by using id as sting
$user = Users::findById("45cbc4a0e4123f6920000002");

// Validate input
if ($user = Users::findById($_POST["id"])) {
    // ...
}
```

public static **findFirst** ([array \$parameters])

Allows to query the first record that match the specified conditions

```
<?php

// What's the first robot in the robots table?
$robot = Robots::findFirst();

echo "The robot name is ", $robot->name, "\n";

// What's the first mechanical robot in robots table?
$robot = Robots::findFirst(
    [
        [
            "type" => "mechanical",
        ]
    ]
);

echo "The first mechanical robot name is ", $robot->name, "\n";

// Get first virtual robot ordered by name
$robot = Robots::findFirst(
    [
        [
            "type" => "mechanical",
        ],
        "order" => [
            "name" => 1,
        ],
    ]
);

echo "The first virtual robot name is ", $robot->name, "\n";

// Get first robot by id (_id)
$robot = Robots::findFirst(
    [
        [
            "_id" => new \MongoId("45cbc4a0e4123f6920000002"),
        ]
    ]
);

echo "The robot id is ", $robot->_id, "\n";
```

public static **find** ([array \$parameters])

Allows to query a set of records that match the specified conditions

```
<?php

// How many robots are there?
$robots = Robots::find();

echo "There are ", count($robots), "\n";

// How many mechanical robots are there?
$robots = Robots::find(
    [
        [
            "type" => "mechanical",
        ]
    ]
);

echo "There are ", count($robots), "\n";

// Get and print virtual robots ordered by name
$robots = Robots::findFirst(
    [
        [
            "type" => "virtual"
        ],
        "order" => [
            "name" => 1,
        ]
    ]
);

foreach ($robots as $robot) {
    echo $robot->name, "\n";
}

// Get first 100 virtual robots ordered by name
$robots = Robots::find(
    [
        [
            "type" => "virtual",
        ],
        "order" => [
            "name" => 1,
        ],
        "limit" => 100,
    ]
);

foreach ($robots as $robot) {
    echo $robot->name, "\n";
}
```

public static **count** ([array \$parameters])

Perform a count over a collection

```
<?php
```

```
echo "There are ", Robots::count(), " robots";
```

public static **aggregate** ([array \$parameters])

Perform an aggregation using the Mongo aggregation framework

public static **summatory** (*mixed* \$field, [*mixed* \$conditions], [*mixed* \$finalize])

Allows to perform a summatory group for a column in the collection

public **delete** ()

Deletes a model instance. Returning true on success or false otherwise.

```
<?php

$robot = Robots::findFirst();

$robot->delete();

$robots = Robots::find();

foreach ($robots as $robot) {
    $robot->delete();
}
```

public **setDirtyState** (*mixed* \$dirtyState)

Sets the dirty state of the object using one of the DIRTY_STATE_* constants

public **getDirtyState** ()

Returns one of the DIRTY_STATE_* constants telling if the document exists in the collection or not

protected **addBehavior** (*Phalcon\Mvc\Collection\BehaviorInterface* \$behavior)

Sets up a behavior in a collection

public **skipOperation** (*mixed* \$skip)

Skips the current operation forcing a success state

public **toArray** ()

Returns the instance as an array representation

```
<?php

print_r(
    $robot->toArray()
);
```

public **serialize** ()

Serializes the object ignoring connections or protected properties

public **unserialize** (*mixed* \$data)

Unserializes the object from a serialized string

Abstract class **Phalcon\Mvc\Collection\Behavior**

implements *Phalcon\Mvc\Collection\BehaviorInterface*

This is an optional base class for ORM behaviors

Methods

public **__construct** ([array \$options])

protected **mustTakeAction** (mixed \$eventName)

Checks whether the behavior must take action on certain event

protected array **getOptions** ([string \$eventName])

Returns the behavior options related to an event

public **notify** (mixed \$type, *Phalcon\Mvc\CollectionInterface* \$model)

This method receives the notifications from the EventsManager

public **missingMethod** (*Phalcon\Mvc\CollectionInterface* \$model, mixed \$method, [mixed \$arguments])

Acts as fallbacks when a missing method is called on the collection

Class *Phalcon\Mvc\Collection\Behavior\SoftDelete*

extends abstract class *Phalcon\Mvc\Collection\Behavior*

implements *Phalcon\Mvc\Collection\BehaviorInterface*

Instead of permanently delete a record it marks the record as deleted changing the value of a flag column

Methods

public **notify** (mixed \$type, *Phalcon\Mvc\CollectionInterface* \$model)

Listens for notifications from the models manager

public **__construct** ([array \$options]) inherited from *Phalcon\Mvc\Collection\Behavior*

Phalcon\Mvc\Collection\Behavior

protected **mustTakeAction** (mixed \$eventName) inherited from *Phalcon\Mvc\Collection\Behavior*

Checks whether the behavior must take action on certain event

protected array **getOptions** ([string \$eventName]) inherited from *Phalcon\Mvc\Collection\Behavior*

Returns the behavior options related to an event

public **missingMethod** (*Phalcon\Mvc\CollectionInterface* \$model, mixed \$method, [mixed \$arguments]) inherited from *Phalcon\Mvc\Collection\Behavior*

Acts as fallbacks when a missing method is called on the collection

Class *Phalcon\Mvc\Collection\Behavior\Timestampable*

extends abstract class *Phalcon\Mvc\Collection\Behavior*

implements *Phalcon\Mvc\Collection\BehaviorInterface*

Allows to automatically update a model's attribute saving the datetime when a record is created or updated

Methods

public **notify** (*mixed* \$type, *Phalcon\Mvc\CollectionInterface* \$model)

Listens for notifications from the models manager

public **__construct** ([*array* \$options]) inherited from *Phalcon\Mvc\Collection\Behavior*

Phalcon\Mvc\Collection\Behavior

protected **mustTakeAction** (*mixed* \$eventName) inherited from *Phalcon\Mvc\Collection\Behavior*

Checks whether the behavior must take action on certain event

protected *array* **getOptions** ([*string* \$eventName]) inherited from *Phalcon\Mvc\Collection\Behavior*

Returns the behavior options related to an event

public **missingMethod** (*Phalcon\Mvc\CollectionInterface* \$model, *mixed* \$method, [*mixed* \$arguments]) inherited from *Phalcon\Mvc\Collection\Behavior*

Acts as fallbacks when a missing method is called on the collection

Class *Phalcon\Mvc\Collection\Document*

implements *Phalcon\Mvc\EntityInterface*, *ArrayAccess*

This component allows *Phalcon\Mvc\Collection* to return rows without an associated entity. This objects implements the *ArrayAccess* interface to allow access the object as `object->x` or `array[x]`.

Methods

public *boolean* **offsetExists** (*int* \$index)

Checks whether an offset exists in the document

public **offsetGet** (*mixed* \$index)

Returns the value of a field using the *ArrayAccess* interface

public **offsetSet** (*mixed* \$index, *mixed* \$value)

Change a value using the *ArrayAccess* interface

public **offsetUnset** (*string* \$offset)

Rows cannot be changed. It has only been implemented to meet the definition of the *ArrayAccess* interface

public *mixed* **readAttribute** (*string* \$attribute)

Reads an attribute value by its name

```
<?php
echo $robot->readAttribute("name");
```

public **writeAttribute** (*string* \$attribute, *mixed* \$value)

Writes an attribute value by its name

```
<?php
$robot->writeAttribute("name", "Rosey");
```

public array **toArray** ()

Returns the instance as an array representation

Class `Phalcon\Mvc\Collection\Exception`

extends class `Phalcon\Exception`

implements `Throwable`

Methods

final private `Exception` **__clone** () inherited from `Exception`

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [`Exception` \$previous]) inherited from `Exception`

Exception constructor

public **__wakeup** () inherited from `Exception`

...

final public *string* **getMessage** () inherited from `Exception`

Gets the Exception message

final public *int* **getCode** () inherited from `Exception`

Gets the Exception code

final public *string* **getFile** () inherited from `Exception`

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from `Exception`

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from `Exception`

Gets the stack trace

final public `Exception` **getPrevious** () inherited from `Exception`

Returns previous Exception

final public `Exception` **getTraceAsString** () inherited from `Exception`

Gets the stack trace as a string

public *string* **__toString** () inherited from `Exception`

String representation of the exception

Class Phalcon\Mvc\Collection\Manager

implements *Phalcon\Di\InjectionAwareInterface*, *Phalcon\Events\EventsAwareInterface*

This components controls the initialization of models, keeping record of relations between the different models of the application.

A CollectionManager is injected to a model via a Dependency Injector Container such as Phalcon\Di.

```

<?php

$di = new \Phalcon\Di();

$di->set(
    "collectionManager",
    function () {
        return new \Phalcon\Mvc\Collection\Manager();
    }
);

$robot = new Robots($di);

```

Methods

public **getServiceName** ()

...

public **setServiceName** (*mixed* \$serviceName)

...

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the DependencyInjector container

public **getDI** ()

Returns the DependencyInjector container

public **setEventsManager** (*Phalcon\Events\ManagerInterface* \$eventsManager)

Sets the event manager

public **getEventsManager** ()

Returns the internal event manager

public **setCustomEventsManager** (*Phalcon\Mvc\CollectionInterface* \$model, *Phalcon\Events\ManagerInterface* \$eventsManager)

Sets a custom events manager for a specific model

public **getCustomEventsManager** (*Phalcon\Mvc\CollectionInterface* \$model)

Returns a custom events manager related to a model

public **initialize** (*Phalcon\Mvc\CollectionInterface* \$model)

Initializes a model in the models manager

public **isInitialized** (*mixed* \$modelName)

Check whether a model is already initialized

public **getLastInitialized** ()

Get the latest initialized model

public **setConnectionService** (*Phalcon\Mvc\CollectionInterface* \$model, *mixed* \$connectionService)

Sets a connection service for a specific model

public **getConnectionService** (*Phalcon\Mvc\CollectionInterface* \$model)

Gets a connection service for a specific model

public **useImplicitObjectIds** (*Phalcon\Mvc\CollectionInterface* \$model, *mixed* \$useImplicitObjectIds)

Sets whether a model must use implicit objects ids

public **isUsingImplicitObjectIds** (*Phalcon\Mvc\CollectionInterface* \$model)

Checks if a model is using implicit object ids

public *Mongo* **getConnection** (*Phalcon\Mvc\CollectionInterface* \$model)

Returns the connection related to a model

public **notifyEvent** (*mixed* \$eventName, *Phalcon\Mvc\CollectionInterface* \$model)

Receives events generated in the models and dispatches them to an events-manager if available Notify the behaviors that are listening in the model

public **missingMethod** (*Phalcon\Mvc\CollectionInterface* \$model, *mixed* \$eventName, *mixed* \$data)

Dispatch an event to the listeners and behaviors This method expects that the endpoint listeners/behaviors returns true meaning that at least one was implemented

public **addBehavior** (*Phalcon\Mvc\CollectionInterface* \$model, *Phalcon\Mvc\Collection\BehaviorInterface* \$behavior)

Binds a behavior to a model

Abstract class **Phalcon\Mvc\Controller**

extends abstract class *Phalcon\Di\Injectable*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\Di\InjectionAwareInterface*, *Phalcon\Mvc\ControllerInterface*

Every application controller should extend this class that encapsulates all the controller functionality

The controllers provide the “flow” between models and views. Controllers are responsible for processing the incoming requests from the web browser, interrogating the models for data, and passing that data on to the views for presentation.

```
<?php
<?php

class PeopleController extends \Phalcon\Mvc\Controller
{
    // This action will be executed by default
    public function indexAction()
    {

    }

    public function findAction()
```



```

{

}

public function saveAction()
{
    // Forwards flow to the index action
    return $this->dispatcher->forward(
        [
            "controller" => "people",
            "action"      => "index",
        ]
    );
}
}

```

Methods

final public **__construct** ()

Phalcon\Mvc\Controller constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\DiInjectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\DiInjectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\DiInjectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\DiInjectable*

Returns the internal event manager

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\DiInjectable*

Magic method __get

Class Phalcon\Mvc\Dispatcher

extends abstract class *Phalcon\Dispatcher*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\Di\InjectionAwareInterface*, *Phalcon\DispatcherInterface*, *Phalcon\Mvc\DispatcherInterface*

Dispatching is the process of taking the request object, extracting the module name, controller name, action name, and optional parameters contained in it, and then instantiating a controller and calling an action of that controller.

```

<?php

$di = new \Phalcon\Di();

$dispatcher = new \Phalcon\Mvc\Dispatcher();

```

```
$dispatcher->setDI ($di);

$dispatcher->setControllerName ("posts");
$dispatcher->setActionName ("index");
$dispatcher->setParams ( [] );

$controller = $dispatcher->dispatch ();
```

Constants

integer **EXCEPTION_NO_DI**

integer **EXCEPTION_CYCLIC_ROUTING**

integer **EXCEPTION_HANDLER_NOT_FOUND**

integer **EXCEPTION_INVALID_HANDLER**

integer **EXCEPTION_INVALID_PARAMS**

integer **EXCEPTION_ACTION_NOT_FOUND**

Methods

public **setControllerSuffix** (*mixed* \$controllerSuffix)

Sets the default controller suffix

public **setDefaultController** (*mixed* \$controllerName)

Sets the default controller name

public **setControllerName** (*mixed* \$controllerName)

Sets the controller name to be dispatched

public **getControllerName** ()

Gets last dispatched controller name

public **getPreviousNamespaceName** ()

Gets previous dispatched namespace name

public **getPreviousControllerName** ()

Gets previous dispatched controller name

public **getPreviousActionName** ()

Gets previous dispatched action name

protected **_throwDispatchException** (*mixed* \$message, [*mixed* \$exceptionCode])

Throws an internal exception

protected **_handleException** ([Exception](#) \$exception)

Handles a user exception

public **getControllerClass** ()

Possible controller class name that will be located to dispatch the request

public **getLastController** ()

Returns the latest dispatched controller

public **getActiveController** ()

Returns the active controller in the dispatcher

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Dispatcher*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Dispatcher*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Dispatcher*

Sets the events manager

public **getEventManager** () inherited from *Phalcon\Dispatcher*

Returns the internal event manager

public **setActionSuffix** (*mixed* \$actionSuffix) inherited from *Phalcon\Dispatcher*

Sets the default action suffix

public **getActionSuffix** () inherited from *Phalcon\Dispatcher*

Gets the default action suffix

public **setModuleName** (*mixed* \$moduleName) inherited from *Phalcon\Dispatcher*

Sets the module where the controller is (only informative)

public **getModuleName** () inherited from *Phalcon\Dispatcher*

Gets the module where the controller class is

public **setNamespaceName** (*mixed* \$namespaceName) inherited from *Phalcon\Dispatcher*

Sets the namespace where the controller class is

public **getNamespaceName** () inherited from *Phalcon\Dispatcher*

Gets a namespace to be prepended to the current handler name

public **setDefaultNamespace** (*mixed* \$namespaceName) inherited from *Phalcon\Dispatcher*

Sets the default namespace

public **getDefaultNamespace** () inherited from *Phalcon\Dispatcher*

Returns the default namespace

public **setDefaultAction** (*mixed* \$actionName) inherited from *Phalcon\Dispatcher*

Sets the default action name

public **setActionName** (*mixed* \$actionName) inherited from *Phalcon\Dispatcher*

Sets the action name to be dispatched

public **getActionName** () inherited from *Phalcon\Dispatcher*

Gets the latest dispatched action name

public **setParams** (*array* \$params) inherited from *Phalcon\Dispatcher*

Sets action params to be dispatched

public **getParams** () inherited from *Phalcon\Dispatcher*

Gets action params

public **setParam** (*mixed* \$param, *mixed* \$value) inherited from *Phalcon\Dispatcher*

Set a param by its name or numeric index

public *mixed* **getParam** (*mixed* \$param, [*string* | *array* \$filters], [*mixed* \$defaultValue]) inherited from *Phalcon\Dispatcher*

Gets a param by its name or numeric index

public *boolean* **hasParam** (*mixed* \$param) inherited from *Phalcon\Dispatcher*

Check if a param exists

public **getActiveMethod** () inherited from *Phalcon\Dispatcher*

Returns the current method to be/executed in the dispatcher

public **isFinished** () inherited from *Phalcon\Dispatcher*

Checks if the dispatch loop is finished or has more pendent controllers/tasks to dispatch

public **setReturnedValue** (*mixed* \$value) inherited from *Phalcon\Dispatcher*

Sets the latest returned value by an action manually

public *mixed* **getReturnedValue** () inherited from *Phalcon\Dispatcher*

Returns value returned by the latest dispatched action

public **setModelBinding** (*mixed* \$value, [*mixed* \$cache]) inherited from *Phalcon\Dispatcher*

Enable/Disable model binding during dispatch

```
<?php
$di->set('dispatcher', function() {
    $dispatcher = new Dispatcher();

    $dispatcher->setModelBinding(true, 'cache');
    return $dispatcher;
});
```

public **setModelBinder** (Phalcon\Mvc\Model\BinderInterface \$modelBinder, [*mixed* \$cache]) inherited from *Phalcon\Dispatcher*

Enable model binding during dispatch

```
<?php
$di->set('dispatcher', function() {
    $dispatcher = new Dispatcher();

    $dispatcher->setModelBinder(new Binder(), 'cache');
    return $dispatcher;
});
```

public **getModelBinder** () inherited from *Phalcon\Dispatcher*

Gets model binder

public *object* **dispatch** () inherited from *Phalcon\Dispatcher*

Dispatches a handle action taking into account the routing parameters

protected *object* **_dispatch** () inherited from *Phalcon\Dispatcher*

Dispatches a handle action taking into account the routing parameters

public **forward** (*array* \$forward) inherited from *Phalcon\Dispatcher*

Forwards the execution flow to another controller/action Dispatchers are unique per module. Forwarding between modules is not allowed

```
<?php

$this->dispatcher->forward(
    [
        "controller" => "posts",
        "action"      => "index",
    ]
);
```

public **wasForwarded** () inherited from *Phalcon\Dispatcher*

Check if the current executed action was forwarded by another one

public **getHandlerClass** () inherited from *Phalcon\Dispatcher*

Possible class name that will be located to dispatch the request

public **callActionMethod** (*mixed* \$handler, *mixed* \$actionMethod, [*array* \$params]) inherited from *Phalcon\Dispatcher*

...

public **getBoundModels** () inherited from *Phalcon\Dispatcher*

Returns bound models from binder instance

```
<?php

class UserController extends Controller
{
    public function showAction(User $user)
    {
        $boundModels = $this->dispatcher->getBoundModels(); // return array with $user
    }
}
```

protected **_resolveEmptyProperties** () inherited from *Phalcon\Dispatcher*

Set empty properties to their defaults (where defaults are available)

Class *Phalcon\Mvc\Dispatcher\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Class Phalcon\Mvc\Micro

extends abstract class *Phalcon\DI\Injectable*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\DI\InjectionAwareInterface*, *ArrayAccess*

With Phalcon you can create “Micro-Framework like” applications. By doing this, you only need to write a minimal amount of code to create a PHP application. Micro applications are suitable to small applications, APIs and prototypes in a practical way.

```
<?php

$app = new \Phalcon\Mvc\Micro();

$app->get (
    "/say/welcome/{name}",
    function ($name) {
        echo "<h1>Welcome $name!</h1>";
    }
);

$app->handle();
```

Methods

public **__construct** ([*Phalcon\DiInterface* \$dependencyInjector])

Phalcon\Mvc\Micro constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the DependencyInjector container

public *Phalcon\Mvc\Router\RouteInterface* **map** (*string* \$routePattern, *callable* \$handler)

Maps a route to a handler without any HTTP method constraint

public *Phalcon\Mvc\Router\RouteInterface* **get** (*string* \$routePattern, *callable* \$handler)

Maps a route to a handler that only matches if the HTTP method is GET

public *Phalcon\Mvc\Router\RouteInterface* **post** (*string* \$routePattern, *callable* \$handler)

Maps a route to a handler that only matches if the HTTP method is POST

public *Phalcon\Mvc\Router\RouteInterface* **put** (*string* \$routePattern, *callable* \$handler)

Maps a route to a handler that only matches if the HTTP method is PUT

public *Phalcon\Mvc\Router\RouteInterface* **patch** (*string* \$routePattern, *callable* \$handler)

Maps a route to a handler that only matches if the HTTP method is PATCH

public *Phalcon\Mvc\Router\RouteInterface* **head** (*string* \$routePattern, *callable* \$handler)

Maps a route to a handler that only matches if the HTTP method is HEAD

public *Phalcon\Mvc\Router\RouteInterface* **delete** (*string* \$routePattern, *callable* \$handler)

Maps a route to a handler that only matches if the HTTP method is DELETE

public *Phalcon\Mvc\Router\RouteInterface* **options** (*string* \$routePattern, *callable* \$handler)

Maps a route to a handler that only matches if the HTTP method is OPTIONS

public **mount** (*Phalcon\Mvc\Micro\CollectionInterface* \$collection)

Mounts a collection of handlers

public *Phalcon\Mvc\Micro* **notFound** (*callable* \$handler)

Sets a handler that will be called when the router doesn't match any of the defined routes

public *Phalcon\Mvc\Micro* **error** (*callable* \$handler)

Sets a handler that will be called when an exception is thrown handling the route

public **getRouter** ()

Returns the internal router used by the application

public *Phalcon\Di\ServiceInterface* **setService** (*string* \$serviceName, *mixed* \$definition, [*boolean* \$shared])

Sets a service from the DI

public **hasService** (*mixed* \$serviceName)

Checks if a service is registered in the DI

public *object* **getService** (*string* \$serviceName)

Obtains a service from the DI

public *mixed* **getSharedService** (*string* \$serviceName)

Obtains a shared service from the DI

public *mixed* **handle** ([*string* \$uri])

Handle the whole request

public **stop** ()

Stops the middleware execution avoiding than other middlewares be executed

public **setActiveHandler** (*callable* \$activeHandler)

Sets externally the handler that must be called by the matched route

public *callable* **getActiveHandler** ()

Return the handler that will be called for the matched route

public *mixed* **getReturnedValue** ()

Returns the value returned by the executed handler

public *boolean* **offsetExists** (*string* \$alias)

Check if a service is registered in the internal services container using the array syntax

public **offsetSet** (*string* \$alias, *mixed* \$definition)

Allows to register a shared service in the internal services container using the array syntax

```
<?php
$app["request"] = new \Phalcon\Http\Request();
```

public *mixed* **offsetGet** (*string* \$alias)

Allows to obtain a shared service in the internal services container using the array syntax

```
<?php
var_dump(
    $app["request"]
);
```

public **offsetUnset** (*string* \$alias)

Removes a service from the internal services container using the array syntax

public *Phalcon\Mvc\Micro* **before** (*callable* \$handler)

Appends a before middleware to be called before execute the route

public *Phalcon\Mvc\Micro* **afterBinding** (*callable* \$handler)

Appends a afterBinding middleware to be called after model binding

public *Phalcon\Mvc\Micro* **after** (*callable* \$handler)

Appends an ‘after’ middleware to be called after execute the route

public *Phalcon\Mvc\Micro* **finish** (*callable* \$handler)

Appends a ‘finish’ middleware to be called when the request is finished

public **getHandlers** ()

Returns the internal handlers attached to the application

public **getModelBinder** ()

Gets model binder

public **setModelBinder** (Phalcon\Mvc\Model\BinderInterface \$modelBinder, [*mixed* \$cache])

Sets model binder

```
<?php
$micro = new Micro($di);
$micro->setModelBinder(new Binder(), 'cache');
```

public **getBoundModels** ()

Returns bound models from binder instance

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **setEventManager** (Phalcon\Events\ManagerInterface \$eventsManager) inherited from *Phalcon\Di\Injectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Di\Injectable*

Returns the internal event manager

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method __get

Class Phalcon\Mvc\Micro\Collection

implements *Phalcon\Mvc\Micro\CollectionInterface*

Groups Micro-Mvc handlers as controllers

```
<?php
$app = new \Phalcon\Mvc\Micro();

$collection = new Collection();

$collection->setHandler(
    new PostsController()
);

$collection->get("/posts/edit/{id}", "edit");

$app->mount($collection);
```

Methods

protected **_addMap** (*string* | *array* \$method, *string* \$routePattern, *mixed* \$handler, *string* \$name)

Internal function to add a handler to the group

public **setPrefix** (*mixed* \$prefix)

Sets a prefix for all routes added to the collection

public **getPrefix** ()

Returns the collection prefix if any

public *array* **getHandlers** ()

Returns the registered handlers

public *Phalcon\Mvc\Micro\Collection* **setHandler** (*mixed* \$handler, [*boolean* \$lazy])

Sets the main handler

public **setLazy** (*mixed* \$lazy)

Sets if the main handler must be lazy loaded

public **isLazy** ()

Returns if the main handler must be lazy loaded

public *mixed* **getHandler** ()

Returns the main handler

public *Phalcon\Mvc\Micro\Collection* **map** (*string* \$routePattern, *callable* \$handler, [*string* \$name])

Maps a route to a handler

public *Phalcon\Mvc\Micro\Collection* **get** (*string* \$routePattern, *callable* \$handler, [*string* \$name])

Maps a route to a handler that only matches if the HTTP method is GET

public *Phalcon\Mvc\Micro\Collection* **post** (*string* \$routePattern, *callable* \$handler, [*string* \$name])

Maps a route to a handler that only matches if the HTTP method is POST

public *Phalcon\Mvc\Micro\Collection* **put** (*string* \$routePattern, *callable* \$handler, [*string* \$name])

Maps a route to a handler that only matches if the HTTP method is PUT

public *Phalcon\Mvc\Micro\Collection* **patch** (*string* \$routePattern, *callable* \$handler, [*string* \$name])

Maps a route to a handler that only matches if the HTTP method is PATCH

public *Phalcon\Mvc\Micro\Collection* **head** (*string* \$routePattern, *callable* \$handler, [*string* \$name])

Maps a route to a handler that only matches if the HTTP method is HEAD

public *Phalcon\Mvc\Micro\Collection* **delete** (*string* \$routePattern, *callable* \$handler, [*string* \$name])

Maps a route to a handler that only matches if the HTTP method is DELETE

public *Phalcon\Mvc\Micro\Collection* **options** (*string* \$routePattern, *callable* \$handler, [*mixed* \$name])

Maps a route to a handler that only matches if the HTTP method is OPTIONS

Class *Phalcon\Mvc\Micro\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private [Exception](#) **__clone** () inherited from [Exception](#)

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [[Exception](#) \$previous]) inherited from [Exception](#)

Exception constructor

public **__wakeup** () inherited from [Exception](#)

...

final public *string* **getMessage** () inherited from [Exception](#)

Gets the Exception message

final public *int* **getCode** () inherited from [Exception](#)

Gets the Exception code

final public *string* **getFile** () inherited from [Exception](#)

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from [Exception](#)

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Mvc\Micro\LazyLoader](#)

Lazy-Load of handlers for Mvc\Micro using auto-loading

Methods

public **getDefinition** ()

...

public **__construct** (*mixed* \$definition)

[Phalcon\Mvc\Micro\LazyLoader](#) constructor

public *mixed* **__call** (*string* \$method, *array* \$arguments)

Initializes the internal handler, calling functions on it

public *mixed* **callMethod** (*string* \$method, *array* \$arguments, [[Phalcon\Mvc\Model\BinderInterface](#) \$modelBinder])

Calling `__call` method

Abstract class `Phalcon\Mvc\Model`

implements `Phalcon\Mvc\EntityInterface`, `Phalcon\Mvc\ModelInterface`, `Phalcon\Mvc\Model\ResultInterface`, `Phalcon\Di\InjectionAwareInterface`, `Serializable`, `JsonSerializable`

`Phalcon\Mvc\Model` connects business objects and database tables to create a persistable domain model where logic and data are presented in one wrapping. It's an implementation of the object-relational mapping (ORM).

A model represents the information (data) of the application and the rules to manipulate that data. Models are primarily used for managing the rules of interaction with a corresponding database table. In most cases, each table in your database will correspond to one model in your application. The bulk of your application's business logic will be concentrated in the models.

`Phalcon\Mvc\Model` is the first ORM written in Zephir/C languages for PHP, giving to developers high performance when interacting with databases while is also easy to use.

```
<?php

$robot = new Robots();

$robot->type = "mechanical";
$robot->name = "Astro Boy";
$robot->year = 1952;

if ($robot->save() === false) {
    echo "Umh, We can store robots: ";

    $messages = $robot->getMessages();

    foreach ($messages as $message) {
        echo message;
    }
} else {
    echo "Great, a new robot was saved successfully!";
}
```

Constants

integer `OP_NONE`

integer `OP_CREATE`

integer `OP_UPDATE`

integer `OP_DELETE`

integer `DIRTY_STATE_PERSISTENT`

integer `DIRTY_STATE_TRANSIENT`

integer `DIRTY_STATE_DETACHED`

Methods

final public **__construct** ([mixed \$data], [*Phalcon\DiInterface* \$dependencyInjector], [*Phalcon\Mvc\Model\ManagerInterface* \$modelsManager])

Phalcon\Mvc\Model constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injection container

public **getDI** ()

Returns the dependency injection container

protected **setEventsManager** (*Phalcon\Events\ManagerInterface* \$eventsManager)

Sets a custom events manager

protected **getEventsManager** ()

Returns the custom events manager

public **getModelsMetaData** ()

Returns the models meta-data service related to the entity instance

public **getModelsManager** ()

Returns the models manager related to the entity instance

public **setTransaction** (*Phalcon\Mvc\Model\TransactionInterface* \$transaction)

Sets a transaction related to the Model instance

```
<?php

use Phalcon\Mvc\Model\Transaction\Manager as TxManager;
use Phalcon\Mvc\Model\Transaction\Failed as TxFailed;

try {
    $txManager = new TxManager();

    $transaction = $txManager->get();

    $robot = new Robots();

    $robot->setTransaction($transaction);

    $robot->name          = "WALL·E";
    $robot->created_at    = date("Y-m-d");

    if ($robot->save() === false) {
        $transaction->rollback("Can't save robot");
    }

    $robotPart = new RobotParts();

    $robotPart->setTransaction($transaction);

    $robotPart->type = "head";

    if ($robotPart->save() === false) {
```

```
        $transaction->rollback("Robot part cannot be saved");
    }

    $transaction->commit();
} catch (TxFailed $e) {
    echo "Failed, reason: ", $e->getMessage();
}
```

protected **setSource** (*mixed* \$source)

Sets the table name to which model should be mapped

public **getSource** ()

Returns the table name mapped in the model

protected **setSchema** (*mixed* \$schema)

Sets schema name where the mapped table is located

public **getSchema** ()

Returns schema name where the mapped table is located

public **setConnectionService** (*mixed* \$connectionService)

Sets the DependencyInjection connection service name

public **setReadConnectionService** (*mixed* \$connectionService)

Sets the DependencyInjection connection service name used to read data

public **setWriteConnectionService** (*mixed* \$connectionService)

Sets the DependencyInjection connection service name used to write data

public **getReadConnectionService** ()

Returns the DependencyInjection connection service name used to read data related the model

public **getWriteConnectionService** ()

Returns the DependencyInjection connection service name used to write data related to the model

public **setDirtyState** (*mixed* \$dirtyState)

Sets the dirty state of the object using one of the DIRTY_STATE_* constants

public **getDirtyState** ()

Returns one of the DIRTY_STATE_* constants telling if the record exists in the database or not

public **getReadConnection** ()

Gets the connection used to read data for the model

public **getWriteConnection** ()

Gets the connection used to write data to the model

public *Phalcon\Mvc\Model* **assign** (array \$data, [*mixed* \$dataColumnMap], [array \$whiteList])

Assigns values to a model from an array

```
<?php
$robot->assign(
    [
```

```

        "type" => "mechanical",
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);

// Assign by db row, column map needed
$robot->assign(
    $dbRow,
    [
        "db_type" => "type",
        "db_name" => "name",
        "db_year" => "year",
    ]
);

// Allow assign only name and year
$robot->assign(
    $_POST,
    null,
    [
        "name",
        "year",
    ]
);

```

public static **cloneResultMap** (*Phalcon\Mvc\ModelInterface* | *Phalcon\Mvc\Model\Row* \$base, array \$data, array \$columnMap, [int \$dirtyState], [boolean \$keepSnapshots])

Assigns values to a model from an array, returning a new model.

```

<?php

$robot = \Phalcon\Mvc\Model::cloneResultMap(
    new Robots(),
    [
        "type" => "mechanical",
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);

```

public static *mixed* **cloneResultMapHydrate** (array \$data, array \$columnMap, int \$hydrationMode)

Returns an hydrated result based on the data and the column map

public static *Phalcon\Mvc\ModelInterface* **cloneResult** (*Phalcon\Mvc\ModelInterface* \$base, array \$data, [int \$dirtyState])

Assigns values to a model from an array returning a new model

```

<?php

$robot = Phalcon\Mvc\Model::cloneResult(
    new Robots(),
    [
        "type" => "mechanical",
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);

```

```
    ]  
);
```

public static **find** ([mixed \$parameters])

Query for a set of records that match the specified conditions

```
<?php  
  
// How many robots are there?  
$robots = Robots::find();  
  
echo "There are ", count($robots), "\n";  
  
// How many mechanical robots are there?  
$robots = Robots::find(  
    "type = 'mechanical'"  
);  
  
echo "There are ", count($robots), "\n";  
  
// Get and print virtual robots ordered by name  
$robots = Robots::find(  
    [  
        "type = 'virtual'",  
        "order" => "name",  
    ]  
);  
  
foreach ($robots as $robot) {  
    echo $robot->name, "\n";  
}  
  
// Get first 100 virtual robots ordered by name  
$robots = Robots::find(  
    [  
        "type = 'virtual'",  
        "order" => "name",  
        "limit" => 100,  
    ]  
);  
  
foreach ($robots as $robot) {  
    echo $robot->name, "\n";  
}
```

public static *static* **findFirst** ([string | array \$parameters])

Query the first record that matches the specified conditions

```
<?php  
  
// What's the first robot in robots table?  
$robot = Robots::findFirst();  
  
echo "The robot name is ", $robot->name;  
  
// What's the first mechanical robot in robots table?  
$robot = Robots::findFirst(  
    "type = 'mechanical'"  
);
```



```

        "type = 'mechanical'"
    );

    echo "The first mechanical robot name is ", $robot->name;

    // Get first virtual robot ordered by name
    $robot = Robots::findFirst(
        [
            "type = 'virtual'",
            "order" => "name",
        ]
    );

    echo "The first virtual robot name is ", $robot->name;

```

public static **query** ([*Phalcon\DiInterface* \$dependencyInjector])

Create a criteria for a specific model

protected *boolean* **_exists** (*Phalcon\Mvc\Model\MetaDataInterface* \$metaData, *Phalcon\Db\AdapterInterface* \$connection, [*string* | *array* \$table])

Checks whether the current record already exists

protected static *Phalcon\Mvc\Model\ResultSetInterface* **_groupResult** (*mixed* \$functionName, *string* \$alias, *array* \$parameters)

Generate a PHQL SELECT statement for an aggregate

public static *mixed* **count** ([*array* \$parameters])

Counts how many records match the specified conditions

```

<?php

// How many robots are there?
$number = Robots::count();

echo "There are ", $number, "\n";

// How many mechanical robots are there?
$number = Robots::count("type = 'mechanical'");

echo "There are ", $number, " mechanical robots\n";

```

public static *mixed* **sum** ([*array* \$parameters])

Calculates the sum on a column for a result-set of rows that match the specified conditions

```

<?php

// How much are all robots?
$sum = Robots::sum(
    [
        "column" => "price",
    ]
);

echo "The total price of robots is ", $sum, "\n";

// How much are mechanical robots?

```

```
$sum = Robots::sum(
    [
        "type = 'mechanical'",
        "column" => "price",
    ]
);

echo "The total price of mechanical robots is ", $sum, "\n";
```

public static *mixed* **maximum** ([array \$parameters])

Returns the maximum value of a column for a result-set of rows that match the specified conditions

```
<?php

// What is the maximum robot id?
$id = Robots::maximum(
    [
        "column" => "id",
    ]
);

echo "The maximum robot id is: ", $id, "\n";

// What is the maximum id of mechanical robots?
$sum = Robots::maximum(
    [
        "type = 'mechanical'",
        "column" => "id",
    ]
);

echo "The maximum robot id of mechanical robots is ", $id, "\n";
```

public static *mixed* **minimum** ([array \$parameters])

Returns the minimum value of a column for a result-set of rows that match the specified conditions

```
<?php

// What is the minimum robot id?
$id = Robots::minimum(
    [
        "column" => "id",
    ]
);

echo "The minimum robot id is: ", $id;

// What is the minimum id of mechanical robots?
$sum = Robots::minimum(
    [
        "type = 'mechanical'",
        "column" => "id",
    ]
);

echo "The minimum robot id of mechanical robots is ", $id;
```

public static *double* **average** ([array \$parameters])

Returns the average value on a column for a result-set of rows matching the specified conditions

```
<?php

// What's the average price of robots?
$average = Robots::average(
    [
        "column" => "price",
    ]
);

echo "The average price is ", $average, "\n";

// What's the average price of mechanical robots?
$average = Robots::average(
    [
        "type = 'mechanical'",
        "column" => "price",
    ]
);

echo "The average price of mechanical robots is ", $average, "\n";
```

public **fireEvent** (*mixed* \$eventName)

Fires an event, implicitly calls behaviors and listeners in the events manager are notified

public **fireEventCancel** (*mixed* \$eventName)

Fires an event, implicitly calls behaviors and listeners in the events manager are notified This method stops if one of the callbacks/listeners returns boolean false

protected **_cancelOperation** ()

Cancel the current operation

public **appendMessage** (*Phalcon\Mvc\Model\MessageInterface* \$message)

Appends a customized message on the validation process

```
<?php

use Phalcon\Mvc\Model;
use Phalcon\Mvc\Model\Message as Message;

class Robots extends Model
{
    public function beforeSave()
    {
        if ($this->name === "Peter") {
            $message = new Message(
                "Sorry, but a robot cannot be named Peter"
            );

            $this->appendMessage($message);
        }
    }
}
```

protected **validate** (*Phalcon\ValidationInterface* \$validator)

Executes validators on every validation call

```
<?php

use Phalcon\Mvc\Model;
use Phalcon\Validation;
use Phalcon\Validation\Validator\ExclusionIn;

class Subscriptors extends Model
{
    public function validation()
    {
        $validator = new Validation();

        $validator->add(
            "status",
            new ExclusionIn(
                [
                    "domain" => [
                        "A",
                        "I",
                    ],
                ]
            )
        );

        return $this->validate($validator);
    }
}
```

public **validationHasFailed** ()

Check whether validation process has generated any messages

```
<?php

use Phalcon\Mvc\Model;
use Phalcon\Validation;
use Phalcon\Validation\Validator\ExclusionIn;

class Subscriptors extends Model
{
    public function validation()
    {
        $validator = new Validation();

        $validator->validate(
            "status",
            new ExclusionIn(
                [
                    "domain" => [
                        "A",
                        "I",
                    ],
                ]
            )
        );
    }
}
```

```

        return $this->validate($validator);
    }
}

```

public **getMessages** ([mixed \$filter])

Returns array of validation messages

```

<?php

$robot = new Robots();

$robot->type = "mechanical";
$robot->name = "Astro Boy";
$robot->year = 1952;

if ($robot->save() === false) {
    echo "Umh, We can't store robots right now ";

    $messages = $robot->getMessages();

    foreach ($messages as $message) {
        echo $message;
    }
} else {
    echo "Great, a new robot was saved successfully!";
}

```

final protected **_checkForeignKeysRestrict** ()

Reads “belongs to” relations and check the virtual foreign keys when inserting or updating records to verify that inserted/updated values are present in the related entity

final protected **_checkForeignKeysReverseCascade** ()

Reads both “hasMany” and “hasOne” relations and checks the virtual foreign keys (cascade) when deleting records

final protected **_checkForeignKeysReverseRestrict** ()

Reads both “hasMany” and “hasOne” relations and checks the virtual foreign keys (restrict) when deleting records

protected **_preSave** (*Phalcon\Mvc\Model\MetaDataInterface* \$metaData, *mixed* \$exists, *mixed* \$identityField)

Executes internal hooks before save a record

protected **_postSave** (*mixed* \$success, *mixed* \$exists)

Executes internal events after save a record

protected *boolean* **_doLowInsert** (*Phalcon\Mvc\Model\MetaDataInterface* \$metaData, *Phalcon\Db\AdapterInterface* \$connection, *string* | *array* \$table, *boolean* | *string* \$identityField)

Sends a pre-build INSERT SQL statement to the relational database system

protected *boolean* **_doLowUpdate** (*Phalcon\Mvc\Model\MetaDataInterface* \$metaData, *Phalcon\Db\AdapterInterface* \$connection, *string* | *array* \$table)

Sends a pre-build UPDATE SQL statement to the relational database system

protected *boolean* **_preSaveRelatedRecords** (*Phalcon\Db\AdapterInterface* \$connection, *Phalcon\Mvc\ModelInterface*[] \$related)

Saves related records that must be stored prior to save the master record

protected *boolean* **_postSaveRelatedRecords** (*Phalcon\Db\AdapterInterface* \$connection, *Phalcon\Mvc\ModelInterface*[] \$related)

Save the related records assigned in the has-one/has-many relations

public *boolean* **save** ([*array* \$data], [*array* \$whiteList])

Inserts or updates a model instance. Returning true on success or false otherwise.

```
<?php

// Creating a new robot
$robot = new Robots();

$robot->type = "mechanical";
$robot->name = "Astro Boy";
$robot->year = 1952;

$robot->save();

// Updating a robot name
$robot = Robots::findFirst("id = 100");

$robot->name = "Biomass";

$robot->save();
```

public **create** ([*mixed* \$data], [*mixed* \$whiteList])

Inserts a model instance. If the instance already exists in the persistence it will throw an exception Returning true on success or false otherwise.

```
<?php

// Creating a new robot
$robot = new Robots();

$robot->type = "mechanical";
$robot->name = "Astro Boy";
$robot->year = 1952;

$robot->create();

// Passing an array to create
$robot = new Robots();

$robot->create(
    [
        "type" => "mechanical",
        "name" => "Astro Boy",
        "year" => 1952,
    ]
);
```

public **update** ([*mixed* \$data], [*mixed* \$whiteList])

Updates a model instance. If the instance doesn't exist in the persistence it will throw an exception Returning true on success or false otherwise.

```
<?php

// Updating a robot name
$robot = Robots::findFirst("id = 100");

$robot->name = "Biomass";

$robot->update();
```

public **delete** ()

Deletes a model instance. Returning true on success or false otherwise.

```
<?php

$robot = Robots::findFirst("id=100");

$robot->delete();

$robots = Robots::find("type = 'mechanical'");

foreach ($robots as $robot) {
    $robot->delete();
}
```

public **getOperationMade** ()

Returns the type of the latest operation performed by the ORM Returns one of the OP_* class constants

public **refresh** ()

Refreshes the model attributes re-querying the record from the database

public **skipOperation** (*mixed* \$skip)

Skips the current operation forcing a success state

public **readAttribute** (*mixed* \$attribute)

Reads an attribute value by its name

```
<?php

echo $robot->readAttribute("name");
```

public **writeAttribute** (*mixed* \$attribute, *mixed* \$value)

Writes an attribute value by its name

```
<?php

$robot->writeAttribute("name", "Rosey");
```

protected **skipAttributes** (*array* \$attributes)

Sets a list of attributes that must be skipped from the generated INSERT/UPDATE statement

```
<?php

<?php
```

```
class Robots extends \Phalcon\Mvc\Model
{
    public function initialize()
    {
        $this->skipAttributes(
            [
                "price",
            ]
        );
    }
}
```

protected **skipAttributesOnCreate** (*array* \$attributes)

Sets a list of attributes that must be skipped from the generated INSERT statement

```
<?php

<?php

class Robots extends \Phalcon\Mvc\Model
{
    public function initialize()
    {
        $this->skipAttributesOnCreate(
            [
                "created_at",
            ]
        );
    }
}
```

protected **skipAttributesOnUpdate** (*array* \$attributes)

Sets a list of attributes that must be skipped from the generated UPDATE statement

```
<?php

<?php

class Robots extends \Phalcon\Mvc\Model
{
    public function initialize()
    {
        $this->skipAttributesOnUpdate(
            [
                "modified_in",
            ]
        );
    }
}
```

protected **allowEmptyStringValue** (*array* \$attributes)

Sets a list of attributes that must be skipped from the generated UPDATE statement

```
<?php

<?php
```



```

class Robots extends \Phalcon\Mvc\Model
{
    public function initialize()
    {
        $this->allowEmptyStringValues(
            [
                "name",
            ]
        );
    }
}

```

protected **hasOne** (*mixed* \$fields, *mixed* \$referenceModel, *mixed* \$referencedFields, [*mixed* \$options])

Setup a 1-1 relation between two models

```

<?php
<?php

class Robots extends \Phalcon\Mvc\Model
{
    public function initialize()
    {
        $this->hasOne("id", "RobotsDescription", "robots_id");
    }
}

```

protected **belongsTo** (*mixed* \$fields, *mixed* \$referenceModel, *mixed* \$referencedFields, [*mixed* \$options])

Setup a reverse 1-1 or n-1 relation between two models

```

<?php
<?php

class RobotsParts extends \Phalcon\Mvc\Model
{
    public function initialize()
    {
        $this->belongsTo("robots_id", "Robots", "id");
    }
}

```

protected **hasMany** (*mixed* \$fields, *mixed* \$referenceModel, *mixed* \$referencedFields, [*mixed* \$options])

Setup a 1-n relation between two models

```

<?php
<?php

class Robots extends \Phalcon\Mvc\Model
{
    public function initialize()
    {
        $this->hasMany("id", "RobotsParts", "robots_id");
    }
}

```

```
}  
}
```

protected *Phalcon\Mvc\Model\Relation* **hasManyToMany** (*string* | *array* \$fields, *string* \$intermediateModel, *string* | *array* \$intermediateFields, *string* | *array* \$intermediateReferencedFields, *mixed* \$referenceModel, *string* | *array* \$referencedFields, [*array* \$options])

Setup an n-n relation between two models, through an intermediate relation

```
<?php  
  
<?php  
  
class Robots extends \Phalcon\Mvc\Model  
{  
    public function initialize()  
    {  
        // Setup a many-to-many relation to Parts through RobotsParts  
        $this->hasManyToMany(  
            "id",  
            "RobotsParts",  
            "robots_id",  
            "parts_id",  
            "Parts",  
            "id",  
        );  
    }  
}
```

public **addBehavior** (*Phalcon\Mvc\Model\BehaviorInterface* \$behavior)

Setups a behavior in a model

```
<?php  
  
<?php  
  
use Phalcon\Mvc\Model;  
use Phalcon\Mvc\Model\Behavior\Timestampable;  
  
class Robots extends Model  
{  
    public function initialize()  
    {  
        $this->addBehavior(  
            new Timestampable(  
                [  
                    "onCreate" => [  
                        "field" => "created_at",  
                        "format" => "Y-m-d",  
                    ],  
                ],  
            )  
        );  
    }  
}
```

protected **keepSnapshots** (*mixed* \$keepSnapshot)

Sets if the model must keep the original record snapshot in memory

```
<?php

<?php

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function initialize()
    {
        $this->keepSnapshots(true);
    }
}
```

public **setSnapshotData** (array \$data, [array \$columnMap])

Sets the record's snapshot data. This method is used internally to set snapshot data when the model was set up to keep snapshot data

public **hasSnapshotData** ()

Checks if the object has internal snapshot data

public **getSnapshotData** ()

Returns the internal snapshot data

public **hasChanged** ([string | array \$fieldName])

Check if a specific attribute has changed This only works if the model is keeping data snapshots

public **getChangedFields** ()

Returns a list of changed values.

```
<?php

$robots = Robots::findFirst();
print_r($robots->getChangedFields()); // []

$robots->deleted = 'Y';

$robots->getChangedFields();
print_r($robots->getChangedFields()); // ["deleted"]
```

protected **useDynamicUpdate** (mixed \$dynamicUpdate)

Sets if a model must use dynamic update instead of the all-field update

```
<?php

<?php

use Phalcon\Mvc\Model;

class Robots extends Model
{
    public function initialize()
    {
        $this->useDynamicUpdate(true);
    }
}
```

```
}
}
```

public *Phalcon\Mvc\Model\ResultsetInterface* **getRelated** (*string* \$alias, [*array* \$arguments])

Returns related records based on defined relations

protected *mixed* **__getRelatedRecords** (*string* \$modelName, *string* \$method, *array* \$arguments)

Returns related records defined relations depending on the method name

final protected static *Phalcon\Mvc\ModelInterface*[] | *Phalcon\Mvc\ModelInterface* | *boolean* **__invokeFinder** (*string* \$method, *array* \$arguments)

Try to check if the query must invoke a finder

public *mixed* **__call** (*string* \$method, *array* \$arguments)

Handles method calls when a method is not implemented

public static *mixed* **__callStatic** (*string* \$method, *array* \$arguments)

Handles method calls when a static method is not implemented

public **__set** (*string* \$property, *mixed* \$value)

Magic method to assign values to the the model

final protected *string* **__possibleSetter** (*string* \$property, *mixed* \$value)

Check for, and attempt to use, possible setter.

public *Phalcon\Mvc\Model\Resultset* | *Phalcon\Mvc\Model* **__get** (*string* \$property)

Magic method to get related records using the relation alias as a property

public **__isset** (*mixed* \$property)

Magic method to check if a property is a valid relation

public **serialize** ()

Serializes the object ignoring connections, services, related objects or static properties

public **unserialize** (*mixed* \$data)

Unserializes the object from a serialized string

public **dump** ()

Returns a simple representation of the object that can be used with var_dump

```
<?php
var_dump (
    $robot->dump ()
);
```

public *array* **toArray** ([*array* \$columns])

Returns the instance as an array representation

```
<?php
print_r (
    $robot->toArray ()
);
```

```
public array jsonSerialize ()
```

Serializes the object for json_encode

```
<?php
```

```
echo json_encode($robot);
```

```
public static setup (array $options)
```

Enables/disables options in the ORM

```
public reset ()
```

Reset a model instance data

Abstract class `Phalcon\Mvc\Model\Behavior`

implements `Phalcon\Mvc\Model\BehaviorInterface`

This is an optional base class for ORM behaviors

Methods

```
public __construct ([array $options])
```

```
protected mustTakeAction (mixed $eventName)
```

Checks whether the behavior must take action on certain event

```
protected array getOptions ([string $eventName])
```

Returns the behavior options related to an event

```
public notify (mixed $type, Phalcon\Mvc\ModelInterface $model)
```

This method receives the notifications from the EventsManager

```
public missingMethod (Phalcon\Mvc\ModelInterface $model, string $method, [array $arguments])
```

Acts as fallbacks when a missing method is called on the model

Class `Phalcon\Mvc\Model\Behavior\SoftDelete`

extends abstract class `Phalcon\Mvc\Model\Behavior`

implements `Phalcon\Mvc\Model\BehaviorInterface`

Instead of permanently delete a record it marks the record as deleted changing the value of a flag column

Methods

```
public notify (mixed $type, Phalcon\Mvc\ModelInterface $model)
```

Listens for notifications from the models manager

```
public __construct ([array $options]) inherited from Phalcon\Mvc\Model\Behavior
```

Phalcon\Mvc\Model\Behavior

protected **mustTakeAction** (*mixed* \$eventName) inherited from *Phalcon\Mvc\Model\Behavior*

Checks whether the behavior must take action on certain event

protected array **getOptions** ([*string* \$eventName]) inherited from *Phalcon\Mvc\Model\Behavior*

Returns the behavior options related to an event

public **missingMethod** (*Phalcon\Mvc\ModelInterface* \$model, *string* \$method, [*array* \$arguments]) inherited from *Phalcon\Mvc\Model\Behavior*

Acts as fallbacks when a missing method is called on the model

Class Phalcon\Mvc\Model\Behavior\Timestampable

extends abstract class *Phalcon\Mvc\Model\Behavior*

implements *Phalcon\Mvc\Model\BehaviorInterface*

Allows to automatically update a model's attribute saving the datetime when a record is created or updated

Methods

public **notify** (*mixed* \$type, *Phalcon\Mvc\ModelInterface* \$model)

Listens for notifications from the models manager

public **__construct** ([*array* \$options]) inherited from *Phalcon\Mvc\Model\Behavior*

Phalcon\Mvc\Model\Behavior

protected **mustTakeAction** (*mixed* \$eventName) inherited from *Phalcon\Mvc\Model\Behavior*

Checks whether the behavior must take action on certain event

protected array **getOptions** ([*string* \$eventName]) inherited from *Phalcon\Mvc\Model\Behavior*

Returns the behavior options related to an event

public **missingMethod** (*Phalcon\Mvc\ModelInterface* \$model, *string* \$method, [*array* \$arguments]) inherited from *Phalcon\Mvc\Model\Behavior*

Acts as fallbacks when a missing method is called on the model

Class Phalcon\Mvc\Model\Criteria

implements *Phalcon\Mvc\Model\CriteriaInterface*, *Phalcon\DI\InjectionAwareInterface*

This class is used to build the array parameter required by *Phalcon\Mvc\Model::find()* and *Phalcon\Mvc\Model::findFirst()* using an object-oriented interface.

```
<?php
$robots = Robots::query()
    ->where("type = :type:")
    ->andWhere("year < 2000")
    ->bind(["type" => "mechanical"])
    ->limit(5, 10)
```

```
->orderBy("name")
->execute();
```

Methods

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the DependencyInjector container

public **getDI** ()

Returns the DependencyInjector container

public **setModelName** (*mixed* \$modelName)

Set a model on which the query will be executed

public **getModelName** ()

Returns an internal model name on which the criteria will be applied

public **bind** (*array* \$bindParam, [*mixed* \$merge])

Sets the bound parameters in the criteria This method replaces all previously set bound parameters

public **bindTypes** (*array* \$bindTypes)

Sets the bind types in the criteria This method replaces all previously set bound parameters

public **distinct** (*mixed* \$distinct)

Sets SELECT DISTINCT / SELECT ALL flag

public *Phalcon\Mvc\Model\Criteria* **columns** (*string* | *array* \$columns)

Sets the columns to be queried

```
<?php
$criteria->columns (
    [
        "id",
        "name",
    ]
);
```

public **join** (*mixed* \$model, [*mixed* \$conditions], [*mixed* \$alias], [*mixed* \$type])

Adds an INNER join to the query

```
<?php
$criteria->join("Robots");
$criteria->join("Robots", "r.id = RobotsParts.robots_id");
$criteria->join("Robots", "r.id = RobotsParts.robots_id", "r");
$criteria->join("Robots", "r.id = RobotsParts.robots_id", "r", "LEFT");
```

public **innerJoin** (*mixed* \$model, [*mixed* \$conditions], [*mixed* \$alias])

Adds an INNER join to the query

```
<?php

$criteria->innerJoin("Robots");
$criteria->innerJoin("Robots", "r.id = RobotsParts.robots_id");
$criteria->innerJoin("Robots", "r.id = RobotsParts.robots_id", "r");
```

public **leftJoin** (*mixed* \$model, [*mixed* \$conditions], [*mixed* \$alias])

Adds a LEFT join to the query

```
<?php

$criteria->leftJoin("Robots", "r.id = RobotsParts.robots_id", "r");
```

public **rightJoin** (*mixed* \$model, [*mixed* \$conditions], [*mixed* \$alias])

Adds a RIGHT join to the query

```
<?php

$criteria->rightJoin("Robots", "r.id = RobotsParts.robots_id", "r");
```

public **where** (*mixed* \$conditions, [*mixed* \$bindParam], [*mixed* \$bindTypes])

Sets the conditions parameter in the criteria

public **addWhere** (*mixed* \$conditions, [*mixed* \$bindParam], [*mixed* \$bindTypes])

Appends a condition to the current conditions using an AND operator (deprecated)

public **andWhere** (*mixed* \$conditions, [*mixed* \$bindParam], [*mixed* \$bindTypes])

Appends a condition to the current conditions using an AND operator

public **orWhere** (*mixed* \$conditions, [*mixed* \$bindParam], [*mixed* \$bindTypes])

Appends a condition to the current conditions using an OR operator

public **betweenWhere** (*mixed* \$expr, *mixed* \$minimum, *mixed* \$maximum)

Appends a BETWEEN condition to the current conditions

```
<?php

$criteria->betweenWhere("price", 100.25, 200.50);
```

public **notBetweenWhere** (*mixed* \$expr, *mixed* \$minimum, *mixed* \$maximum)

Appends a NOT BETWEEN condition to the current conditions

```
<?php

$criteria->notBetweenWhere("price", 100.25, 200.50);
```

public **inWhere** (*mixed* \$expr, array \$values)

Appends an IN condition to the current conditions

```
<?php

$criteria->inWhere("id", [1, 2, 3]);
```


public **notInWhere** (*mixed* \$expr, *array* \$values)

Appends a NOT IN condition to the current conditions

```
<?php
$criteria->notInWhere("id", [1, 2, 3]);
```

public **conditions** (*mixed* \$conditions)

Adds the conditions parameter to the criteria

public **order** (*mixed* \$orderColumns)

Adds the order-by parameter to the criteria (deprecated)

public **orderBy** (*mixed* \$orderColumns)

Adds the order-by clause to the criteria

public **groupBy** (*mixed* \$group)

Adds the group-by clause to the criteria

public **having** (*mixed* \$having)

Adds the having clause to the criteria

public **limit** (*mixed* \$limit, [*mixed* \$offset])

Adds the limit parameter to the criteria.

```
<?php
$criteria->limit(100);
$criteria->limit(100, 200);
$criteria->limit("100", "200");
```

public **forUpdate** ([*mixed* \$forUpdate])

Adds the “for_update” parameter to the criteria

public **sharedLock** ([*mixed* \$sharedLock])

Adds the “shared_lock” parameter to the criteria

public **cache** (*array* \$cache)

Sets the cache options in the criteria This method replaces all previously set cache options

public **getWhere** ()

Returns the conditions parameter in the criteria

public *string* | *array* | *null* **getColumns** ()

Returns the columns to be queried

public **getConditions** ()

Returns the conditions parameter in the criteria

public *int* | *array* | *null* **getLimit** ()

Returns the limit parameter in the criteria, which will be an integer if limit was set without an offset, an array with ‘number’ and ‘offset’ keys if an offset was set with the limit, or null if limit has not been set.

public **getOrderBy** ()

Returns the order clause in the criteria

public **getGroupBy** ()

Returns the group clause in the criteria

public **getHaving** ()

Returns the having clause in the criteria

public *array* **getParams** ()

Returns all the parameters defined in the criteria

public static **fromInput** (*Phalcon\DiInterface* \$dependencyInjector, *mixed* \$modelName, *array* \$data, [*mixed* \$operator])

Builds a Phalcon\Mvc\Model\Criteria based on an input array like _POST

public **execute** ()

Executes a find using the parameters built with the criteria

Class Phalcon\Mvc\Model\Exception

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Mvc\Model\Manager](#)

implements [Phalcon\Mvc\Model\ManagerInterface](#), [Phalcon\Di\InjectionAwareInterface](#), [Phalcon\Events\EventsAwareInterface](#)

This components controls the initialization of models, keeping record of relations between the different models of the application.

A ModelsManager is injected to a model via a Dependency Injector/Services Container such as [Phalcon\Di](#).

```
<?php

use Phalcon\Di;
use Phalcon\Mvc\Model\Manager as ModelsManager;

$di = new Di();

$di->set(
    "modelsManager",
    function() {
        return new ModelsManager();
    }
);

$robot = new Robots($di);
```

Methods

public **setDI** ([Phalcon\DiInterface](#) \$dependencyInjector)

Sets the DependencyInjector container

public **getDI** ()

Returns the DependencyInjector container

public **setEventsManager** ([Phalcon\Events\ManagerInterface](#) \$eventsManager)

Sets a global events manager

public **getEventsManager** ()

Returns the internal event manager

public **setCustomEventsManager** ([Phalcon\Mvc\ModelInterface](#) \$model, [Phalcon\Events\ManagerInterface](#) \$eventsManager)

Sets a custom events manager for a specific model

public **getCustomEventsManager** ([Phalcon\Mvc\ModelInterface](#) \$model)

Returns a custom events manager related to a model

public **initialize** ([Phalcon\Mvc\ModelInterface](#) \$model)

Initializes a model in the model manager

public **isInitialized** (*mixed* \$modelName)

Check whether a model is already initialized

public **getLastInitialized** ()

Get last initialized model

public **load** (*mixed* \$modelName, [*mixed* \$newInstance])

Loads a model throwing an exception if it doesn't exist

public **setModelSource** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$source)

Sets the mapped source for a model

final public **isVisibleModelProperty** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$property)

Check whether a model property is declared as public.

```
<?php

$isPublic = $manager->isVisibleModelProperty(
    new Robots(),
    "name"
);
```

public **getModelSource** (*Phalcon\Mvc\ModelInterface* \$model)

Returns the mapped source for a model

public **setModelSchema** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$schema)

Sets the mapped schema for a model

public **getModelSchema** (*Phalcon\Mvc\ModelInterface* \$model)

Returns the mapped schema for a model

public **setConnectionService** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$connectionService)

Sets both write and read connection service for a model

public **setWriteConnectionService** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$connectionService)

Sets write connection service for a model

public **setReadConnectionService** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$connectionService)

Sets read connection service for a model

public **getReadConnection** (*Phalcon\Mvc\ModelInterface* \$model)

Returns the connection to read data related to a model

public **getWriteConnection** (*Phalcon\Mvc\ModelInterface* \$model)

Returns the connection to write data related to a model

protected **_getConnection** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$connectionServices)

Returns the connection to read or write data related to a model depending on the connection services.

public **getReadConnectionService** (*Phalcon\Mvc\ModelInterface* \$model)

Returns the connection service name used to read data related to a model

public **getWriteConnectionService** (*Phalcon\Mvc\ModelInterface* \$model)

Returns the connection service name used to write data related to a model

public **_getConnectionService** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$connectionServices)

Returns the connection service name used to read or write data related to a model depending on the connection services

public **notifyEvent** (*mixed* \$eventName, *Phalcon\Mvc\ModelInterface* \$model)

Receives events generated in the models and dispatches them to an events-manager if available Notify the behaviors that are listening in the model

public **missingMethod** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$eventName, *mixed* \$data)

Dispatch an event to the listeners and behaviors This method expects that the endpoint listeners/behaviors returns true meaning that a least one was implemented

public **addBehavior** (*Phalcon\Mvc\ModelInterface* \$model, *Phalcon\Mvc\Model\BehaviorInterface* \$behavior)

Binds a behavior to a model

public **keepSnapshots** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$keepSnapshots)

Sets if a model must keep snapshots

public **isKeepingSnapshots** (*Phalcon\Mvc\ModelInterface* \$model)

Checks if a model is keeping snapshots for the queried records

public **useDynamicUpdate** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$dynamicUpdate)

Sets if a model must use dynamic update instead of the all-field update

public **isUsingDynamicUpdate** (*Phalcon\Mvc\ModelInterface* \$model)

Checks if a model is using dynamic update instead of all-field update

public *Phalcon\Mvc\Model\Relation* **addHasOne** (*Phalcon\Mvc\Model* \$model, *mixed* \$fields, *string* \$referencedModel, *mixed* \$referencedFields, [*array* \$options])

Setup a 1-1 relation between two models

public *Phalcon\Mvc\Model\Relation* **addBelongsTo** (*Phalcon\Mvc\Model* \$model, *mixed* \$fields, *string* \$referencedModel, *mixed* \$referencedFields, [*array* \$options])

Setup a relation reverse many to one between two models

public **addHasMany** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$fields, *string* \$referencedModel, *mixed* \$referencedFields, [*array* \$options])

Setup a relation 1-n between two models

public *Phalcon\Mvc\Model\Relation* **addHasManyToMany** (*Phalcon\Mvc\ModelInterface* \$model, *string* \$fields, *string* \$intermediateModel, *string* \$intermediateFields, *string* \$intermediateReferencedFields, *string* \$referencedModel, *string* \$referencedFields, [*array* \$options])

Setups a relation n-m between two models

public **existsBelongsTo** (*mixed* \$modelName, *mixed* \$modelRelation)

Checks whether a model has a belongsTo relation with another model

public **existsHasMany** (*mixed* \$modelName, *mixed* \$modelRelation)

Checks whether a model has a hasMany relation with another model

public **existsHasOne** (*mixed* \$modelName, *mixed* \$modelRelation)

Checks whether a model has a hasOne relation with another model

public **existsHasManyToMany** (*mixed* \$modelName, *mixed* \$modelRelation)

Checks whether a model has a hasManyToMany relation with another model

public **getRelationByAlias** (*mixed* \$modelName, *mixed* \$alias)

Returns a relation by its alias

final protected **_mergeFindParameters** (*mixed* \$findParamsOne, *mixed* \$findParamsTwo)

Merge two arrays of find parameters

public *Phalcon\Mvc\Model\Resultset\Simple* | *Phalcon\Mvc\Model\Resultset\Simple* | *int* | *false* **getRelationRecords** (*Phalcon\Mvc\Model\RelationInterface* \$relation, *mixed* \$method, *Phalcon\Mvc\ModelInterface* \$record, [*mixed* \$parameters])

Helper method to query records based on a relation definition

public **getReusableRecords** (*mixed* \$modelName, *mixed* \$key)

Returns a reusable object from the internal list

public **setReusableRecords** (*mixed* \$modelName, *mixed* \$key, *mixed* \$records)

Stores a reusable record in the internal list

public **clearReusableObjects** ()

Clears the internal reusable list

public **getBelongsToManyRecords** (*mixed* \$method, *mixed* \$modelName, *mixed* \$modelRelation, *Phalcon\Mvc\ModelInterface* \$record, [*mixed* \$parameters])

Gets belongsToMany related records from a model

public **getHasManyRecords** (*mixed* \$method, *mixed* \$modelName, *mixed* \$modelRelation, *Phalcon\Mvc\ModelInterface* \$record, [*mixed* \$parameters])

Gets hasMany related records from a model

public **getHasOneRecords** (*mixed* \$method, *mixed* \$modelName, *mixed* \$modelRelation, *Phalcon\Mvc\ModelInterface* \$record, [*mixed* \$parameters])

Gets belongsToMany related records from a model

public **getBelongsToMany** (*Phalcon\Mvc\ModelInterface* \$model)

Gets all the belongsToMany relations defined in a model

```
<?php
$relations = $modelsManager->getBelongsToMany(
    new Robots()
);
```

public **getHasMany** (*Phalcon\Mvc\ModelInterface* \$model)

Gets hasMany relations defined on a model

public **getHasOne** (*Phalcon\Mvc\ModelInterface* \$model)

Gets hasOne relations defined on a model

public **getHasManyToMany** (*Phalcon\Mvc\ModelInterface* \$model)

Gets hasManyToMany relations defined on a model

public **getHasOneAndHasMany** (*Phalcon\Mvc\ModelInterface* \$model)

Gets hasOne relations defined on a model

public **getRelations** (*mixed* \$modelName)

Query all the relationships defined on a model

public **getRelationsBetween** (*mixed* \$first, *mixed* \$second)

Query the first relationship defined between two models

public **createQuery** (*mixed* \$phql)

Creates a Phalcon\Mvc\Model\Query without execute it

public **executeQuery** (*mixed* \$phql, [*mixed* \$placeholders], [*mixed* \$types])

Creates a Phalcon\Mvc\Model\Query and execute it

public **createBuilder** ([*mixed* \$params])

Creates a Phalcon\Mvc\Model\Query\Builder

public **getLastQuery** ()

Returns the last query created or executed in the models manager

public **registerNamespaceAlias** (*mixed* \$alias, *mixed* \$namespaceName)

Registers shorter aliases for namespaces in PHQL statements

public **getNamespaceAlias** (*mixed* \$alias)

Returns a real namespace from its alias

public **getNamespaceAliases** ()

Returns all the registered namespace aliases

public **__destruct** ()

Destroys the current PHQL cache

Class Phalcon\Mvc\Model\Message

implements Phalcon\Mvc\Model\MessageInterface

Encapsulates validation info generated before save/delete records fails

```
<?php

use Phalcon\Mvc\Model\Message as Message;

class Robots extends \Phalcon\Mvc\Model
{
    public function beforeSave()
    {
        if ($this->name === "Peter") {
            $text = "A robot cannot be named Peter";
            $field = "name";
            $type = "InvalidValue";

            $message = new Message($text, $field, $type);

            $this->appendMessage($message);
        }
    }
}
```

```
}  
}
```

Methods

public **__construct** (*string* \$message, [*string* | *array* \$field], [*string* \$type], [*Phalcon\Mvc\ModelInterface* \$model], [*int* | *null* \$code])

Phalcon\Mvc\Model\Message constructor

public **setType** (*mixed* \$type)

Sets message type

public **getType** ()

Returns message type

public **setMessage** (*mixed* \$message)

Sets verbose message

public **getMessage** ()

Returns verbose message

public **setField** (*mixed* \$field)

Sets field name related to message

public **getField** ()

Returns field name related to message

public **setModel** (*Phalcon\Mvc\ModelInterface* \$model)

Set the model who generates the message

public **setCode** (*mixed* \$code)

Sets code for the message

public **getModel** ()

Returns the model that produced the message

public **getCode** ()

Returns the message code

public **__toString** ()

Magic __toString method returns verbose message

public static **__set_state** (*array* \$message)

Magic __set_state helps to re-build messages variable exporting

Abstract class Phalcon\Mvc\Model\MetaData

implements *Phalcon\Di\InjectionAwareInterface*, *Phalcon\Mvc\Model\MetaDataInterface*

Because Phalcon\Mvc\Model requires meta-data like field names, data types, primary keys, etc. this component collect them and store for further querying by Phalcon\Mvc\Model. Phalcon\Mvc\Model\MetaData can also use adapters to store temporarily or permanently the meta-data.

A standard Phalcon\Mvc\Model\MetaData can be used to query model attributes:

```
<?php

$metaData = new \Phalcon\Mvc\Model\MetaData\Memory();

$attributes = $metaData->getAttributes(
    new Robots()
);

print_r($attributes);
```

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**

integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**

integer **MODELS_DEFAULT_VALUES**

integer **MODELS_EMPTY_STRING_VALUES**

integer **MODELS_COLUMN_MAP**

integer **MODELS_REVERSE_COLUMN_MAP**

Methods

final protected **_initialize** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$key, *mixed* \$table, *mixed* \$schema)

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the DependencyInjector container

public **getDI** ()

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\StrategyInterface* \$strategy)

Set the meta-data extraction strategy

public **getStrategy** ()

Return the strategy to obtain the meta-data

final public **readMetaData** (*Phalcon\Mvc\ModelInterface* \$model)

Reads the complete meta-data for certain model

```
<?php

print_r(
    $metaData->readMetaData(
        new Robots()
    )
);
```

final public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index)

Reads meta-data for certain model

```
<?php

print_r(
    $metaData->readMetaDataIndex(
        new Robots(),
        0
    )
);
```

final public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index, *mixed* \$data)

Writes meta-data for certain model using a MODEL_* constant

```
<?php

print_r(
    $metaData->writeColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP,
        [
            "leName" => "name",
        ]
    )
);
```

final public **readColumnMap** (*Phalcon\Mvc\ModelInterface* \$model)

Reads the ordered/reversed column map for certain model

```
<?php

print_r(
    $metaData->readColumnMap(
        new Robots()
    )
);
```

final public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* \$model, mixed \$index)

Reads column-map information for certain model using a MODEL_* constant

```
<?php

print_r(
    $metaData->readColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP
    )
);
```

public **getAttributes** (*Phalcon\Mvc\ModelInterface* \$model)

Returns table attributes names (fields)

```
<?php

print_r(
    $metaData->getAttributes(
        new Robots()
    )
);
```

public **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model)

Returns an array of fields which are part of the primary key

```
<?php

print_r(
    $metaData->getPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model)

Returns an array of fields which are not part of the primary key

```
<?php

print_r(
    $metaData->getNonPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* \$model)

Returns an array of not null attributes

```
<?php

print_r(
    $metaData->getNotNullAttributes(
        new Robots()
    )
);
```

public **getDataTypes** (*Phalcon\Mvc\ModelInterface* \$model)

Returns attributes and their data types

```
<?php

print_r(
    $metaData->getDataTypes (
        new Robots ()
    )
);
```

public **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* \$model)

Returns attributes which types are numerical

```
<?php

print_r(
    $metaData->getDataTypesNumeric (
        new Robots ()
    )
);
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* \$model)

Returns the name of identity field (if one is present)

```
<?php

print_r(
    $metaData->getIdentityField (
        new Robots ()
    )
);
```

public **getBindTypes** (*Phalcon\Mvc\ModelInterface* \$model)

Returns attributes and their bind data types

```
<?php

print_r(
    $metaData->getBindTypes (
        new Robots ()
    )
);
```

public **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model)

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r(
    $metaData->getAutomaticCreateAttributes (
        new Robots ()
    )
);
```

public **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model)

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r(
    $metaData->getAutomaticUpdateAttributes (
        new Robots ()
    )
);
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes)

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes (
    new Robots (),
    [
        "created_at" => true,
    ]
);
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes)

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes (
    new Robots (),
    [
        "modified_at" => true,
    ]
);
```

public **setEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes)

Set the attributes that allow empty string values

```
<?php

$metaData->setEmptyStringAttributes (
    new Robots (),
    [
        "name" => true,
    ]
);
```

public **getEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model)

Returns attributes allow empty strings

```
<?php

print_r(
    $metaData->getEmptyStringAttributes (
        new Robots ()
    )
);
```

```
    )  
);
```

public **getDefaultValues** (*Phalcon\Mvc\ModelInterface* \$model)

Returns attributes (which have default values) and their default values

```
<?php  
  
print_r(  
    $metaData->getDefaultValues(  
        new Robots()  
    )  
);
```

public **getColumnMap** (*Phalcon\Mvc\ModelInterface* \$model)

Returns the column map if any

```
<?php  
  
print_r(  
    $metaData->getColumnMap(  
        new Robots()  
    )  
);
```

public **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* \$model)

Returns the reverse column map if any

```
<?php  
  
print_r(  
    $metaData->getReverseColumnMap(  
        new Robots()  
    )  
);
```

public **hasAttribute** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$attribute)

Check if a model has certain attribute

```
<?php  
  
var_dump(  
    $metaData->hasAttribute(  
        new Robots(),  
        "name"  
    )  
);
```

public **isEmpty** ()

Checks if the internal meta-data container is empty

```
<?php  
  
var_dump(  
    $metaData->isEmpty()  
);
```

```
$metaData->isEmpty()
);
```

public **reset** ()

Resets internal meta-data in order to regenerate it

```
<?php
$metaData->reset();
```

abstract public **read** (*mixed* \$key) inherited from *Phalcon\Mvc\Model\MetaDataInterface*

...

abstract public **write** (*mixed* \$key, *mixed* \$data) inherited from *Phalcon\Mvc\Model\MetaDataInterface*

...

Class *Phalcon\Mvc\Model\MetaData\Apc*

extends abstract class *Phalcon\Mvc\Model\MetaData*

implements *Phalcon\Mvc\Model\MetaDataInterface*, *Phalcon\DI\InjectionAwareInterface*

Stores model meta-data in the APC cache. Data will erased if the web server is restarted

By default meta-data is stored for 48 hours (172800 seconds)

You can query the meta-data by printing `apc_fetch('PMM$')` or `apc_fetch('PMM$my-app-id')`

```
<?php
$metaData = new \Phalcon\Mvc\Model\Metadatas\Apc(
    [
        "prefix" => "my-app-id",
        "lifetime" => 86400,
    ]
);
```

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**
integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**
integer **MODELS_DEFAULT_VALUES**
integer **MODELS_EMPTY_STRING_VALUES**
integer **MODELS_COLUMN_MAP**
integer **MODELS_REVERSE_COLUMN_MAP**

Methods

public **__construct** ([array \$options])

Phalcon\Mvc\Model\MetaData\Apc constructor

public **read** (*mixed* \$key)

Reads meta-data from APC

public **write** (*mixed* \$key, *mixed* \$data)

Writes the meta-data to APC

final protected **_initialize** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$key, *mixed* \$table, *mixed* \$schema) inherited from *Phalcon\Mvc\Model\MetaData*

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Mvc\Model\MetaData*

Sets the DependencyInjector container

public **getDI** () inherited from *Phalcon\Mvc\Model\MetaData*

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\StrategyInterface* \$strategy) inherited from *Phalcon\Mvc\Model\MetaData*

Set the meta-data extraction strategy

public **getStrategy** () inherited from *Phalcon\Mvc\Model\MetaData*

Return the strategy to obtain the meta-data

final public **readMetaData** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the complete meta-data for certain model

```
<?php

print_r (
    $metaData->readMetaData (
        new Robots ()
    )
);
```

final public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads meta-data for certain model


```
<?php

print_r(
    $metaData->readMetaDataIndex(
        new Robots(),
        0
    )
);
```

final public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index, *mixed* \$data) inherited from *Phalcon\Mvc\Model\MetaData*

Writes meta-data for certain model using a MODEL_* constant

```
<?php

print_r(
    $metaData->writeColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP,
        [
            "leName" => "name",
        ]
    )
);
```

final public **readColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the ordered/reversed column map for certain model

```
<?php

print_r(
    $metaData->readColumnMap(
        new Robots()
    )
);
```

final public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads column-map information for certain model using a MODEL_* constant

```
<?php

print_r(
    $metaData->readColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP
    )
);
```

public **getAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns table attributes names (fields)

```
<?php

print_r(
```

```
$metaData->getAttributes(  
    new Robots()  
)  
);
```

public **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are part of the primary key

```
<?php  
  
print_r(  
    $metaData->getPrimaryKeyAttributes(  
        new Robots()  
    )  
);
```

public **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are not part of the primary key

```
<?php  
  
print_r(  
    $metaData->getNonPrimaryKeyAttributes(  
        new Robots()  
    )  
);
```

public **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of not null attributes

```
<?php  
  
print_r(  
    $metaData->getNotNullAttributes(  
        new Robots()  
    )  
);
```

public **getDataTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their data types

```
<?php  
  
print_r(  
    $metaData->getDataTypes(  
        new Robots()  
    )  
);
```

public **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes which types are numerical

```
<?php

print_r(
    $metaData->getDataTypesNumeric(
        new Robots()
    )
);
```

public **getIdentityField** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns the name of identity field (if one is present)

```
<?php

print_r(
    $metaData->getIdentityField(
        new Robots()
    )
);
```

public **getBindTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns attributes and their bind data types

```
<?php

print_r(
    $metaData->getBindTypes(
        new Robots()
    )
);
```

public **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r(
    $metaData->getAutomaticCreateAttributes(
        new Robots()
    )
);
```

public **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r(
    $metaData->getAutomaticUpdateAttributes(
        new Robots()
    )
);
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes (
    new Robots (),
    [
        "created_at" => true,
    ]
);
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes (
    new Robots (),
    [
        "modified_at" => true,
    ]
);
```

public **setEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that allow empty string values

```
<?php

$metaData->setEmptyStringAttributes (
    new Robots (),
    [
        "name" => true,
    ]
);
```

public **getEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes allow empty strings

```
<?php

print_r (
    $metaData->getEmptyStringAttributes (
        new Robots ()
    )
);
```

public **getDefaultValues** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes (which have default values) and their default values

```
<?php

print_r(
    $metaData->getDefaultValue(
        new Robots()
    )
);
```

public **getColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns the column map if any

```
<?php

print_r(
    $metaData->getColumnMap(
        new Robots()
    )
);
```

public **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the reverse column map if any

```
<?php

print_r(
    $metaData->getReverseColumnMap(
        new Robots()
    )
);
```

public **hasAttribute** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$attribute) inherited from *Phalcon\Mvc\Model\MetaData*

Check if a model has certain attribute

```
<?php

var_dump(
    $metaData->hasAttribute(
        new Robots(),
        "name"
    )
);
```

public **isEmpty** () inherited from *Phalcon\Mvc\Model\MetaData*

Checks if the internal meta-data container is empty

```
<?php

var_dump(
    $metaData->isEmpty()
);
```

public **reset** () inherited from *Phalcon\Mvc\Model\MetaData*

Resets internal meta-data in order to regenerate it

```
<?php
$metadata->reset();
```

Class `Phalcon\Mvc\Model\MetaData\Files`

extends abstract class `Phalcon\Mvc\Model\MetaData`

implements `Phalcon\Mvc\Model\MetaDataInterface`, `Phalcon\Di\InjectionAwareInterface`

Stores model meta-data in PHP files.

```
<?php
$metadata = new \Phalcon\Mvc\Model\MetaData\Files(
    [
        "metadataDir" => "app/cache/metadata/",
    ]
);
```

Constants

integer `MODELS_ATTRIBUTES`

integer `MODELS_PRIMARY_KEY`

integer `MODELS_NON_PRIMARY_KEY`

integer `MODELS_NOT_NULL`

integer `MODELS_DATA_TYPES`

integer `MODELS_DATA_TYPES_NUMERIC`

integer `MODELS_DATE_AT`

integer `MODELS_DATE_IN`

integer `MODELS_IDENTITY_COLUMN`

integer `MODELS_DATA_TYPES_BIND`

integer `MODELS_AUTOMATIC_DEFAULT_INSERT`

integer `MODELS_AUTOMATIC_DEFAULT_UPDATE`

integer `MODELS_DEFAULT_VALUES`

integer `MODELS_EMPTY_STRING_VALUES`

integer `MODELS_COLUMN_MAP`

integer `MODELS_REVERSE_COLUMN_MAP`

Methods

public `__construct` ([array \$options])

Phalcon\Mvc\Model\MetaData\Files constructor

public *mixed* **read** (*string* \$key)

Reads meta-data from files

public **write** (*string* \$key, *array* \$data)

Writes the meta-data to files

final protected **_initialize** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$key, *mixed* \$table, *mixed* \$schema) inherited from *Phalcon\Mvc\Model\MetaData*

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Mvc\Model\MetaData*

Sets the DependencyInjector container

public **getDI** () inherited from *Phalcon\Mvc\Model\MetaData*

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\StrategyInterface* \$strategy) inherited from *Phalcon\Mvc\Model\MetaData*

Set the meta-data extraction strategy

public **getStrategy** () inherited from *Phalcon\Mvc\Model\MetaData*

Return the strategy to obtain the meta-data

final public **readMetaData** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the complete meta-data for certain model

```
<?php
print_r(
    $metaData->readMetaData(
        new Robots()
    )
);
```

final public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads meta-data for certain model

```
<?php
print_r(
    $metaData->readMetaDataIndex(
        new Robots(),
        0
    )
);
```

final public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index, *mixed* \$data) inherited from *Phalcon\Mvc\Model\MetaData*

Writes meta-data for certain model using a MODEL_* constant

```
<?php
print_r(
```

```
$metaData->writeColumnMapIndex(  
    new Robots(),  
    MetaData::MODELS_REVERSE_COLUMN_MAP,  
    [  
        "leName" => "name",  
    ]  
);
```

final public **readColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Reads the ordered/reversed column map for certain model

```
<?php  
  
print_r(  
    $metaData->readColumnMap(  
        new Robots()  
    )  
);
```

final public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads column-map information for certain model using a MODEL_* constant

```
<?php  
  
print_r(  
    $metaData->readColumnMapIndex(  
        new Robots(),  
        MetaData::MODELS_REVERSE_COLUMN_MAP  
    )  
);
```

public **getAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns table attributes names (fields)

```
<?php  
  
print_r(  
    $metaData->getAttributes(  
        new Robots()  
    )  
);
```

public **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are part of the primary key

```
<?php  
  
print_r(  
    $metaData->getPrimaryKeyAttributes(  
        new Robots()  
    )  
);
```


public **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are not part of the primary key

```
<?php

print_r(
    $metaData->getNonPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of not null attributes

```
<?php

print_r(
    $metaData->getNotNullAttributes(
        new Robots()
    )
);
```

public **getDataTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their data types

```
<?php

print_r(
    $metaData->getDataTypes(
        new Robots()
    )
);
```

public **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes which types are numerical

```
<?php

print_r(
    $metaData->getDataTypesNumeric(
        new Robots()
    )
);
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the name of identity field (if one is present)

```
<?php

print_r(
    $metaData->getIdentityField(
        new Robots()
    )
);
```

public **getBindTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their bind data types

```
<?php
print_r(
    $metaData->getBindTypes (
        new Robots ()
    )
);
```

public **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php
print_r(
    $metaData->getAutomaticCreateAttributes (
        new Robots ()
    )
);
```

public **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php
print_r(
    $metaData->getAutomaticUpdateAttributes (
        new Robots ()
    )
);
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php
$metaData->setAutomaticCreateAttributes (
    new Robots (),
    [
        "created_at" => true,
    ]
);
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php
$metaData->setAutomaticUpdateAttributes (
```

```

new Robots(),
[
    "modified_at" => true,
]
);

```

public **setEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that allow empty string values

```

<?php

$metaData->setEmptyStringAttributes(
    new Robots(),
    [
        "name" => true,
    ]
);

```

public **getEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes allow empty strings

```

<?php

print_r(
    $metaData->getEmptyStringAttributes(
        new Robots()
    )
);

```

public **getDefaultValues** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes (which have default values) and their default values

```

<?php

print_r(
    $metaData->getDefaultValues(
        new Robots()
    )
);

```

public **getColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the column map if any

```

<?php

print_r(
    $metaData->getColumnMap(
        new Robots()
    )
);

```

public **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the reverse column map if any

```
<?php

print_r(
    $metaData->getReverseColumnMap(
        new Robots()
    )
);
```

public **hasAttribute** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$attribute) inherited from *Phalcon\Mvc\Model\MetaData*

Check if a model has certain attribute

```
<?php

var_dump(
    $metaData->hasAttribute(
        new Robots(),
        "name"
    )
);
```

public **isEmpty** () inherited from *Phalcon\Mvc\Model\MetaData*

Checks if the internal meta-data container is empty

```
<?php

var_dump(
    $metaData->isEmpty()
);
```

public **reset** () inherited from *Phalcon\Mvc\Model\MetaData*

Resets internal meta-data in order to regenerate it

```
<?php

$metaData->reset();
```

Class *Phalcon\Mvc\Model\MetaData\Libmemcached*

extends abstract class *Phalcon\Mvc\Model\MetaData*

implements *Phalcon\Mvc\Model\MetaDataInterface*, *Phalcon\Di\InjectionAwareInterface*

Stores model meta-data in the Memcache.

By default meta-data is stored for 48 hours (172800 seconds)

```
<?php

$metaData = new Phalcon\Mvc\Model\Metaadta\Libmemcached(
    [
        "servers" => [
            [
                "host" => "localhost",
```

```

        "port"    => 11211,
        "weight" => 1,
    ],
],
"client" => [
    Memcached::OPT_HASH => Memcached::HASH_MD5,
    Memcached::OPT_PREFIX_KEY => "prefix.",
],
"lifetime" => 3600,
"prefix"    => "my_",
]
);

```

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**

integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**

integer **MODELS_DEFAULT_VALUES**

integer **MODELS_EMPTY_STRING_VALUES**

integer **MODELS_COLUMN_MAP**

integer **MODELS_REVERSE_COLUMN_MAP**

Methods

public **__construct** ([array \$options])

Phalcon\Mvc\Model\MetaData\Libmemcached constructor

public **read** (mixed \$key)

Reads metadata from Memcache

public **write** (mixed \$key, mixed \$data)

Writes the metadata to Memcache

public **reset** ()

Flush Memcache data and resets internal meta-data in order to regenerate it

final protected **_initialize** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$key, *mixed* \$table, *mixed* \$schema) inherited from *Phalcon\Mvc\Model\MetaData*

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Mvc\Model\MetaData*

Sets the DependencyInjector container

public **getDI** () inherited from *Phalcon\Mvc\Model\MetaData*

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\StrategyInterface* \$strategy) inherited from *Phalcon\Mvc\Model\MetaData*

Set the meta-data extraction strategy

public **getStrategy** () inherited from *Phalcon\Mvc\Model\MetaData*

Return the strategy to obtain the meta-data

final public **readMetaData** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the complete meta-data for certain model

```
<?php

print_r(
    $metaData->readMetaData(
        new Robots()
    )
);
```

final public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads meta-data for certain model

```
<?php

print_r(
    $metaData->readMetaDataIndex(
        new Robots(),
        0
    )
);
```

final public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index, *mixed* \$data) inherited from *Phalcon\Mvc\Model\MetaData*

Writes meta-data for certain model using a MODEL_* constant

```
<?php

print_r(
    $metaData->writeColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP,
        [
            "leName" => "name",
        ]
    )
);
```

```

        ]
    )
};

```

final public **readColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
 Reads the ordered/reversed column map for certain model

```

<?php

print_r(
    $metaData->readColumnMap(
        new Robots()
    )
);

```

final public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads column-map information for certain model using a MODEL_* constant

```

<?php

print_r(
    $metaData->readColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP
    )
);

```

public **getAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns table attributes names (fields)

```

<?php

print_r(
    $metaData->getAttributes(
        new Robots()
    )
);

```

public **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are part of the primary key

```

<?php

print_r(
    $metaData->getPrimaryKeyAttributes(
        new Robots()
    )
);

```

public **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are not part of the primary key

```
<?php

print_r(
    $metaData->getNonPrimaryKeyAttributes (
        new Robots ()
    )
);
```

public **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns an array of not null attributes

```
<?php

print_r(
    $metaData->getNotNullAttributes (
        new Robots ()
    )
);
```

public **getDataTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns attributes and their data types

```
<?php

print_r(
    $metaData->getDataTypes (
        new Robots ()
    )
);
```

public **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns attributes which types are numerical

```
<?php

print_r(
    $metaData->getDataTypesNumeric (
        new Robots ()
    )
);
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns the name of identity field (if one is present)

```
<?php

print_r(
    $metaData->getIdentityField (
        new Robots ()
    )
);
```

public **getBindTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns attributes and their bind data types


```
<?php

print_r(
    $metaData->getBindTypes(
        new Robots()
    )
);
```

public **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r(
    $metaData->getAutomaticCreateAttributes(
        new Robots()
    )
);
```

public **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r(
    $metaData->getAutomaticUpdateAttributes(
        new Robots()
    )
);
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes(
    new Robots(),
    [
        "created_at" => true,
    ]
);
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes(
    new Robots(),
    [
        "modified_at" => true,
    ]
);
```

```
    ]
);
```

public **setEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that allow empty string values

```
<?php

$metaData->setEmptyStringAttributes (
    new Robots (),
    [
        "name" => true,
    ]
);
```

public **getEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes allow empty strings

```
<?php

print_r (
    $metaData->getEmptyStringAttributes (
        new Robots ()
    )
);
```

public **getDefaultValues** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes (which have default values) and their default values

```
<?php

print_r (
    $metaData->getDefaultValues (
        new Robots ()
    )
);
```

public **getColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the column map if any

```
<?php

print_r (
    $metaData->getColumnMap (
        new Robots ()
    )
);
```

public **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the reverse column map if any

```
<?php

print_r(
    $metaData->getReverseColumnMap(
        new Robots()
    )
);
```

public **hasAttribute** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$attribute) inherited from *Phalcon\Mvc\Model\MetaData*

Check if a model has certain attribute

```
<?php

var_dump(
    $metaData->hasAttribute(
        new Robots(),
        "name"
    )
);
```

public **isEmpty** () inherited from *Phalcon\Mvc\Model\MetaData*

Checks if the internal meta-data container is empty

```
<?php

var_dump(
    $metaData->isEmpty()
);
```

Class *Phalcon\Mvc\Model\MetaData\Memcache*

extends abstract class *Phalcon\Mvc\Model\MetaData*

implements *Phalcon\Mvc\Model\MetaDataInterface*, *Phalcon\Di\InjectionAwareInterface*

Stores model meta-data in the Memcache.

By default meta-data is stored for 48 hours (172800 seconds)

```
<?php

$metaData = new Phalcon\Mvc\Model\Metadata\Memcache(
    [
        "prefix"      => "my-app-id",
        "lifetime"     => 86400,
        "host"         => "localhost",
        "port"         => 11211,
        "persistent"   => false,
    ]
);
```

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**
integer **MODELS_NON_PRIMARY_KEY**
integer **MODELS_NOT_NULL**
integer **MODELS_DATA_TYPES**
integer **MODELS_DATA_TYPES_NUMERIC**
integer **MODELS_DATE_AT**
integer **MODELS_DATE_IN**
integer **MODELS_IDENTITY_COLUMN**
integer **MODELS_DATA_TYPES_BIND**
integer **MODELS_AUTOMATIC_DEFAULT_INSERT**
integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**
integer **MODELS_DEFAULT_VALUES**
integer **MODELS_EMPTY_STRING_VALUES**
integer **MODELS_COLUMN_MAP**
integer **MODELS_REVERSE_COLUMN_MAP**

Methods

public **__construct** ([array \$options])

Phalcon\Mvc\Model\MetaData\Memcache constructor

public **read** (*mixed* \$key)

Reads metadata from Memcache

public **write** (*mixed* \$key, *mixed* \$data)

Writes the metadata to Memcache

public **reset** ()

Flush Memcache data and resets internal meta-data in order to regenerate it

final protected **_initialize** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$key, *mixed* \$table, *mixed* \$schema) inherited from *Phalcon\Mvc\Model\MetaData*

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Mvc\Model\MetaData*

Sets the DependencyInjector container

public **getDI** () inherited from *Phalcon\Mvc\Model\MetaData*

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\StrategyInterface* \$strategy) inherited from *Phalcon\Mvc\Model\MetaData*

Set the meta-data extraction strategy

public **getStrategy** () inherited from *Phalcon\Mvc\Model\MetaData*

Return the strategy to obtain the meta-data

final public **readMetaData** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the complete meta-data for certain model

```
<?php

print_r(
    $metaData->readMetaData(
        new Robots()
    )
);
```

final public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads meta-data for certain model

```
<?php

print_r(
    $metaData->readMetaDataIndex(
        new Robots(),
        0
    )
);
```

final public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index, *mixed* \$data) inherited from *Phalcon\Mvc\Model\MetaData*

Writes meta-data for certain model using a MODEL_* constant

```
<?php

print_r(
    $metaData->writeColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP,
        [
            "leName" => "name",
        ]
    )
);
```

final public **readColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the ordered/reversed column map for certain model

```
<?php

print_r(
    $metaData->readColumnMap(
        new Robots()
    )
);
```

final public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads column-map information for certain model using a MODEL_* constant

```
<?php

print_r(
    $metaData->readColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP
    )
);
```

public **getAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns table attributes names (fields)

```
<?php

print_r(
    $metaData->getAttributes(
        new Robots()
    )
);
```

public **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are part of the primary key

```
<?php

print_r(
    $metaData->getPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are not part of the primary key

```
<?php

print_r(
    $metaData->getNonPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of not null attributes

```
<?php

print_r(
    $metaData->getNotNullAttributes(
        new Robots()
    )
);
```

public **getDataTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their data types

```
<?php

print_r(
    $metaData->getDataTypes (
        new Robots ()
    )
);
```

public **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes which types are numerical

```
<?php

print_r(
    $metaData->getDataTypesNumeric (
        new Robots ()
    )
);
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the name of identity field (if one is present)

```
<?php

print_r(
    $metaData->getIdentityField (
        new Robots ()
    )
);
```

public **getBindTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their bind data types

```
<?php

print_r(
    $metaData->getBindTypes (
        new Robots ()
    )
);
```

public **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r(
    $metaData->getAutomaticCreateAttributes (
        new Robots ()
    )
);
```

public **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r(
    $metaData->getAutomaticUpdateAttributes (
        new Robots ()
    )
);
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes (
    new Robots (),
    [
        "created_at" => true,
    ]
);
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes (
    new Robots (),
    [
        "modified_at" => true,
    ]
);
```

public **setEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that allow empty string values

```
<?php

$metaData->setEmptyStringAttributes (
    new Robots (),
    [
        "name" => true,
    ]
);
```

public **getEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes allow empty strings


```
<?php

print_r(
    $metaData->getEmptyStringAttributes(
        new Robots()
    )
);
```

public **getDefaultValues** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns attributes (which have default values) and their default values

```
<?php

print_r(
    $metaData->getDefaultValues(
        new Robots()
    )
);
```

public **getColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns the column map if any

```
<?php

print_r(
    $metaData->getColumnMap(
        new Robots()
    )
);
```

public **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the reverse column map if any

```
<?php

print_r(
    $metaData->getReverseColumnMap(
        new Robots()
    )
);
```

public **hasAttribute** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$attribute) inherited from *Phalcon\Mvc\Model\MetaData*

Check if a model has certain attribute

```
<?php

var_dump(
    $metaData->hasAttribute(
        new Robots(),
        "name"
    )
);
```

public **isEmpty** () inherited from *Phalcon\Mvc\Model\MetaData*

Checks if the internal meta-data container is empty

```
<?php
var_dump (
    $metaData->isEmpty ()
);
```

Class *Phalcon\Mvc\Model\MetaData\Memory*

extends abstract class *Phalcon\Mvc\Model\MetaData*

implements *Phalcon\Mvc\Model\MetaDataInterface*, *Phalcon\DI\InjectionAwareInterface*

Stores model meta-data in memory. Data will be erased when the request finishes

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**

integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**

integer **MODELS_DEFAULT_VALUES**

integer **MODELS_EMPTY_STRING_VALUES**

integer **MODELS_COLUMN_MAP**

integer **MODELS_REVERSE_COLUMN_MAP**

Methods

public **__construct** ([array \$options])

Phalcon\Mvc\Model\MetaData\Memory constructor

public array **read** (string \$key)

Reads the meta-data from temporal memory

public **write** (*string* \$key, *array* \$data)

Writes the meta-data to temporal memory

final protected **_initialize** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$key, *mixed* \$table, *mixed* \$schema) inherited from *Phalcon\Mvc\Model\MetaData*

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Mvc\Model\MetaData*

Sets the DependencyInjector container

public **getDI** () inherited from *Phalcon\Mvc\Model\MetaData*

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\StrategyInterface* \$strategy) inherited from *Phalcon\Mvc\Model\MetaData*

Set the meta-data extraction strategy

public **getStrategy** () inherited from *Phalcon\Mvc\Model\MetaData*

Return the strategy to obtain the meta-data

final public **readMetaData** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the complete meta-data for certain model

```
<?php
print_r(
    $metaData->readMetaData(
        new Robots()
    )
);
```

final public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads meta-data for certain model

```
<?php
print_r(
    $metaData->readMetaDataIndex(
        new Robots(),
        0
    )
);
```

final public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index, *mixed* \$data) inherited from *Phalcon\Mvc\Model\MetaData*

Writes meta-data for certain model using a MODEL_* constant

```
<?php
print_r(
    $metaData->writeColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP,
        [
```

```
        "leName" => "name",
    ]
)
);
```

final public **readColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the ordered/reversed column map for certain model

```
<?php
print_r(
    $metaData->readColumnMap(
        new Robots()
    )
);
```

final public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads column-map information for certain model using a MODEL_* constant

```
<?php
print_r(
    $metaData->readColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP
    )
);
```

public **getAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns table attributes names (fields)

```
<?php
print_r(
    $metaData->getAttributes(
        new Robots()
    )
);
```

public **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are part of the primary key

```
<?php
print_r(
    $metaData->getPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are not part of the primary key

```
<?php

print_r(
    $metaData->getNonPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of not null attributes

```
<?php

print_r(
    $metaData->getNotNullAttributes(
        new Robots()
    )
);
```

public **getDataTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their data types

```
<?php

print_r(
    $metaData->getDataTypes(
        new Robots()
    )
);
```

public **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes which types are numerical

```
<?php

print_r(
    $metaData->getDataTypesNumeric(
        new Robots()
    )
);
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the name of identity field (if one is present)

```
<?php

print_r(
    $metaData->getIdentityField(
        new Robots()
    )
);
```

public **getBindTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their bind data types

```
<?php

print_r(
    $metaData->getBindTypes (
        new Robots ()
    )
);
```

public **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r(
    $metaData->getAutomaticCreateAttributes (
        new Robots ()
    )
);
```

public **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r(
    $metaData->getAutomaticUpdateAttributes (
        new Robots ()
    )
);
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes (
    new Robots (),
    [
        "created_at" => true,
    ]
);
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes (
    new Robots (),
    [
        "modified_at" => true,
    ]
);
```

```
    ]
);
```

public **setEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that allow empty string values

```
<?php

$metaData->setEmptyStringAttributes (
    new Robots (),
    [
        "name" => true,
    ]
);
```

public **getEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes allow empty strings

```
<?php

print_r (
    $metaData->getEmptyStringAttributes (
        new Robots ()
    )
);
```

public **getDefaultValues** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes (which have default values) and their default values

```
<?php

print_r (
    $metaData->getDefaultValues (
        new Robots ()
    )
);
```

public **getColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the column map if any

```
<?php

print_r (
    $metaData->getColumnMap (
        new Robots ()
    )
);
```

public **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the reverse column map if any

```
<?php

print_r(
    $metaData->getReverseColumnMap(
        new Robots()
    )
);
```

public **hasAttribute** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$attribute) inherited from *Phalcon\Mvc\Model\MetaData*

Check if a model has certain attribute

```
<?php

var_dump(
    $metaData->hasAttribute(
        new Robots(),
        "name"
    )
);
```

public **isEmpty** () inherited from *Phalcon\Mvc\Model\MetaData*

Checks if the internal meta-data container is empty

```
<?php

var_dump(
    $metaData->isEmpty()
);
```

public **reset** () inherited from *Phalcon\Mvc\Model\MetaData*

Resets internal meta-data in order to regenerate it

```
<?php

$metaData->reset();
```

Class *Phalcon\Mvc\Model\MetaData\Redis*

extends abstract class *Phalcon\Mvc\Model\MetaData*

implements *Phalcon\Mvc\Model\MetaDataInterface*, *Phalcon\DI\InjectionAwareInterface*

Stores model meta-data in the Redis.

By default meta-data is stored for 48 hours (172800 seconds)

```
<?php

use Phalcon\Mvc\Model\Metadata\Redis;

$metaData = new Redis(
    [
        "host"      => "127.0.0.1",
        "port"      => 6379,
```



```

        "persistent" => 0,
        "statsKey"   => "_PHCM_MM",
        "lifetime"   => 172800,
        "index"      => 2,
    ]
);

```

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**

integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**

integer **MODELS_DEFAULT_VALUES**

integer **MODELS_EMPTY_STRING_VALUES**

integer **MODELS_COLUMN_MAP**

integer **MODELS_REVERSE_COLUMN_MAP**

Methods

public **__construct** ([array \$options])

Phalcon\Mvc\Model\MetaData\Redis constructor

public **read** (*mixed* \$key)

Reads metadata from Redis

public **write** (*mixed* \$key, *mixed* \$data)

Writes the metadata to Redis

public **reset** ()

Flush Redis data and resets internal meta-data in order to regenerate it

final protected **_initialize** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$key, *mixed* \$table, *mixed* \$schema) inherited from *Phalcon\Mvc\Model\MetaData*

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Mvc\Model\MetaData*

Sets the DependencyInjector container

public **getDI** () inherited from *Phalcon\Mvc\Model\MetaData*

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\StrategyInterface* \$strategy) inherited from *Phalcon\Mvc\Model\MetaData*

Set the meta-data extraction strategy

public **getStrategy** () inherited from *Phalcon\Mvc\Model\MetaData*

Return the strategy to obtain the meta-data

final public **readMetaData** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the complete meta-data for certain model

```
<?php
print_r(
    $metaData->readMetaData(
        new Robots()
    )
);
```

final public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads meta-data for certain model

```
<?php
print_r(
    $metaData->readMetaDataIndex(
        new Robots(),
        0
    )
);
```

final public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index, *mixed* \$data) inherited from *Phalcon\Mvc\Model\MetaData*

Writes meta-data for certain model using a MODEL_* constant

```
<?php
print_r(
    $metaData->writeColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP,
        [
            "leName" => "name",
        ]
    )
);
```

final public **readColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the ordered/reversed column map for certain model

```
<?php

print_r(
    $metaData->readColumnMap(
        new Robots()
    )
);
```

final public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads column-map information for certain model using a MODEL_* constant

```
<?php

print_r(
    $metaData->readColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP
    )
);
```

public **getAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns table attributes names (fields)

```
<?php

print_r(
    $metaData->getAttributes(
        new Robots()
    )
);
```

public **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are part of the primary key

```
<?php

print_r(
    $metaData->getPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are not part of the primary key

```
<?php

print_r(
    $metaData->getNonPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of not null attributes

```
<?php

print_r(
    $metaData->getNotNullAttributes(
        new Robots()
    )
);
```

public **getDataTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their data types

```
<?php

print_r(
    $metaData->getDataTypes(
        new Robots()
    )
);
```

public **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes which types are numerical

```
<?php

print_r(
    $metaData->getDataTypesNumeric(
        new Robots()
    )
);
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the name of identity field (if one is present)

```
<?php

print_r(
    $metaData->getIdentityField(
        new Robots()
    )
);
```

public **getBindTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their bind data types

```
<?php

print_r(
    $metaData->getBindTypes(
        new Robots()
    )
);
```

public **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r(
    $metaData->getAutomaticCreateAttributes (
        new Robots ()
    )
);
```

public **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r(
    $metaData->getAutomaticUpdateAttributes (
        new Robots ()
    )
);
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes (
    new Robots (),
    [
        "created_at" => true,
    ]
);
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes (
    new Robots (),
    [
        "modified_at" => true,
    ]
);
```

public **setEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that allow empty string values

```
<?php

$metaData->setEmptyStringAttributes (
    new Robots (),
    [
        "name" => true,
    ]
);
```

public **getEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes allow empty strings

```
<?php

print_r (
    $metaData->getEmptyStringAttributes (
        new Robots ()
    )
);
```

public **getDefaultValues** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes (which have default values) and their default values

```
<?php

print_r (
    $metaData->getDefaultValues (
        new Robots ()
    )
);
```

public **getColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the column map if any

```
<?php

print_r (
    $metaData->getColumnMap (
        new Robots ()
    )
);
```

public **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the reverse column map if any

```
<?php

print_r (
    $metaData->getReverseColumnMap (
        new Robots ()
    )
);
```

public **hasAttribute** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$attribute) inherited from *Phalcon\Mvc\Model\MetaData*

Check if a model has certain attribute

```
<?php
var_dump(
    $metaData->hasAttribute(
        new Robots(),
        "name"
    )
);
```

public **isEmpty** () inherited from *Phalcon\Mvc\Model\MetaData*

Checks if the internal meta-data container is empty

```
<?php
var_dump(
    $metaData->isEmpty()
);
```

Class *Phalcon\Mvc\Model\MetaData\Session*

extends abstract class *Phalcon\Mvc\Model\MetaData*

implements *Phalcon\Mvc\Model\MetaDataInterface*, *Phalcon\DI\InjectionAwareInterface*

Stores model meta-data in session. Data will erased when the session finishes. Meta-data are permanent while the session is active.

You can query the meta-data by printing \$_SESSION['\$PMM\$']

```
<?php
$metaData = new \Phalcon\Mvc\Model\Metadatas\Session(
    [
        "prefix" => "my-app-id",
    ]
);
```

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**

integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**

integer **MODELS_DEFAULT_VALUES**

integer **MODELS_EMPTY_STRING_VALUES**

integer **MODELS_COLUMN_MAP**

integer **MODELS_REVERSE_COLUMN_MAP**

Methods

public **__construct** ([array \$options])

Phalcon\Mvc\Model\MetaData\Session constructor

public array **read** (string \$key)

Reads meta-data from \$_SESSION

public **write** (string \$key, array \$data)

Writes the meta-data to \$_SESSION

final protected **_initialize** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$key, *mixed* \$table, *mixed* \$schema) inherited from *Phalcon\Mvc\Model\MetaData*

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Mvc\Model\MetaData*

Sets the DependencyInjector container

public **getDI** () inherited from *Phalcon\Mvc\Model\MetaData*

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\StrategyInterface* \$strategy) inherited from *Phalcon\Mvc\Model\MetaData*

Set the meta-data extraction strategy

public **getStrategy** () inherited from *Phalcon\Mvc\Model\MetaData*

Return the strategy to obtain the meta-data

final public **readMetaData** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the complete meta-data for certain model

```
<?php
print_r(
    $metaData->readMetaData(
        new Robots()
    )
);
```


final public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads meta-data for certain model

```
<?php

print_r(
    $metaData->readMetaDataIndex(
        new Robots(),
        0
    )
);
```

final public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index, *mixed* \$data) inherited from *Phalcon\Mvc\Model\MetaData*

Writes meta-data for certain model using a MODEL_* constant

```
<?php

print_r(
    $metaData->writeColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP,
        [
            "leName" => "name",
        ]
    )
);
```

final public **readColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the ordered/reversed column map for certain model

```
<?php

print_r(
    $metaData->readColumnMap(
        new Robots()
    )
);
```

final public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads column-map information for certain model using a MODEL_* constant

```
<?php

print_r(
    $metaData->readColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP
    )
);
```

public **getAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns table attributes names (fields)

```
<?php

print_r(
    $metaData->getAttributes(
        new Robots()
    )
);
```

public **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are part of the primary key

```
<?php

print_r(
    $metaData->getPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are not part of the primary key

```
<?php

print_r(
    $metaData->getNonPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of not null attributes

```
<?php

print_r(
    $metaData->getNotNullAttributes(
        new Robots()
    )
);
```

public **getDataTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their data types

```
<?php

print_r(
    $metaData->getDataTypes(
        new Robots()
    )
);
```

public **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes which types are numerical

```
<?php

print_r(
    $metaData->getDataTypesNumeric(
        new Robots()
    )
);
```

public **getIdentityField** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the name of identity field (if one is present)

```
<?php

print_r(
    $metaData->getIdentityField(
        new Robots()
    )
);
```

public **getBindTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their bind data types

```
<?php

print_r(
    $metaData->getBindTypes(
        new Robots()
    )
);
```

public **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r(
    $metaData->getAutomaticCreateAttributes(
        new Robots()
    )
);
```

public **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r(
    $metaData->getAutomaticUpdateAttributes(
        new Robots()
    )
);
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes (
    new Robots (),
    [
        "created_at" => true,
    ]
);
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes (
    new Robots (),
    [
        "modified_at" => true,
    ]
);
```

public **setEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that allow empty string values

```
<?php

$metaData->setEmptyStringAttributes (
    new Robots (),
    [
        "name" => true,
    ]
);
```

public **getEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes allow empty strings

```
<?php

print_r (
    $metaData->getEmptyStringAttributes (
        new Robots ()
    )
);
```

public **getDefaultValues** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes (which have default values) and their default values

```
<?php

print_r(
    $metaData->getDefaultValue(
        new Robots()
    )
);
```

public **getColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns the column map if any

```
<?php

print_r(
    $metaData->getColumnMap(
        new Robots()
    )
);
```

public **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the reverse column map if any

```
<?php

print_r(
    $metaData->getReverseColumnMap(
        new Robots()
    )
);
```

public **hasAttribute** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$attribute) inherited from *Phalcon\Mvc\Model\MetaData*

Check if a model has certain attribute

```
<?php

var_dump(
    $metaData->hasAttribute(
        new Robots(),
        "name"
    )
);
```

public **isEmpty** () inherited from *Phalcon\Mvc\Model\MetaData*

Checks if the internal meta-data container is empty

```
<?php

var_dump(
    $metaData->isEmpty()
);
```

public **reset** () inherited from *Phalcon\Mvc\Model\MetaData*

Resets internal meta-data in order to regenerate it

```
<?php
$metadata->reset();
```

Class `Phalcon\Mvc\Model\Metadata\Strategy\Annotations`

implements `Phalcon\Mvc\Model\Metadata\StrategyInterface`

Methods

final public **getMetadata** (*Phalcon\Mvc\ModelInterface* \$model, *Phalcon\DiInterface* \$dependencyInjector)

The meta-data is obtained by reading the column descriptions from the database information schema

final public **getColumnMaps** (*Phalcon\Mvc\ModelInterface* \$model, *Phalcon\DiInterface* \$dependencyInjector)

Read the model's column map, this can't be inferred

Class `Phalcon\Mvc\Model\Metadata\Strategy\Introspection`

implements `Phalcon\Mvc\Model\Metadata\StrategyInterface`

Queries the table meta-data in order to introspect the model's metadata

Methods

final public **getMetadata** (*Phalcon\Mvc\ModelInterface* \$model, *Phalcon\DiInterface* \$dependencyInjector)

The meta-data is obtained by reading the column descriptions from the database information schema

final public **getColumnMaps** (*Phalcon\Mvc\ModelInterface* \$model, *Phalcon\DiInterface* \$dependencyInjector)

Read the model's column map, this can't be inferred

Class `Phalcon\Mvc\Model\Metadata\Xcache`

extends abstract class `Phalcon\Mvc\Model\Metadata`

implements `Phalcon\Mvc\Model\MetadataInterface`, `Phalcon\Di\InjectionAwareInterface`

Stores model meta-data in the XCache cache. Data will be erased if the web server is restarted

By default meta-data is stored for 48 hours (172800 seconds)

You can query the meta-data by printing `xcache_get('PMM')` or `xcache_get('PMMmy-app-id')`

```
<?php
$metadata = new Phalcon\Mvc\Model\Metadata\Xcache (
    [
        "prefix"    => "my-app-id",
        "lifetime"  => 86400,
    ]
);
```

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**

integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**

integer **MODELS_DEFAULT_VALUES**

integer **MODELS_EMPTY_STRING_VALUES**

integer **MODELS_COLUMN_MAP**

integer **MODELS_REVERSE_COLUMN_MAP**

Methods

public **__construct** ([array \$options])

Phalcon\Mvc\Model\MetaData\Xcache constructor

public array **read** (string \$key)

Reads metadata from XCache

public **write** (string \$key, array \$data)

Writes the metadata to XCache

final protected **_initialize** (Phalcon\Mvc\ModelInterface \$model, mixed \$key, mixed \$table, mixed \$schema) inherited from [Phalcon\Mvc\Model\MetaData](#)

Initialize the metadata for certain table

public **setDI** (Phalcon\DiInterface \$dependencyInjector) inherited from [Phalcon\Mvc\Model\MetaData](#)

Sets the DependencyInjector container

public **getDI** () inherited from [Phalcon\Mvc\Model\MetaData](#)

Returns the DependencyInjector container

public **setStrategy** (Phalcon\Mvc\Model\MetaData\StrategyInterface \$strategy) inherited from [Phalcon\Mvc\Model\MetaData](#)

Set the meta-data extraction strategy

public **getStrategy** () inherited from [Phalcon\Mvc\Model\MetaData](#)

Return the strategy to obtain the meta-data

final public **readMetaData** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the complete meta-data for certain model

```
<?php

print_r(
    $metaData->readMetaData(
        new Robots()
    )
);
```

final public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads meta-data for certain model

```
<?php

print_r(
    $metaData->readMetaDataIndex(
        new Robots(),
        0
    )
);
```

final public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index, *mixed* \$data) inherited from *Phalcon\Mvc\Model\MetaData*

Writes meta-data for certain model using a MODEL_* constant

```
<?php

print_r(
    $metaData->writeColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP,
        [
            "leName" => "name",
        ]
    )
);
```

final public **readColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Reads the ordered/reversed column map for certain model

```
<?php

print_r(
    $metaData->readColumnMap(
        new Robots()
    )
);
```

final public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$index) inherited from *Phalcon\Mvc\Model\MetaData*

Reads column-map information for certain model using a MODEL_* constant


```
<?php

print_r(
    $metaData->readColumnMapIndex(
        new Robots(),
        MetaData::MODELS_REVERSE_COLUMN_MAP
    )
);
```

public **getAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns table attributes names (fields)

```
<?php

print_r(
    $metaData->getAttributes(
        new Robots()
    )
);
```

public **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are part of the primary key

```
<?php

print_r(
    $metaData->getPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of fields which are not part of the primary key

```
<?php

print_r(
    $metaData->getNonPrimaryKeyAttributes(
        new Robots()
    )
);
```

public **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns an array of not null attributes

```
<?php

print_r(
    $metaData->getNotNullAttributes(
        new Robots()
    )
);
```

public **getDataTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their data types

```
<?php

print_r(
    $metaData->getDataTypes (
        new Robots ()
    )
);
```

public **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes which types are numerical

```
<?php

print_r(
    $metaData->getDataTypesNumeric (
        new Robots ()
    )
);
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the name of identity field (if one is present)

```
<?php

print_r(
    $metaData->getIdentityField (
        new Robots ()
    )
);
```

public **getBindTypes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes and their bind data types

```
<?php

print_r(
    $metaData->getBindTypes (
        new Robots ()
    )
);
```

public **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r(
    $metaData->getAutomaticCreateAttributes (
        new Robots ()
    )
);
```

public **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r(
    $metaData->getAutomaticUpdateAttributes (
        new Robots ()
    )
);
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes (
    new Robots (),
    [
        "created_at" => true,
    ]
);
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes (
    new Robots (),
    [
        "modified_at" => true,
    ]
);
```

public **setEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model, array \$attributes) inherited from *Phalcon\Mvc\Model\MetaData*

Set the attributes that allow empty string values

```
<?php

$metaData->setEmptyStringAttributes (
    new Robots (),
    [
        "name" => true,
    ]
);
```

public **getEmptyStringAttributes** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns attributes allow empty strings

```
<?php

print_r(
    $metaData->getEmptyStringAttributes(
        new Robots()
    )
);
```

public **getDefaultValues** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns attributes (which have default values) and their default values

```
<?php

print_r(
    $metaData->getDefaultValues(
        new Robots()
    )
);
```

public **getColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*
Returns the column map if any

```
<?php

print_r(
    $metaData->getColumnMap(
        new Robots()
    )
);
```

public **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* \$model) inherited from *Phalcon\Mvc\Model\MetaData*

Returns the reverse column map if any

```
<?php

print_r(
    $metaData->getReverseColumnMap(
        new Robots()
    )
);
```

public **hasAttribute** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$attribute) inherited from *Phalcon\Mvc\Model\MetaData*

Check if a model has certain attribute

```
<?php

var_dump(
    $metaData->hasAttribute(
        new Robots(),
        "name"
    )
);
```

public **isEmpty** () inherited from *Phalcon\Mvc\Model\MetaData*

Checks if the internal meta-data container is empty

```
<?php
var_dump (
    $metaData->isEmpty ()
);
```

public **reset** () inherited from *Phalcon\Mvc\Model\MetaData*

Resets internal meta-data in order to regenerate it

```
<?php
$metaData->reset ();
```

Class *Phalcon\Mvc\Model\Query*

implements *Phalcon\Mvc\Model\QueryInterface*, *Phalcon\Di\InjectionAwareInterface*

This class takes a PHQL intermediate representation and executes it.

```
<?php
$phql = "SELECT c.price*0.16 AS taxes, c.* FROM Cars AS c JOIN Brands AS b
        WHERE b.name = :name: ORDER BY c.name";

$result = $manager->executeQuery (
    $phql,
    [
        "name" => "Lamborghini",
    ]
);

foreach ($result as $row) {
    echo "Name: ", $row->cars->name, "\n";
    echo "Price: ", $row->cars->price, "\n";
    echo "Taxes: ", $row->taxes, "\n";
}
```

Constants

integer **TYPE_SELECT**

integer **TYPE_INSERT**

integer **TYPE_UPDATE**

integer **TYPE_DELETE**

Methods

public **__construct** ([*string* \$phql], [*Phalcon\DiInterface* \$dependencyInjector], [*mixed* \$options])

Phalcon\Mvc\Model\Query constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injection container

public **getDI** ()

Returns the dependency injection container

public **setUniqueRow** (*mixed* \$uniqueRow)

Tells to the query if only the first row in the resultset must be returned

public **getUniqueRow** ()

Check if the query is programmed to get only the first row in the resultset

final protected **_getQualified** (*array* \$expr)

Replaces the model's name to its source name in a qualified-name expression

final protected **_getCallArgument** (*array* \$argument)

Resolves an expression in a single call argument

final protected **_getCaseExpression** (*array* \$expr)

Resolves an expression in a single call argument

final protected **_getFunctionCall** (*array* \$expr)

Resolves an expression in a single call argument

final protected *string* **_getExpression** (*array* \$expr, [*boolean* \$quoting])

Resolves an expression from its intermediate code into a string

final protected **_getSelectColumn** (*array* \$column)

Resolves a column from its intermediate representation into an array used to determine if the resultset produced is simple or complex

final protected *string* **_getTable** (*Phalcon\Mvc\Model\ManagerInterface* \$manager, *array* \$qualifiedName)

Resolves a table in a SELECT statement checking if the model exists

final protected **_getJoin** (*Phalcon\Mvc\Model\ManagerInterface* \$manager, *mixed* \$join)

Resolves a JOIN clause checking if the associated models exist

final protected *string* **_getJoinType** (*array* \$join)

Resolves a JOIN type

final protected *array* **_getSingleJoin** (*string* \$joinType, *string* \$joinSource, *string* \$modelAlias, *string* \$joinAlias, *Phalcon\Mvc\Model\RelationInterface* \$relation)

Resolves joins involving has-one/belongs-to/has-many relations

final protected *array* **_getMultiJoin** (*string* \$joinType, *string* \$joinSource, *string* \$modelAlias, *string* \$joinAlias, *Phalcon\Mvc\Model\RelationInterface* \$relation)

Resolves joins involving many-to-many relations

final protected *array* **_getJoins** (*array* \$select)

Processes the JOINS in the query returning an internal representation for the database dialect

final protected *array* **_getOrderClause** (*array* | *string* \$order)

Returns a processed order clause for a SELECT statement

final protected **_getGroupClause** (*array* \$group)

Returns a processed group clause for a SELECT statement

final protected **_getLimitClause** (*array* \$limitClause)

Returns a processed limit clause for a SELECT statement

final protected **_prepareSelect** (*[mixed]* \$ast, *[mixed]* \$merge)

Analyzes a SELECT intermediate code and produces an array to be executed later

final protected **_prepareInsert** ()

Analyzes an INSERT intermediate code and produces an array to be executed later

final protected **_prepareUpdate** ()

Analyzes an UPDATE intermediate code and produces an array to be executed later

final protected **_prepareDelete** ()

Analyzes a DELETE intermediate code and produces an array to be executed later

public **parse** ()

Parses the intermediate code produced by `Phalcon\Mvc\Model\Query\Lang` generating another intermediate representation that could be executed by `Phalcon\Mvc\Model\Query`

public **getCache** ()

Returns the current cache backend instance

final protected **_executeSelect** (*mixed* \$intermediate, *mixed* \$bindParam, *mixed* \$bindTypes, *[mixed]* \$simulate)

Executes the SELECT intermediate representation producing a `Phalcon\Mvc\Model\Resultset`

final protected *Phalcon\Mvc\Model\Query\StatusInterface* **_executeInsert** (*array* \$intermediate, *array* \$bindParam, *array* \$bindTypes)

Executes the INSERT intermediate representation producing a `Phalcon\Mvc\Model\Query\Status`

final protected *Phalcon\Mvc\Model\Query\StatusInterface* **_executeUpdate** (*array* \$intermediate, *array* \$bindParam, *array* \$bindTypes)

Executes the UPDATE intermediate representation producing a `Phalcon\Mvc\Model\Query\Status`

final protected *Phalcon\Mvc\Model\Query\StatusInterface* **_executeDelete** (*array* \$intermediate, *array* \$bindParam, *array* \$bindTypes)

Executes the DELETE intermediate representation producing a `Phalcon\Mvc\Model\Query\Status`

final protected *Phalcon\Mvc\Model\ResultsetInterface* **_getRelatedRecords** (*Phalcon\Mvc\ModelInterface* \$model, *array* \$intermediate, *array* \$bindParam, *array* \$bindTypes)

Query the records on which the UPDATE/DELETE operation will be done

public *mixed* **execute** (*[array]* \$bindParam, *[array]* \$bindTypes)

Executes a parsed PHQL statement

public *Phalcon\Mvc\ModelInterface* **getSingleResult** (*[array]* \$bindParam, *[array]* \$bindTypes)

Executes the query returning the first result

public **setType** (*mixed* \$type)

Sets the type of PHQL statement to be executed

public **getType** ()

Gets the type of PHQL statement executed

public **setBindParams** (array \$bindParams, [mixed \$merge])

Set default bind parameters

public array **getBindParams** ()

Returns default bind params

public **setBindTypes** (array \$bindTypes, [mixed \$merge])

Set default bind parameters

public **setSharedLock** ([mixed \$sharedLock])

Set SHARED LOCK clause

public array **getBindTypes** ()

Returns default bind types

public **setIntermediate** (array \$intermediate)

Allows to set the IR to be executed

public array **getIntermediate** ()

Returns the intermediate representation of the PHQL statement

public **cache** (mixed \$cacheOptions)

Sets the cache parameters of the query

public **getCacheOptions** ()

Returns the current cache options

public **getSql** ()

Returns the SQL to be generated by the internal PHQL (only works in SELECT statements)

public static **clean** ()

Destroys the internal PHQL cache

Class **Phalcon\Mvc\Model\Query\Builder**

*implements **Phalcon\Mvc\Model\Query\BuilderInterface**, **Phalcon\DI\InjectionAwareInterface***

Helps to create PHQL queries using an OO interface

```
<?php
$params = [
    "models"      => ["Users"],
    "columns"     => ["id", "name", "status"],
    "conditions" => [
        [
            "created > :min: AND created < :max:",
            [
                "min" => "2013-01-01",
                "max" => "2014-01-01",
            ],
        ],
    ],
];
```



```

        [
            "min" => PDO::PARAM_STR,
            "max" => PDO::PARAM_STR,
        ],
    ],
    // or "conditions" => "created > '2013-01-01' AND created < '2014-01-01'",
    "group"      => ["id", "name"],
    "having"     => "name = 'Kamil'",
    "order"      => ["name", "id"],
    "limit"      => 20,
    "offset"     => 20,
    // or "limit" => [20, 20],
];

$queryBuilder = new \Phalcon\Mvc\Model\Query\Builder($params);

```

Constants

string **OPERATOR_OR**

string **OPERATOR_AND**

Methods

public **__construct** ([*mixed* \$params], [*Phalcon\DiInterface* \$dependencyInjector])

Phalcon\Mvc\Model\Query\Builder constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the DependencyInjector container

public **getDI** ()

Returns the DependencyInjector container

public **distinct** (*mixed* \$distinct)

Sets SELECT DISTINCT / SELECT ALL flag

```

<?php
$builder->distinct("status");
$builder->distinct(null);

```

public **getDistinct** ()

Returns SELECT DISTINCT / SELECT ALL flag

public **columns** (*mixed* \$columns)

Sets the columns to be queried

```

<?php
$builder->columns("id, name");

$builder->columns(

```

```
[
    "id",
    "name",
]
);

$builder->columns(
    [
        "name",
        "number" => "COUNT(*)",
    ]
);
```

public *string* | *array* **getColumns** ()

Return the columns to be queried

public **from** (*mixed* \$models)

Sets the models who makes part of the query

```
<?php

$builder->from("Robots");

$builder->from(
    [
        "Robots",
        "RobotsParts",
    ]
);

$builder->from(
    [
        "r"  => "Robots",
        "rp" => "RobotsParts",
    ]
);
```

public **addFrom** (*mixed* \$model, [*mixed* \$alias], [*mixed* \$with])

Add a model to take part of the query

```
<?php

// Load data from models Robots
$builder->addFrom("Robots");

// Load data from model 'Robots' using 'r' as alias in PHQL
$builder->addFrom("Robots", "r");

// Load data from model 'Robots' using 'r' as alias in PHQL
// and eager load model 'RobotsParts'
$builder->addFrom("Robots", "r", "RobotsParts");

// Load data from model 'Robots' using 'r' as alias in PHQL
// and eager load models 'RobotsParts' and 'Parts'
$builder->addFrom(
    "Robots",
```

```

        "r",
        [
            "RobotsParts",
            "Parts",
        ]
    );

```

public *string* | *array* **getFrom** ()

Return the models who makes part of the query

public *Phalcon\Mvc\Model\Query\Builder* **join** (*string* \$model, [*string* \$conditions], [*string* \$alias], [*string* \$type])

Adds an INNER join to the query

```

<?php

// Inner Join model 'Robots' with automatic conditions and alias
$builder->join("Robots");

// Inner Join model 'Robots' specifying conditions
$builder->join("Robots", "Robots.id = RobotsParts.robots_id");

// Inner Join model 'Robots' specifying conditions and alias
$builder->join("Robots", "r.id = RobotsParts.robots_id", "r");

// Left Join model 'Robots' specifying conditions, alias and type of join
$builder->join("Robots", "r.id = RobotsParts.robots_id", "r", "LEFT");

```

public *Phalcon\Mvc\Model\Query\Builder* **innerJoin** (*string* \$model, [*string* \$conditions], [*string* \$alias])

Adds an INNER join to the query

```

<?php

// Inner Join model 'Robots' with automatic conditions and alias
$builder->innerJoin("Robots");

// Inner Join model 'Robots' specifying conditions
$builder->innerJoin("Robots", "Robots.id = RobotsParts.robots_id");

// Inner Join model 'Robots' specifying conditions and alias
$builder->innerJoin("Robots", "r.id = RobotsParts.robots_id", "r");

```

public *Phalcon\Mvc\Model\Query\Builder* **leftJoin** (*string* \$model, [*string* \$conditions], [*string* \$alias])

Adds a LEFT join to the query

```

<?php

$builder->leftJoin("Robots", "r.id = RobotsParts.robots_id", "r");

```

public *Phalcon\Mvc\Model\Query\Builder* **rightJoin** (*string* \$model, [*string* \$conditions], [*string* \$alias])

Adds a RIGHT join to the query

```

<?php

$builder->rightJoin("Robots", "r.id = RobotsParts.robots_id", "r");

```

public array **getJoins** ()

Return join parts of the query

public *Phalcon\Mvc\Model\Query\Builder* **where** (*mixed* \$conditions, [*array* \$bindParam], [*array* \$bindTypes])

Sets the query conditions

```
<?php
$builder->where(100);

$builder->where("name = 'Peter'");

$builder->where(
    "name = :name: AND id > :id:",
    [
        "name" => "Peter",
        "id"   => 100,
    ]
);
```

public *Phalcon\Mvc\Model\Query\Builder* **andWhere** (*string* \$conditions, [*array* \$bindParam], [*array* \$bindTypes])

Appends a condition to the current conditions using a AND operator

```
<?php
$builder->andWhere("name = 'Peter'");

$builder->andWhere(
    "name = :name: AND id > :id:",
    [
        "name" => "Peter",
        "id"   => 100,
    ]
);
```

public *Phalcon\Mvc\Model\Query\Builder* **orWhere** (*string* \$conditions, [*array* \$bindParam], [*array* \$bindTypes])

Appends a condition to the current conditions using an OR operator

```
<?php
$builder->orWhere("name = 'Peter'");

$builder->orWhere(
    "name = :name: AND id > :id:",
    [
        "name" => "Peter",
        "id"   => 100,
    ]
);
```

public **betweenWhere** (*mixed* \$expr, *mixed* \$minimum, *mixed* \$maximum, [*mixed* \$operator])

Appends a BETWEEN condition to the current conditions

```
<?php
$builder->betweenWhere("price", 100.25, 200.50);
```

public **notBetweenWhere** (*mixed* \$expr, *mixed* \$minimum, *mixed* \$maximum, [*mixed* \$operator])

Appends a NOT BETWEEN condition to the current conditions

```
<?php
$builder->notBetweenWhere("price", 100.25, 200.50);
```

public **inWhere** (*mixed* \$expr, *array* \$values, [*mixed* \$operator])

Appends an IN condition to the current conditions

```
<?php
$builder->inWhere("id", [1, 2, 3]);
```

public **notInWhere** (*mixed* \$expr, *array* \$values, [*mixed* \$operator])

Appends a NOT IN condition to the current conditions

```
<?php
$builder->notInWhere("id", [1, 2, 3]);
```

public *string* | *array* **getWhere** ()

Return the conditions for the query

public *Phalcon\Mvc\Model\Query\Builder* **orderBy** (*string* | *array* \$orderBy)

Sets an ORDER BY condition clause

```
<?php
$builder->orderBy("Robots.name");
$builder->orderBy(["1", "Robots.name"]);
```

public *string* | *array* **getOrderBy** ()

Returns the set ORDER BY clause

public **having** (*mixed* \$having)

Sets a HAVING condition clause. You need to escape PHQL reserved words using [and] delimiters

```
<?php
$builder->having("SUM(Robots.price) > 0");
```

public **forUpdate** (*mixed* \$forUpdate)

Sets a FOR UPDATE clause

```
<?php
$builder->forUpdate(true);
```

public *string* | *array* **getHaving** ()

Return the current having clause

public **limit** (*mixed* \$limit, [*mixed* \$offset])

Sets a LIMIT clause, optionally an offset clause

```
<?php

$builder->limit(100);
$builder->limit(100, 20);
$builder->limit("100", "20");
```

public *string* | *array* **getLimit** ()

Returns the current LIMIT clause

public **offset** (*mixed* \$offset)

Sets an OFFSET clause

```
<?php

$builder->offset(30);
```

public *string* | *array* **getOffset** ()

Returns the current OFFSET clause

public *Phalcon\Mvc\Model\Query\Builder* **groupBy** (*string* | *array* \$group)

Sets a GROUP BY clause

```
<?php

$builder->groupBy (
    [
        "Robots.name",
    ]
);
```

public *string* **getGroupBy** ()

Returns the GROUP BY clause

final public *string* **getPhql** ()

Returns a PHQL statement built based on the builder parameters

public **getQuery** ()

Returns the query built

final public **autoescape** (*mixed* \$identifier)

Automatically escapes identifiers but only if they need to be escaped.

Abstract class *Phalcon\Mvc\Model\Query\Lang*

PHQL is implemented as a parser (written in C) that translates syntax in that of the target RDBMS. It allows Phalcon to offer a unified SQL language to the developer, while internally doing all the work of translating PHQL instructions to the most optimal SQL instructions depending on the RDBMS type associated with a model.

To achieve the highest performance possible, we wrote a parser that uses the same technology as SQLite. This technology provides a small in-memory parser with a very low memory footprint that is also thread-safe.

```
<?php

$intermediate = Phalcon\Mvc\Model\Query\Lang::parsePHQL("SELECT r.* FROM Robots r_
↪LIMIT 10");
```

Methods

public static *string* **parsePHQL** (*string* \$phql)

Parses a PHQL statement returning an intermediate representation (IR)

Class Phalcon\Mvc\Model\Query\Status

implements *Phalcon\Mvc\Model\Query\StatusInterface*

This class represents the status returned by a PHQL statement like INSERT, UPDATE or DELETE. It offers context information and the related messages produced by the model which finally executes the operations when it fails

```
<?php

$phql = "UPDATE Robots SET name = :name:, type = :type:, year = :year: WHERE id = :id:
↪";

$status = $app->modelsManager->executeQuery(
    $phql,
    [
        "id"    => 100,
        "name"  => "Astroy Boy",
        "type"  => "mechanical",
        "year"  => 1959,
    ]
);

// Check if the update was successful
if ($status->success() === true) {
    echo "OK";
}
```

Methods

public **__construct** (*mixed* \$success, [*Phalcon\Mvc\ModelInterface* \$model])

public **getModel** ()

Returns the model that executed the action

public **getMessages** ()

Returns the messages produced because of a failed operation

public **success** ()

Allows to check if the executed operation was successful

Class Phalcon\Mvc\Model\Relation

implements *Phalcon\Mvc\Model\RelationInterface*

This class represents a relationship between two models

Constants

integer **BELONGS_TO**

integer **HAS_ONE**

integer **HAS_MANY**

integer **HAS_ONE_THROUGH**

integer **HAS_MANY_THROUGH**

integer **NO_ACTION**

integer **ACTION_RESTRICT**

integer **ACTION_CASCADE**

Methods

public **__construct** (*int* \$type, *string* \$referencedModel, *string* | *array* \$fields, *string* | *array* \$referencedFields, [*array* \$options])

Phalcon\Mvc\Model\Relation constructor

public **setIntermediateRelation** (*string* | *array* \$intermediateFields, *string* \$intermediateModel, *string* \$intermediateReferencedFields)

Sets the intermediate model data for has-*^{*}-through relations

public **getType** ()

Returns the relation type

public **getReferencedModel** ()

Returns the referenced model

public *string* | *array* **getFields** ()

Returns the fields

public *string* | *array* **getReferencedFields** ()

Returns the referenced fields

public *string* | *array* **getOptions** ()

Returns the options

public **getOption** (*mixed* \$name)

Returns an option by the specified name If the option doesn't exist null is returned

public **isForeignKey** ()

Check whether the relation act as a foreign key

public *string* | *array* **getForeignKey** ()

Returns the foreign key configuration

public array **getParams** ()

Returns parameters that must be always used when the related records are obtained

public **isThrough** ()

Check whether the relation is a 'many-to-many' relation or not

public **isReusable** ()

Check if records returned by getting belongs-to/has-many are implicitly cached during the current request

public string | array **getIntermediateFields** ()

Gets the intermediate fields for has-*^{*}-through relations

public **getIntermediateModel** ()

Gets the intermediate model for has-*^{*}-through relations

public string | array **getIntermediateReferencedFields** ()

Gets the intermediate referenced fields for has-*^{*}-through relations

Abstract class **Phalcon\Mvc\Model\Resultset**

implements *Phalcon\Mvc\Model\ResultsetInterface*, *Iterator*, *Traversable*, *SeekableIterator*, *Countable*, *ArrayAccess*, *Serializable*, *JsonSerializable*

This component allows to **Phalcon\Mvc\Model** returns large resultsets with the minimum memory consumption. Resultsets can be traversed using a standard `foreach` or a `while` statement. If a resultset is serialized it will dump all the rows into a big array. Then unserialize will retrieve the rows as they were before serializing.

```
<?php

// Using a standard foreach
$robots = Robots::find(
    [
        "type = 'virtual'",
        "order" => "name",
    ]
);

foreach ($robots as robot) {
    echo robot->name, "\n";
}

// Using a while
$robots = Robots::find(
    [
        "type = 'virtual'",
        "order" => "name",
    ]
);

$robots->rewind();

while ($robots->valid()) {
    $robot = $robots->current();
```

```
echo $robot->name, "\n";

$robots->next ();
}
```

Constants

integer **TYPE_RESULT_FULL**

integer **TYPE_RESULT_PARTIAL**

integer **HYDRATE_RECORDS**

integer **HYDRATE_OBJECTS**

integer **HYDRATE_ARRAYS**

Methods

public **__construct** (*Phalcon\Db\ResultInterface* | *false* \$result, [*Phalcon\Cache\BackendInterface* \$cache])

Phalcon\Mvc\Model\Resultset constructor

public **next** ()

Moves cursor to next row in the resultset

public **valid** ()

Check whether internal resource has rows to fetch

public **key** ()

Gets pointer number of active row in the resultset

final public **rewind** ()

Rewinds resultset to its beginning

final public **seek** (*mixed* \$position)

Changes internal pointer to a specific position in the resultset Set new position if required and set this->_row

final public **count** ()

Counts how many rows are in the resultset

public **offsetExists** (*mixed* \$index)

Checks whether offset exists in the resultset

public **offsetGet** (*mixed* \$index)

Gets row in a specific position of the resultset

public **offsetSet** (*int* \$index, *Phalcon\Mvc\ModelInterface* \$value)

Resultsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public **offsetUnset** (*mixed* \$offset)

Resultsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public **getType** ()

Returns the internal type of data retrieval that the resultset is using

public **getFirst** ()

Get first row in the resultset

public **getLast** ()

Get last row in the resultset

public **setIsFresh** (*mixed* \$isFresh)

Set if the resultset is fresh or an old one cached

public **isFresh** ()

Tell if the resultset if fresh or an old one cached

public **setHydrateMode** (*mixed* \$hydrateMode)

Sets the hydration mode in the resultset

public **getHydrateMode** ()

Returns the current hydration mode

public **getCache** ()

Returns the associated cache for the resultset

public **getMessages** ()

Returns the error messages produced by a batch operation

public *boolean* **update** (*array* \$data, [*Closure* \$conditionCallback])

Updates every record in the resultset

public **delete** ([*Closure* \$conditionCallback])

Deletes every record in the resultset

public *Phalcon\Mvc\Model*[] **filter** (*callback* \$filter)

Filters a resultset returning only those the developer requires

```
<?php

$filtered = $robots->filter(
    function ($robot) {
        if ($robot->id < 3) {
            return $robot;
        }
    }
);
```

public *array* **jsonSerialize** ()

Returns serialised model objects as array for json_encode. Calls jsonSerialize on each object if present

```
<?php

$robots = Robots::find();
echo json_encode($robots);
```

abstract public **toArray** () inherited from *Phalcon\Mvc\Model\ResultsetInterface*

...

abstract public **current** () inherited from *Iterator*

...

abstract public **serialize** () inherited from *Serializable*

...

abstract public **unserialize** (*mixed* \$serialized) inherited from *Serializable*

...

Class *Phalcon\Mvc\Model\Resultset\Complex*

extends abstract class *Phalcon\Mvc\Model\Resultset*

implements *JsonSerializable*, *Serializable*, *ArrayAccess*, *Countable*, *SeekableIterator*, *Traversable*, *Iterator*, *Phalcon\Mvc\Model\ResultsetInterface*

Complex resultsets may include complete objects and scalar values. This class builds every complex row as it is required

Constants

integer **TYPE_RESULT_FULL**

integer **TYPE_RESULT_PARTIAL**

integer **HYDRATE_RECORDS**

integer **HYDRATE_OBJECTS**

integer **HYDRATE_ARRAYS**

Methods

public **__construct** (*array* \$columnTypes, [*Phalcon\Db\ResultInterface* \$result], [*Phalcon\Cache\BackendInterface* \$cache])

Phalcon\Mvc\Model\Resultset\Complex constructor

final public **current** ()

Returns current row in the resultset

public **toArray** ()

Returns a complete resultset as an array, if the resultset has a big number of rows it could consume more memory than currently it does.

public **serialize** ()

Serializing a resultset will dump all related rows into a big array

public **unserialize** (*mixed* \$data)

Unserializing a resultset will allow to only works on the rows present in the saved state

public **next** () inherited from *Phalcon\Mvc\Model\Resultset*

Moves cursor to next row in the resultset

public **valid** () inherited from *Phalcon\Mvc\Model\Resultset*

Check whether internal resource has rows to fetch

public **key** () inherited from *Phalcon\Mvc\Model\Resultset*

Gets pointer number of active row in the resultset

final public **rewind** () inherited from *Phalcon\Mvc\Model\Resultset*

Rewinds resultset to its beginning

final public **seek** (*mixed* \$position) inherited from *Phalcon\Mvc\Model\Resultset*

Changes internal pointer to a specific position in the resultset Set new position if required and set this->_row

final public **count** () inherited from *Phalcon\Mvc\Model\Resultset*

Counts how many rows are in the resultset

public **offsetExists** (*mixed* \$index) inherited from *Phalcon\Mvc\Model\Resultset*

Checks whether offset exists in the resultset

public **offsetGet** (*mixed* \$index) inherited from *Phalcon\Mvc\Model\Resultset*

Gets row in a specific position of the resultset

public **offsetSet** (*int* \$index, *Phalcon\Mvc\ModelInterface* \$value) inherited from *Phalcon\Mvc\Model\Resultset*

Resultsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public **offsetUnset** (*mixed* \$offset) inherited from *Phalcon\Mvc\Model\Resultset*

Resultsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public **getType** () inherited from *Phalcon\Mvc\Model\Resultset*

Returns the internal type of data retrieval that the resultset is using

public **getFirst** () inherited from *Phalcon\Mvc\Model\Resultset*

Get first row in the resultset

public **getLast** () inherited from *Phalcon\Mvc\Model\Resultset*

Get last row in the resultset

public **setIsFresh** (*mixed* \$isFresh) inherited from *Phalcon\Mvc\Model\Resultset*

Set if the resultset is fresh or an old one cached

public **isFresh** () inherited from *Phalcon\Mvc\Model\Resultset*

Tell if the resultset if fresh or an old one cached

public **setHydrateMode** (*mixed* \$hydrateMode) inherited from *Phalcon\Mvc\Model\Resultset*

Sets the hydration mode in the resultset

public **getHydrateMode** () inherited from *Phalcon\Mvc\Model\Resultset*

Returns the current hydration mode

public **getCache** () inherited from *Phalcon\Mvc\Model\Resultset*

Returns the associated cache for the resultset

public **getMessages** () inherited from *Phalcon\Mvc\Model\Resultset*

Returns the error messages produced by a batch operation

public *boolean* **update** (*array* \$data, [*Closure* \$conditionCallback]) inherited from *Phalcon\Mvc\Model\Resultset*

Updates every record in the resultset

public **delete** ([*Closure* \$conditionCallback]) inherited from *Phalcon\Mvc\Model\Resultset*

Deletes every record in the resultset

public *Phalcon\Mvc\Model*[] **filter** (*callback* \$filter) inherited from *Phalcon\Mvc\Model\Resultset*

Filters a resultset returning only those the developer requires

```
<?php

$filtered = $robots->filter(
    function ($robot) {
        if ($robot->id < 3) {
            return $robot;
        }
    }
);
```

public *array* **jsonSerialize** () inherited from *Phalcon\Mvc\Model\Resultset*

Returns serialised model objects as array for json_encode. Calls jsonSerialize on each object if present

```
<?php

$robots = Robots::find();
echo json_encode($robots);
```

Class *Phalcon\Mvc\Model\Resultset\Simple*

extends abstract class *Phalcon\Mvc\Model\Resultset*

implements *JsonSerializable*, *Serializable*, *ArrayAccess*, *Countable*, *SeekableIterator*, *Traversable*, *Iterator*, *Phalcon\Mvc\Model\ResultsetInterface*

Simple resultsets only contains a complete objects This class builds every complete object as it is required

Constants

integer **TYPE_RESULT_FULL**

integer **TYPE_RESULT_PARTIAL**

integer **HYDRATE_RECORDS**

integer **HYDRATE_OBJECTS**

integer **HYDRATE_ARRAYS**

Methods

public **__construct** (*array* \$columnMap, *Phalcon\Mvc\ModelInterface* | *Phalcon\Mvc\Model\Row* \$model, *Phalcon\Db\Result\Pdo* | *null* \$result, [*Phalcon\Cache\BackendInterface* \$cache], [*boolean* \$keepSnapshots])

Phalcon\Mvc\Model\Resultset\Simple constructor

final public **current** ()

Returns current row in the resultset

public **toArray** ([mixed \$renameColumns])

Returns a complete resultset as an array, if the resultset has a big number of rows it could consume more memory than currently it does. Export the resultset to an array couldn't be faster with a large number of records

public **serialize** ()

Serializing a resultset will dump all related rows into a big array

public **unserialize** (mixed \$data)

Unserializing a resultset will allow to only works on the rows present in the saved state

public **next** () inherited from *Phalcon\Mvc\Model\Resultset*

Moves cursor to next row in the resultset

public **valid** () inherited from *Phalcon\Mvc\Model\Resultset*

Check whether internal resource has rows to fetch

public **key** () inherited from *Phalcon\Mvc\Model\Resultset*

Gets pointer number of active row in the resultset

final public **rewind** () inherited from *Phalcon\Mvc\Model\Resultset*

Rewinds resultset to its beginning

final public **seek** (mixed \$position) inherited from *Phalcon\Mvc\Model\Resultset*

Changes internal pointer to a specific position in the resultset Set new position if required and set this->_row

final public **count** () inherited from *Phalcon\Mvc\Model\Resultset*

Counts how many rows are in the resultset

public **offsetExists** (mixed \$index) inherited from *Phalcon\Mvc\Model\Resultset*

Checks whether offset exists in the resultset

public **offsetGet** (mixed \$index) inherited from *Phalcon\Mvc\Model\Resultset*

Gets row in a specific position of the resultset

public **offsetSet** (int \$index, *Phalcon\Mvc\ModelInterface* \$value) inherited from *Phalcon\Mvc\Model\Resultset*

Resultsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public **offsetUnset** (mixed \$offset) inherited from *Phalcon\Mvc\Model\Resultset*

Resultsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public **getType** () inherited from *Phalcon\Mvc\Model\Resultset*

Returns the internal type of data retrieval that the resultset is using

public **getFirst** () inherited from *Phalcon\Mvc\Model\Resultset*

Get first row in the resultset

public **getLast** () inherited from *Phalcon\Mvc\Model\Resultset*

Get last row in the resultset

public **setIsFresh** (*mixed* \$isFresh) inherited from *Phalcon\Mvc\Model\Resultset*

Set if the resultset is fresh or an old one cached

public **isFresh** () inherited from *Phalcon\Mvc\Model\Resultset*

Tell if the resultset if fresh or an old one cached

public **setHydrateMode** (*mixed* \$hydrateMode) inherited from *Phalcon\Mvc\Model\Resultset*

Sets the hydration mode in the resultset

public **getHydrateMode** () inherited from *Phalcon\Mvc\Model\Resultset*

Returns the current hydration mode

public **getCache** () inherited from *Phalcon\Mvc\Model\Resultset*

Returns the associated cache for the resultset

public **getMessages** () inherited from *Phalcon\Mvc\Model\Resultset*

Returns the error messages produced by a batch operation

public *boolean* **update** (array \$data, [*Closure* \$conditionCallback]) inherited from *Phalcon\Mvc\Model\Resultset*

Updates every record in the resultset

public **delete** ([*Closure* \$conditionCallback]) inherited from *Phalcon\Mvc\Model\Resultset*

Deletes every record in the resultset

public *Phalcon\Mvc\Model*[] **filter** (*callback* \$filter) inherited from *Phalcon\Mvc\Model\Resultset*

Filters a resultset returning only those the developer requires

```
<?php
$filterred = $robots->filter(
    function ($robot) {
        if ($robot->id < 3) {
            return $robot;
        }
    }
);
```

public *array* **jsonSerialize** () inherited from *Phalcon\Mvc\Model\Resultset*

Returns serialised model objects as array for json_encode. Calls jsonSerialize on each object if present

```
<?php
$robots = Robots::find();
echo json_encode($robots);
```

Class *Phalcon\Mvc\Model\Row*

implements *Phalcon\Mvc\EntityInterface*, *Phalcon\Mvc\Model\ResultInterface*, *ArrayAccess*, *JsonSerializable*

This component allows *Phalcon\Mvc\Model* to return rows without an associated entity. This objects implements the *ArrayAccess* interface to allow access the object as *object->x* or *array[x]*.

Methods

public **setDirtyState** (*mixed* \$dirtyState)

Set the current object's state

public *boolean* **offsetExists** (*string* | *int* \$index)

Checks whether offset exists in the row

public *string* | *Phalcon\Mvc\ModelInterface* **offsetGet** (*string* | *int* \$index)

Gets a record in a specific position of the row

public **offsetSet** (*string* | *int* \$index, *Phalcon\Mvc\ModelInterface* \$value)

Rows cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public **offsetUnset** (*string* | *int* \$offset)

Rows cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public *mixed* **readAttribute** (*string* \$attribute)

Reads an attribute value by its name

```
<?php
echo $robot->readAttribute("name");
```

public **writeAttribute** (*string* \$attribute, *mixed* \$value)

Writes an attribute value by its name

```
<?php
$robot->writeAttribute("name", "Rosey");
```

public *array* **toArray** ()

Returns the instance as an array representation

public *array* **jsonSerialize** ()

Serializes the object for json_encode

Class *Phalcon\Mvc\Model\Transaction*

implements *Phalcon\Mvc\Model\TransactionInterface*

Transactions are protective blocks where SQL statements are only permanent if they can all succeed as one atomic action. *Phalcon\Transaction* is intended to be used with *Phalcon_Model_Base*. *Phalcon Transactions* should be created using *Phalcon\Transaction\Manager*.

```
<?php
try {
    $manager = new \Phalcon\Mvc\Model\Transaction\Manager();

    $transaction = $manager->get();

    $robot = new Robots();
```

```
$robot->setTransaction($transaction);

$robot->name          = "WALL·E";
$robot->created_at    = date("Y-m-d");

if ($robot->save() === false) {
    $transaction->rollback("Can't save robot");
}

$robotPart = new RobotParts();

$robotPart->setTransaction($transaction);

$robotPart->type = "head";

if ($robotPart->save() === false) {
    $transaction->rollback("Can't save robot part");
}

$transaction->commit();
} catch (Phalcon\Mvc\Model\Transaction\Failed $e) {
    echo "Failed, reason: ", $e->getMessage();
}
```

Methods

public **__construct** (*Phalcon\DiInterface* \$dependencyInjector, [*boolean* \$autoBegin], [*string* \$service])

Phalcon\Mvc\Model\Transaction constructor

public **setTransactionManager** (*Phalcon\Mvc\Model\Transaction\ManagerInterface* \$manager)

Sets transaction manager related to the transaction

public **begin** ()

Starts the transaction

public **commit** ()

Commits the transaction

public *boolean* **rollback** ([*string* \$rollbackMessage], [*Phalcon\Mvc\ModelInterface* \$rollbackRecord])

Rollbacks the transaction

public **getConnection** ()

Returns the connection related to transaction

public **setIsNewTransaction** (*mixed* \$isNew)

Sets if is a reused transaction or new once

public **setRollbackOnAbort** (*mixed* \$rollbackOnAbort)

Sets flag to rollback on abort the HTTP connection

public **isManaged** ()

Checks whether transaction is managed by a transaction manager

public **getMessages** ()

Returns validations messages from last save try

public **isValid** ()

Checks whether internal connection is under an active transaction

public **setRollbackedRecord** (*Phalcon\Mvc\ModelInterface* \$record)

Sets object which generates rollback action

Class **Phalcon\Mvc\Model\Transaction\Exception**

extends class *Phalcon\Mvc\Model\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Class `Phalcon\Mvc\Model\Transaction\Failed`

extends class `Phalcon\Mvc\Model\Transaction\Exception`

implements `Throwable`

This class will be thrown to exit a try/catch block for isolated transactions

Methods

public **__construct** (*mixed* \$message, [`Phalcon\Mvc\ModelInterface` \$record])

`Phalcon\Mvc\Model\Transaction\Failed` constructor

public **getRecordMessages** ()

Returns validation record messages which stop the transaction

public **getRecord** ()

Returns validation record messages which stop the transaction

final private `Exception` **__clone** () inherited from `Exception`

Clone the exception

public **__wakeup** () inherited from `Exception`

...

final public *string* **getMessage** () inherited from `Exception`

Gets the Exception message

final public *int* **getCode** () inherited from `Exception`

Gets the Exception code

final public *string* **getFile** () inherited from `Exception`

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from `Exception`

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from `Exception`

Gets the stack trace

final public `Exception` **getPrevious** () inherited from `Exception`

Returns previous Exception

final public `Exception` **getTraceAsString** () inherited from `Exception`

Gets the stack trace as a string

public *string* **__toString** () inherited from `Exception`

String representation of the exception

Class Phalcon\Mvc\Model\Transaction\Manager

implements *Phalcon\Mvc\Model\Transaction\ManagerInterface*, *Phalcon\Di\InjectionAwareInterface*

A transaction acts on a single database connection. If you have multiple class-specific databases, the transaction will not protect interaction among them.

This class manages the objects that compose a transaction. A transaction produces a unique connection that is passed to every object part of the transaction.

```
<?php

try {
    use Phalcon\Mvc\Model\Transaction\Manager as TransactionManager;

    $transactionManager = new TransactionManager();

    $transaction = $transactionManager->get();

    $robot = new Robots();

    $robot->setTransaction($transaction);

    $robot->name          = "WALL·E";
    $robot->created_at    = date("Y-m-d");

    if ($robot->save() === false) {
        $transaction->rollback("Can't save robot");
    }

    $robotPart = new RobotParts();

    $robotPart->setTransaction($transaction);

    $robotPart->type = "head";

    if ($robotPart->save() === false) {
        $transaction->rollback("Can't save robot part");
    }

    $transaction->commit();
} catch (Phalcon\Mvc\Model\Transaction\Failed $e) {
    echo "Failed, reason: ", $e->getMessage();
}
```

Methods

public **__construct** ([*Phalcon\DiInterface* \$dependencyInjector])

Phalcon\Mvc\Model\Transaction\Manager constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injection container

public **getDI** ()

Returns the dependency injection container

public **setDbService** (*mixed* \$service)

Sets the database service used to run the isolated transactions

public *string* **getDbService** ()

Returns the database service used to isolate the transaction

public **setRollbackPendent** (*mixed* \$rollbackPendent)

Set if the transaction manager must register a shutdown function to clean up pendent transactions

public **getRollbackPendent** ()

Check if the transaction manager is registering a shutdown function to clean up pendent transactions

public **has** ()

Checks whether the manager has an active transaction

public **get** ([*mixed* \$autoBegin])

Returns a new \Phalcon\Mvc\Model\Transaction or an already created once This method registers a shutdown function to rollback active connections

public **getOrCreateTransaction** ([*mixed* \$autoBegin])

Create/Returns a new transaction or an existing one

public **rollbackPendent** ()

Rollbacks active transactions within the manager

public **commit** ()

Commits active transactions within the manager

public **rollback** ([*boolean* \$collect])

Rollbacks active transactions within the manager Collect will remove the transaction from the manager

public **notifyRollback** (*Phalcon\Mvc\Model\TransactionInterface* \$transaction)

Notifies the manager about a rollbacked transaction

public **notifyCommit** (*Phalcon\Mvc\Model\TransactionInterface* \$transaction)

Notifies the manager about a committed transaction

protected **_collectTransaction** (*Phalcon\Mvc\Model\TransactionInterface* \$transaction)

Removes transactions from the TransactionManager

public **collectTransactions** ()

Remove all the transactions from the manager

Class **Phalcon\Mvc\Model\ValidationFailed**

extends class *Phalcon\Mvc\Model\Exception*

implements *Throwable*

This exception is generated when a model fails to save a record Phalcon\Mvc\Model must be set up to have this behavior

Methods

public **__construct** (*Model* \$model, *Message*[] \$validationMessages)

Phalcon\Mvc\Model\ValidationFailed constructor

public **getModel** ()

Returns the model that generated the messages

public **getMessages** ()

Returns the complete group of messages produced in the validation

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Abstract class Phalcon\Mvc\Model\Validator

implements *Phalcon\Mvc\Model\ValidatorInterface*

This is a base class for Phalcon\Mvc\Model validators

This class is only for backward compatibility reasons to use with Phalcon\Mvc\Collection. Otherwise please use the validators provided by Phalcon\Validation.

Methods

public **__construct** (*array* \$options)

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** (*string* \$message, [*string* | *array* \$field], [*string* \$type])

Appends a message to the validator

public **getMessages** ()

Returns messages generated by the validator

public *array* **getOptions** ()

Returns all the options from the validator

public **getOption** (*mixed* \$option, [*mixed* \$defaultValue])

Returns an option

public **isSetOption** (*mixed* \$option)

Check whether an option has been defined in the validator options

abstract public **validate** (*Phalcon\Mvc\EntityInterface* \$record) inherited from *Phalcon\Mvc\Model\ValidatorInterface*

...

Class Phalcon\Mvc\Model\Validator\Email

extends abstract class *Phalcon\Mvc\Model\Validator*

implements *Phalcon\Mvc\Model\ValidatorInterface*

Allows to validate if email fields has correct values

This validator is only for use with Phalcon\Mvc\Collection. If you are using Phalcon\Mvc\Model, please use the validators provided by Phalcon\Validation.

```
<?php

use Phalcon\Mvc\Model\Validator\Email as EmailValidator;

class Subscriptors extends \Phalcon\Mvc\Collection
{
    public function validation()
    {
        $this->validate(
            new EmailValidator(
                [
                    "field" => "electronic_mail",
                ]
            )
        );

        if ($this->validationHasFailed() === true) {
            return false;
        }
    }
}
```


Methods

public **validate** (*Phalcon\Mvc\EntityInterface* \$record)

Executes the validator

public **__construct** (*array* \$options) inherited from *Phalcon\Mvc\Model\Validator*

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** (*string* \$message, [*string* | *array* \$field], [*string* \$type]) inherited from *Phalcon\Mvc\Model\Validator*

Appends a message to the validator

public **getMessages** () inherited from *Phalcon\Mvc\Model\Validator*

Returns messages generated by the validator

public *array* **getOptions** () inherited from *Phalcon\Mvc\Model\Validator*

Returns all the options from the validator

public **getOption** (*mixed* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Mvc\Model\Validator*

Returns an option

public **isSetOption** (*mixed* \$option) inherited from *Phalcon\Mvc\Model\Validator*

Check whether an option has been defined in the validator options

Class *Phalcon\Mvc\Model\Validator\ExclusionIn*

extends abstract class *Phalcon\Mvc\Model\Validator*

implements *Phalcon\Mvc\Model\ValidatorInterface*

Phalcon\Mvc\Model\Validator\ExclusionIn

Check if a value is not included into a list of values

This validator is only for use with *Phalcon\Mvc\Collection*. If you are using *Phalcon\Mvc\Model*, please use the validators provided by *Phalcon\Validation*.

```
<?php
use Phalcon\Mvc\Model\Validator\ExclusionIn as ExclusionInValidator;

class Subscribers extends \Phalcon\Mvc\Collection
{
    public function validation()
    {
        $this->validate(
            new ExclusionInValidator(
                [
                    "field" => "status",
                    "domain" => ["A", "I"],
                ]
            )
        );

        if ($this->validationHasFailed() === true) {
            return false;
        }
    }
}
```

```
}  
}  
}
```

Methods

public **validate** (*Phalcon\Mvc\EntityInterface* \$record)

Executes the validator

public **__construct** (*array* \$options) inherited from *Phalcon\Mvc\Model\Validator*

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** (*string* \$message, [*string* | *array* \$field], [*string* \$type]) inherited from *Phalcon\Mvc\Model\Validator*

Appends a message to the validator

public **getMessages** () inherited from *Phalcon\Mvc\Model\Validator*

Returns messages generated by the validator

public *array* **getOptions** () inherited from *Phalcon\Mvc\Model\Validator*

Returns all the options from the validator

public **getOption** (*mixed* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Mvc\Model\Validator*

Returns an option

public **isSetOption** (*mixed* \$option) inherited from *Phalcon\Mvc\Model\Validator*

Check whether an option has been defined in the validator options

Class Phalcon\Mvc\Model\Validator\InclusionIn

extends abstract class *Phalcon\Mvc\Model\Validator*

implements *Phalcon\Mvc\Model\ValidatorInterface*

Phalcon\Mvc\Model\Validator\InclusionIn

Check if a value is included into a list of values

This validator is only for use with Phalcon\Mvc\Collection. If you are using Phalcon\Mvc\Model, please use the validators provided by Phalcon\Validation.

```
<?php  
  
use Phalcon\Mvc\Model\Validator\InclusionIn as InclusionInValidator;  
  
class Subscribers extends \Phalcon\Mvc\Collection  
{  
    public function validation()  
    {  
        $this->validate(  
            new InclusionInValidator(  
                [  
                    "field" => "status",  
                    "domain" => ["A", "I"],  
                ]  
            )  
        );  
    }  
}
```

```

        ]
    )
);

if ($this->validationHasFailed() === true) {
    return false;
}
}
}

```

Methods

public **validate** (*Phalcon\Mvc\EntityInterface* \$record)

Executes validator

public **__construct** (*array* \$options) inherited from *Phalcon\Mvc\Model\Validator*

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** (*string* \$message, [*string* | *array* \$field], [*string* \$type]) inherited from *Phalcon\Mvc\Model\Validator*

Appends a message to the validator

public **getMessages** () inherited from *Phalcon\Mvc\Model\Validator*

Returns messages generated by the validator

public *array* **getOptions** () inherited from *Phalcon\Mvc\Model\Validator*

Returns all the options from the validator

public **getOption** (*mixed* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Mvc\Model\Validator*

Returns an option

public **isSetOption** (*mixed* \$option) inherited from *Phalcon\Mvc\Model\Validator*

Check whether an option has been defined in the validator options

Class Phalcon\Mvc\Model\Validator\Ip

extends abstract class *Phalcon\Mvc\Model\Validator*

implements *Phalcon\Mvc\Model\ValidatorInterface*

Phalcon\Mvc\Model\Validator\IP

Validates that a value is ipv4 address in valid range

This validator is only for use with Phalcon\Mvc\Collection. If you are using Phalcon\Mvc\Model, please use the validators provided by Phalcon\Validation.

```

<?php
use Phalcon\Mvc\Model\Validator\Ip;

class Data extends \Phalcon\Mvc\Collection
{
    public function validation()

```

```
{
    // Any public IP
    $this->validate(
        new IP (
            [
                "field"      => "server_ip",
                "version"    => IP::VERSION_4 | IP::VERSION_6, // v6 and v4.
↪The same if not specified
                "allowReserved" => false,    // False if not specified. Ignored
↪for v6
                "allowPrivate" => false,    // False if not specified
                "message"    => "IP address has to be correct",
            ]
        )
    );

    // Any public v4 address
    $this->validate(
        new IP (
            [
                "field"      => "ip_4",
                "version"    => IP::VERSION_4,
                "message"    => "IP address has to be correct",
            ]
        )
    );

    // Any v6 address
    $this->validate(
        new IP (
            [
                "field"      => "ip6",
                "version"    => IP::VERSION_6,
                "allowPrivate" => true,
                "message"    => "IP address has to be correct",
            ]
        )
    );

    if ($this->validationHasFailed() === true) {
        return false;
    }
}
}
```

Constants

integer **VERSION_4**

integer **VERSION_6**

Methods

public **validate** (*Phalcon\Mvc\EntityInterface* \$record)

Executes the validator

public **__construct** (*array* \$options) inherited from *Phalcon\Mvc\Model\Validator*

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** (*string* \$message, [*string* | *array* \$field], [*string* \$type]) inherited from *Phalcon\Mvc\Model\Validator*

Appends a message to the validator

public **getMessages** () inherited from *Phalcon\Mvc\Model\Validator*

Returns messages generated by the validator

public *array* **getOptions** () inherited from *Phalcon\Mvc\Model\Validator*

Returns all the options from the validator

public **getOption** (*mixed* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Mvc\Model\Validator*

Returns an option

public **isSetOption** (*mixed* \$option) inherited from *Phalcon\Mvc\Model\Validator*

Check whether an option has been defined in the validator options

Class Phalcon\Mvc\Model\Validator\Numericality

extends abstract class *Phalcon\Mvc\Model\Validator*

implements *Phalcon\Mvc\Model\ValidatorInterface*

Allows to validate if a field has a valid numeric format

This validator is only for use with Phalcon\Mvc\Collection. If you are using Phalcon\Mvc\Model, please use the validators provided by Phalcon\Validation.

```
<?php

use Phalcon\Mvc\Model\Validator\Numericality as NumericalityValidator;

class Products extends \Phalcon\Mvc\Collection
{
    public function validation()
    {
        $this->validate(
            new NumericalityValidator(
                [
                    "field" => "price",
                ]
            )
        );

        if ($this->validationHasFailed() === true) {
            return false;
        }
    }
}
```

Methods

public **validate** (*Phalcon\Mvc\EntityInterface* \$record)

Executes the validator

public **__construct** (*array* \$options) inherited from *Phalcon\Mvc\Model\Validator*

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** (*string* \$message, [*string* | *array* \$field], [*string* \$type]) inherited from *Phalcon\Mvc\Model\Validator*

Appends a message to the validator

public **getMessages** () inherited from *Phalcon\Mvc\Model\Validator*

Returns messages generated by the validator

public **getOptions** () inherited from *Phalcon\Mvc\Model\Validator*

Returns all the options from the validator

public **getOption** (*mixed* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Mvc\Model\Validator*

Returns an option

public **isSetOption** (*mixed* \$option) inherited from *Phalcon\Mvc\Model\Validator*

Check whether an option has been defined in the validator options

Class *Phalcon\Mvc\Model\Validator\PresenceOf*

extends abstract class *Phalcon\Mvc\Model\Validator*

implements *Phalcon\Mvc\Model\ValidatorInterface*

Allows to validate if a field have a value different of null and empty string ("")

This validator is only for use with *Phalcon\Mvc\Collection*. If you are using *Phalcon\Mvc\Model*, please use the validators provided by *Phalcon\Validation*.

```
<?php

use Phalcon\Mvc\Model\Validator\PresenceOf;

class Subscriptors extends \Phalcon\Mvc\Collection
{
    public function validation()
    {
        $this->validate(
            new PresenceOf(
                [
                    "field" => "name",
                    "message" => "The name is required",
                ]
            )
        );

        if ($this->validationHasFailed() === true) {
            return false;
        }
    }
}
```

Methods

public **validate** (*Phalcon\Mvc\EntityInterface* \$record)

Executes the validator

public **__construct** (*array* \$options) inherited from *Phalcon\Mvc\Model\Validator*

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** (*string* \$message, [*string* | *array* \$field], [*string* \$type]) inherited from *Phalcon\Mvc\Model\Validator*

Appends a message to the validator

public **getMessages** () inherited from *Phalcon\Mvc\Model\Validator*

Returns messages generated by the validator

public *array* **getOptions** () inherited from *Phalcon\Mvc\Model\Validator*

Returns all the options from the validator

public **getOption** (*mixed* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Mvc\Model\Validator*

Returns an option

public **isSetOption** (*mixed* \$option) inherited from *Phalcon\Mvc\Model\Validator*

Check whether an option has been defined in the validator options

Class *Phalcon\Mvc\Model\Validator\Regex*

extends abstract class *Phalcon\Mvc\Model\Validator*

implements *Phalcon\Mvc\Model\ValidatorInterface*

Allows validate if the value of a field matches a regular expression

This validator is only for use with *Phalcon\Mvc\Collection*. If you are using *Phalcon\Mvc\Model*, please use the validators provided by *Phalcon\Validation*.

```
<?php

use Phalcon\Mvc\Model\Validator\Regex as RegexValidator;

class Subscriptors extends \Phalcon\Mvc\Collection
{
    public function validation()
    {
        $this->validate(
            new RegexValidator(
                [
                    "field" => "created_at",
                    "pattern" => "/^[0-9]{4}[-\/](0[1-9]|1[12])[-\/](0[1-9]|1[12])[0-9]|3[01])"/,
                ]
            )
        );

        if ($this->validationHasFailed() == true) {
            return false;
        }
    }
}
```

```
}  
}
```

Methods

public **validate** (*Phalcon\Mvc\EntityInterface* \$record)

Executes the validator

public **__construct** (*array* \$options) inherited from *Phalcon\Mvc\Model\Validator*

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** (*string* \$message, [*string* | *array* \$field], [*string* \$type]) inherited from *Phalcon\Mvc\Model\Validator*

Appends a message to the validator

public **getMessages** () inherited from *Phalcon\Mvc\Model\Validator*

Returns messages generated by the validator

public *array* **getOptions** () inherited from *Phalcon\Mvc\Model\Validator*

Returns all the options from the validator

public **getOption** (*mixed* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Mvc\Model\Validator*

Returns an option

public **isSetOption** (*mixed* \$option) inherited from *Phalcon\Mvc\Model\Validator*

Check whether an option has been defined in the validator options

Class Phalcon\Mvc\Model\Validator\StringLength

extends abstract class *Phalcon\Mvc\Model\Validator*

implements *Phalcon\Mvc\Model\ValidatorInterface*

Simply validates specified string length constraints

This validator is only for use with Phalcon\Mvc\Collection. If you are using Phalcon\Mvc\Model, please use the validators provided by Phalcon\Validation.

```
<?php  
  
use Phalcon\Mvc\Model\Validator\StringLength as StringLengthValidator;  
  
class Subscribers extends \Phalcon\Mvc\Collection  
{  
    public function validation()  
    {  
        $this->validate(  
            new StringLengthValidator(  
                [  
                    "field"           => "name_last",  
                    "max"             => 50,  
                    "min"             => 2,  
                    "messageMaximum" => "We don't like really long names",  
                    "messageMinimum" => "We want more than just their initials",  
                ]  
            )  
        );  
    }  
}
```



```

        ]
    )
);

if ($this->validationHasFailed() === true) {
    return false;
}
}
}

```

Methods

public **validate** (*Phalcon\Mvc\EntityInterface* \$record)

Executes the validator

public **__construct** (*array* \$options) inherited from *Phalcon\Mvc\Model\Validator*

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** (*string* \$message, [*string* | *array* \$field], [*string* \$type]) inherited from *Phalcon\Mvc\Model\Validator*

Appends a message to the validator

public **getMessages** () inherited from *Phalcon\Mvc\Model\Validator*

Returns messages generated by the validator

public *array* **getOptions** () inherited from *Phalcon\Mvc\Model\Validator*

Returns all the options from the validator

public **getOption** (*mixed* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Mvc\Model\Validator*

Returns an option

public **isSetOption** (*mixed* \$option) inherited from *Phalcon\Mvc\Model\Validator*

Check whether an option has been defined in the validator options

Class Phalcon\Mvc\Model\Validator\Uniqueness

extends abstract class *Phalcon\Mvc\Model\Validator*

implements *Phalcon\Mvc\Model\ValidatorInterface*

Validates that a field or a combination of a set of fields are not present more than once in the existing records of the related table

This validator is only for use with Phalcon\Mvc\Collection. If you are using Phalcon\Mvc\Model, please use the validators provided by Phalcon\Validation.

```

<?php

use Phalcon\Mvc\Collection;
use Phalcon\Mvc\Model\Validator\Uniqueness;

class Subscribers extends Collection
{
    public function validation()

```

```
{
    $this->validate(
        new Uniqueness(
            [
                "field" => "email",
                "message" => "Value of field 'email' is already present in_
↪another record",
            ]
        )
    );

    if ($this->validationHasFailed() === true) {
        return false;
    }
}
```

Methods

public **validate** (*Phalcon\Mvc\EntityInterface* \$record)

Executes the validator

public **__construct** (array \$options) inherited from *Phalcon\Mvc\Model\Validator*

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** (*string* \$message, [*string* | *array* \$field], [*string* \$type]) inherited from *Phalcon\Mvc\Model\Validator*

Appends a message to the validator

public **getMessages** () inherited from *Phalcon\Mvc\Model\Validator*

Returns messages generated by the validator

public array **getOptions** () inherited from *Phalcon\Mvc\Model\Validator*

Returns all the options from the validator

public **getOption** (*mixed* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Mvc\Model\Validator*

Returns an option

public **isSetOption** (*mixed* \$option) inherited from *Phalcon\Mvc\Model\Validator*

Check whether an option has been defined in the validator options

Class *Phalcon\Mvc\Model\Validator\Url*

extends abstract class *Phalcon\Mvc\Model\Validator*

implements *Phalcon\Mvc\Model\ValidatorInterface*

Allows to validate if a field has a url format

This validator is only for use with *Phalcon\Mvc\Collection*. If you are using *Phalcon\Mvc\Model*, please use the validators provided by *Phalcon\Validation*.

```

<?php

use Phalcon\Mvc\Model\Validator\Url as UrlValidator;

class Posts extends \Phalcon\Mvc\Collection
{
    public function validation()
    {
        $this->validate(
            new UrlValidator(
                [
                    "field" => "source_url",
                ]
            )
        );

        if ($this->validationHasFailed() === true) {
            return false;
        }
    }
}

```

Methods

public **validate** (*Phalcon\Mvc\EntityInterface* \$record)

Executes the validator

public **__construct** (*array* \$options) inherited from *Phalcon\Mvc\Model\Validator*

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** (*string* \$message, [*string* | *array* \$field], [*string* \$type]) inherited from *Phalcon\Mvc\Model\Validator*

Appends a message to the validator

public **getMessages** () inherited from *Phalcon\Mvc\Model\Validator*

Returns messages generated by the validator

public *array* **getOptions** () inherited from *Phalcon\Mvc\Model\Validator*

Returns all the options from the validator

public **getOption** (*mixed* \$option, [*mixed* \$defaultValue]) inherited from *Phalcon\Mvc\Model\Validator*

Returns an option

public **isSetOption** (*mixed* \$option) inherited from *Phalcon\Mvc\Model\Validator*

Check whether an option has been defined in the validator options

Class Phalcon\Mvc\Router

implements *Phalcon\Di\InjectionAwareInterface*, *Phalcon\Mvc\RouterInterface*, *Phalcon\Events\EventsAwareInterface*

Phalcon\MvcRouter is the standard framework router. Routing is the process of taking a URI endpoint (that part of the URI which comes after the base URL) and decomposing it into parameters to determine which module, controller, and action of that controller should receive the request

```
<?php

use Phalcon\MvcRouter;

$routeur = new Router();

$routeur->add(
    "/documentation/{chapter}/{name}\.{type:[a-z]+}",
    [
        "controller" => "documentation",
        "action"      => "show",
    ]
);

$routeur->handle();

echo $routeur->getControllerName();
```

Constants

integer **URI_SOURCE_GET_URL**

integer **URI_SOURCE_SERVER_REQUEST_URI**

integer **POSITION_FIRST**

integer **POSITION_LAST**

Methods

public **__construct** ([*mixed* \$defaultRoutes])

Phalcon\MvcRouter constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injector

public **getDI** ()

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* \$eventsManager)

Sets the events manager

public **getEventsManager** ()

Returns the internal event manager

public **getRewriteUri** ()

Get rewrite info. This info is read from \$_GET['_url']. This returns '/' if the rewrite information cannot be read

public **setUriSource** (*mixed* \$uriSource)

Sets the URI source. One of the URI_SOURCE_* constants

```
<?php

$router->setUriSource(
    Router::URI_SOURCE_SERVER_REQUEST_URI
);
```

public **removeExtraSlashes** (*mixed* \$remove)

Set whether router must remove the extra slashes in the handled routes

public **setDefaultNamespace** (*mixed* \$namespaceName)

Sets the name of the default namespace

public **setDefaultModule** (*mixed* \$moduleName)

Sets the name of the default module

public **setDefaultController** (*mixed* \$controllerName)

Sets the default controller name

public **setDefaultAction** (*mixed* \$actionName)

Sets the default action name

public **setDefaults** (*array* \$defaults)

Sets an array of default paths. If a route is missing a path the router will use the defined here This method must not be used to set a 404 route

```
<?php

$router->setDefaults(
    [
        "module" => "common",
        "action" => "index",
    ]
);
```

public **getDefaults** ()

Returns an array of default parameters

public **handle** ([*mixed* \$uri])

Handles routing information received from the rewrite engine

```
<?php

// Read the info from the rewrite engine
$router->handle();

// Manually passing an URL
$router->handle("/posts/edit/1");
```

public **add** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$httpMethods], [*mixed* \$position])

Adds a route to the router without any HTTP constraint

```
<?php

use Phalcon\Mvc\Router;
```

```
$router->add("/about", "About::index");  
$router->add("/about", "About::index", ["GET", "POST"]);  
$router->add("/about", "About::index", ["GET", "POST"], Router::POSITION_FIRST);
```

public **addGet** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position])

Adds a route to the router that only match if the HTTP method is GET

public **addPost** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position])

Adds a route to the router that only match if the HTTP method is POST

public **addPut** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position])

Adds a route to the router that only match if the HTTP method is PUT

public **addPatch** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position])

Adds a route to the router that only match if the HTTP method is PATCH

public **addDelete** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position])

Adds a route to the router that only match if the HTTP method is DELETE

public **addOptions** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position])

Add a route to the router that only match if the HTTP method is OPTIONS

public **addHead** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position])

Adds a route to the router that only match if the HTTP method is HEAD

public **addPurge** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position])

Adds a route to the router that only match if the HTTP method is PURGE (Squid and Varnish support)

public **addTrace** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position])

Adds a route to the router that only match if the HTTP method is TRACE

public **addConnect** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position])

Adds a route to the router that only match if the HTTP method is CONNECT

public **mount** (*Phalcon\Mvc\Router\GroupInterface* \$group)

Mounts a group of routes in the router

public **notFound** (*mixed* \$paths)

Set a group of paths to be returned when none of the defined routes are matched

public **clear** ()

Removes all the pre-defined routes

public **getNamespaceName** ()

Returns the processed namespace name

public **getModuleName** ()

Returns the processed module name

public **getControllerName** ()

Returns the processed controller name

public **getActionName** ()

Returns the processed action name

public **getParams** ()

Returns the processed parameters

public **getMatchedRoute** ()

Returns the route that matches the handled URI

public **getMatches** ()

Returns the sub expressions in the regular expression matched

public **wasMatched** ()

Checks if the router matches any of the defined routes

public **getRoutes** ()

Returns all the routes defined in the router

public **getRouteById** (*mixed* \$id)

Returns a route object by its id

public **getRouteByName** (*mixed* \$name)

Returns a route object by its name

public **isExactControllerName** ()

Returns whether controller name should not be mangled

Class **Phalcon\Mvc\Router\Annotations**

extends class *Phalcon\Mvc\Router*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\Mvc\RouterInterface*, *Phalcon\DI\InjectionAwareInterface*

A router that reads routes annotations from classes/resources

```
<?php

use Phalcon\Mvc\Router\Annotations;

$di->setShared(
    "router",
    function() {
        // Use the annotations router
        $router = new Annotations(false);

        // This will do the same as above but only if the handled uri starts with /
        ↪ robots
        $router->addResource("Robots", "/robots");

        return $router;
    }
);
```

Constants

integer **URI_SOURCE_GET_URL**

integer **URI_SOURCE_SERVER_REQUEST_URI**

integer **POSITION_FIRST**

integer **POSITION_LAST**

Methods

public **addResource** (*mixed* \$handler, [*mixed* \$prefix])

Adds a resource to the annotations handler A resource is a class that contains routing annotations

public **addModuleResource** (*mixed* \$module, *mixed* \$handler, [*mixed* \$prefix])

Adds a resource to the annotations handler A resource is a class that contains routing annotations The class is located in a module

public **handle** ([*mixed* \$uri])

Produce the routing parameters from the rewrite information

public **processControllerAnnotation** (*mixed* \$handler, *Phalcon\Annotations\Annotation* \$annotation)

Checks for annotations in the controller docblock

public **processActionAnnotation** (*mixed* \$module, *mixed* \$namespaceName, *mixed* \$controller, *mixed* \$action, *Phalcon\Annotations\Annotation* \$annotation)

Checks for annotations in the public methods of the controller

public **setControllerSuffix** (*mixed* \$controllerSuffix)

Changes the controller class suffix

public **setActionSuffix** (*mixed* \$actionSuffix)

Changes the action method suffix

public **getResources** ()

Return the registered resources

public **__construct** ([*mixed* \$defaultRoutes]) inherited from *Phalcon\Mvc\Router*

Phalcon\Mvc\Router constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Mvc\Router*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Mvc\Router*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Mvc\Router*

Sets the events manager

public **getEventManager** () inherited from *Phalcon\Mvc\Router*

Returns the internal event manager

public **getRewriteUri** () inherited from *Phalcon\Mvc\Router*

Get rewrite info. This info is read from `$_GET['_url']`. This returns `'/'` if the rewrite information cannot be read

public **setUriSource** (*mixed* \$uriSource) inherited from *Phalcon\Mvc\Router*

Sets the URI source. One of the `URI_SOURCE_*` constants

```
<?php

$router->setUriSource(
    Router::URI_SOURCE_SERVER_REQUEST_URI
);
```

public **removeExtraSlashes** (*mixed* \$remove) inherited from *Phalcon\Mvc\Router*

Set whether router must remove the extra slashes in the handled routes

public **setDefaultNamespace** (*mixed* \$namespaceName) inherited from *Phalcon\Mvc\Router*

Sets the name of the default namespace

public **setDefaultModule** (*mixed* \$moduleName) inherited from *Phalcon\Mvc\Router*

Sets the name of the default module

public **setDefaultController** (*mixed* \$controllerName) inherited from *Phalcon\Mvc\Router*

Sets the default controller name

public **setDefaultAction** (*mixed* \$actionName) inherited from *Phalcon\Mvc\Router*

Sets the default action name

public **setDefaults** (*array* \$defaults) inherited from *Phalcon\Mvc\Router*

Sets an array of default paths. If a route is missing a path the router will use the defined here This method must not be used to set a 404 route

```
<?php

$router->setDefaults(
    [
        "module" => "common",
        "action" => "index",
    ]
);
```

public **getDefaults** () inherited from *Phalcon\Mvc\Router*

Returns an array of default parameters

public **add** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$httpMethods], [*mixed* \$position]) inherited from *Phalcon\Mvc\Router*

Adds a route to the router without any HTTP constraint

```
<?php

use Phalcon\Mvc\Router;

$router->add("/about", "About::index");
$router->add("/about", "About::index", ["GET", "POST"]);
$router->add("/about", "About::index", ["GET", "POST"], Router::POSITION_FIRST);
```

public **addGet** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position]) inherited from *Phalcon\Mvc\Router*
Adds a route to the router that only match if the HTTP method is GET

public **addPost** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position]) inherited from *Phalcon\Mvc\Router*
Adds a route to the router that only match if the HTTP method is POST

public **addPut** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position]) inherited from *Phalcon\Mvc\Router*
Adds a route to the router that only match if the HTTP method is PUT

public **addPatch** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position]) inherited from *Phalcon\Mvc\Router*
Adds a route to the router that only match if the HTTP method is PATCH

public **addDelete** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position]) inherited from *Phalcon\Mvc\Router*
Adds a route to the router that only match if the HTTP method is DELETE

public **addOptions** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position]) inherited from *Phalcon\Mvc\Router*
Add a route to the router that only match if the HTTP method is OPTIONS

public **addHead** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position]) inherited from *Phalcon\Mvc\Router*
Adds a route to the router that only match if the HTTP method is HEAD

public **addPurge** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position]) inherited from *Phalcon\Mvc\Router*
Adds a route to the router that only match if the HTTP method is PURGE (Squid and Varnish support)

public **addTrace** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position]) inherited from *Phalcon\Mvc\Router*
Adds a route to the router that only match if the HTTP method is TRACE

public **addConnect** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$position]) inherited from *Phalcon\Mvc\Router*
Adds a route to the router that only match if the HTTP method is CONNECT

public **mount** (*Phalcon\Mvc\Router\GroupInterface* \$group) inherited from *Phalcon\Mvc\Router*
Mounts a group of routes in the router

public **notFound** (*mixed* \$paths) inherited from *Phalcon\Mvc\Router*
Set a group of paths to be returned when none of the defined routes are matched

public **clear** () inherited from *Phalcon\Mvc\Router*
Removes all the pre-defined routes

public **getNamespaceName** () inherited from *Phalcon\Mvc\Router*
Returns the processed namespace name

public **getModuleName** () inherited from *Phalcon\Mvc\Router*
Returns the processed module name

public **getControllerName** () inherited from *Phalcon\Mvc\Router*
Returns the processed controller name

public **getActionName** () inherited from *Phalcon\Mvc\Router*
Returns the processed action name

public **getParams** () inherited from *Phalcon\Mvc\Router*
Returns the processed parameters

public **getMatchedRoute** () inherited from *Phalcon\Mvc\Router*

Returns the route that matches the handled URI

public **getMatches** () inherited from *Phalcon\Mvc\Router*

Returns the sub expressions in the regular expression matched

public **wasMatched** () inherited from *Phalcon\Mvc\Router*

Checks if the router matches any of the defined routes

public **getRoutes** () inherited from *Phalcon\Mvc\Router*

Returns all the routes defined in the router

public **getRouteById** (*mixed* \$id) inherited from *Phalcon\Mvc\Router*

Returns a route object by its id

public **getRouteByName** (*mixed* \$name) inherited from *Phalcon\Mvc\Router*

Returns a route object by its name

public **isExactControllerName** () inherited from *Phalcon\Mvc\Router*

Returns whether controller name should not be mangled

Class *Phalcon\Mvc\Router\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Mvc\Router\Group](#)

implements [Phalcon\Mvc\Router\GroupInterface](#)

Helper class to create a group of routes with common attributes

```
<?php

$router = new \Phalcon\Mvc\Router();

//Create a group with a common module and controller
$blog = new Group(
    [
        "module"      => "blog",
        "controller" => "index",
    ]
);

//All the routes start with /blog
$blog->setPrefix("/blog");

//Add a route to the group
$blog->add(
    "/save",
    [
        "action" => "save",
    ]
);

//Add another route to the group
$blog->add(
    "/edit/{id}",
    [
        "action" => "edit",
    ]
);

//This route maps to a controller different than the default
$blog->add(
    "/blog",
    [
        "controller" => "about",
        "action"      => "index",
    ]
);

//Add the group to the router
$router->mount($blog);
```

Methods

public **__construct** (*[mixed \$paths]*)

Phalcon\Mvc\Router\Group constructor

public **setHostname** (*mixed \$hostname*)

Set a hostname restriction for all the routes in the group

public **getHostname** ()

Returns the hostname restriction

public **setPrefix** (*mixed \$prefix*)

Set a common uri prefix for all the routes in this group

public **getPrefix** ()

Returns the common prefix for all the routes

public **beforeMatch** (*mixed \$beforeMatch*)

Sets a callback that is called if the route is matched. The developer can implement any arbitrary conditions here If the callback returns false the route is treated as not matched

public **getBeforeMatch** ()

Returns the 'before match' callback if any

public **setPaths** (*mixed \$paths*)

Set common paths for all the routes in the group

public **getPaths** ()

Returns the common paths defined for this group

public **getRoutes** ()

Returns the routes added to the group

public **add** (*mixed \$pattern*, [*mixed \$paths*], [*mixed \$httpMethods*])

Adds a route to the router on any HTTP method

```
<?php
$router->add("/about", "About::index");
```

public *Phalcon\Mvc\Router\Route* **addGet** (*string \$pattern*, [*string/array \$paths*])

Adds a route to the router that only match if the HTTP method is GET

public *Phalcon\Mvc\Router\Route* **addPost** (*string \$pattern*, [*string/array \$paths*])

Adds a route to the router that only match if the HTTP method is POST

public *Phalcon\Mvc\Router\Route* **addPut** (*string \$pattern*, [*string/array \$paths*])

Adds a route to the router that only match if the HTTP method is PUT

public *Phalcon\Mvc\Router\Route* **addPatch** (*string \$pattern*, [*string/array \$paths*])

Adds a route to the router that only match if the HTTP method is PATCH

public *Phalcon\Mvc\Router\Route* **addDelete** (*string* \$pattern, [*string/array* \$paths])

Adds a route to the router that only match if the HTTP method is DELETE

public *Phalcon\Mvc\Router\Route* **addOptions** (*string* \$pattern, [*string/array* \$paths])

Add a route to the router that only match if the HTTP method is OPTIONS

public *Phalcon\Mvc\Router\Route* **addHead** (*string* \$pattern, [*string/array* \$paths])

Adds a route to the router that only match if the HTTP method is HEAD

public **clear** ()

Removes all the pre-defined routes

protected **_addRoute** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$httpMethods])

Adds a route applying the common attributes

Class *Phalcon\Mvc\Router\Route*

implements *Phalcon\Mvc\Router\RouteInterface*

This class represents every route added to the router

Methods

public **__construct** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$httpMethods])

Phalcon\Mvc\Router\Route constructor

public **compilePattern** (*mixed* \$pattern)

Replaces placeholders from pattern returning a valid PCRE regular expression

public **via** (*mixed* \$httpMethods)

Set one or more HTTP methods that constraint the matching of the route

```
<?php
$route->via ("GET");

$route->via (
    [
        "GET",
        "POST",
    ]
);
```

public **extractNamedParams** (*mixed* \$pattern)

Extracts parameters from a string

public **reConfigure** (*mixed* \$pattern, [*mixed* \$paths])

Reconfigure the route adding a new pattern and a set of paths

public static **getRoutePaths** ([*mixed* \$paths])

Returns routePaths

public **getName** ()

Returns the route's name

public **setName** (*mixed* \$name)

Sets the route's name

```
<?php

$router->add(
    "/about",
    [
        "controller" => "about",
    ]
)->setName("about");
```

public **beforeMatch** (*mixed* \$callback)

Sets a callback that is called if the route is matched. The developer can implement any arbitrary conditions here. If the callback returns false the route is treated as not matched.

```
<?php

$router->add(
    "/login",
    [
        "module"      => "admin",
        "controller" => "session",
    ]
)->beforeMatch(
    function ($uri, $route) {
        // Check if the request was made with Ajax
        if ($_SERVER["HTTP_X_REQUESTED_WITH"] === "xmlhttprequest") {
            return false;
        }

        return true;
    }
);
```

public **getBeforeMatch** ()

Returns the 'before match' callback if any

public **match** (*mixed* \$callback)

Allows to set a callback to handle the request directly in the route

```
<?php

$router->add(
    "/help",
    []
)->match(
    function () {
        return $this->getResponse()->redirect("https://support.google.com/", true);
    }
);
```

public **getMatch** ()

Returns the 'match' callback if any

public **getRouteId** ()

Returns the route's id

public **getPattern** ()

Returns the route's pattern

public **getCompiledPattern** ()

Returns the route's compiled pattern

public **getPaths** ()

Returns the paths

public **getReversedPaths** ()

Returns the paths using positions as keys and names as values

public **setHttpMethods** (*mixed* \$httpMethods)

Sets a set of HTTP methods that constraint the matching of the route (alias of via)

```
<?php
$route->setHttpMethods ("GET");
$route->setHttpMethods (["GET", "POST"]);
```

public **getHttpMethods** ()

Returns the HTTP methods that constraint matching the route

public **setHostname** (*mixed* \$hostname)

Sets a hostname restriction to the route

```
<?php
$route->setHostname ("localhost");
```

public **getHostname** ()

Returns the hostname restriction if any

public **setGroup** (*Phalcon\Mvc\Router\GroupInterface* \$group)

Sets the group associated with the route

public **getGroup** ()

Returns the group associated with the route

public **convert** (*mixed* \$name, *mixed* \$converter)

Adds a converter to perform an additional transformation for certain parameter

public **getConverters** ()

Returns the router converter

public static **reset** ()

Resets the internal route id generator

Class Phalcon\Mvc\Url

implements *Phalcon\Mvc\UrlInterface*, *Phalcon\Di\InjectionAwareInterface*

This components helps in the generation of: URIs, URLs and Paths

```

<?php

// Generate a URL appending the URI to the base URI
echo $url->get("products/edit/1");

// Generate a URL for a predefined route
echo $url->get(
    [
        "for"    => "blog-post",
        "title"  => "some-cool-stuff",
        "year"   => "2012",
    ]
);

```

Methods

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the DependencyInjector container

public **getDI** ()

Returns the DependencyInjector container

public **setBaseUri** (*mixed* \$baseUri)

Sets a prefix for all the URIs to be generated

```

<?php

$url->setBaseUri("/invo/");

$url->setBaseUri("/invo/index.php/");

```

public **setStaticBaseUri** (*mixed* \$staticBaseUri)

Sets a prefix for all static URLs generated

```

<?php

$url->setStaticBaseUri("/invo/");

```

public **getBaseUri** ()

Returns the prefix for all the generated urls. By default /

public **getStaticBaseUri** ()

Returns the prefix for all the generated static urls. By default /

public **setBasePath** (*mixed* \$basePath)

Sets a base path for all the generated paths

```
<?php

$url->setBasePath("/var/www/htdocs/");
```

public **getBasePath** ()

Returns the base path

public **get** ([*mixed* \$uri], [*mixed* \$args], [*mixed* \$local], [*mixed* \$baseUri])

Generates a URL

```
<?php

// Generate a URL appending the URI to the base URI
echo $url->get("products/edit/1");

// Generate a URL for a predefined route
echo $url->get(
    [
        "for"    => "blog-post",
        "title"  => "some-cool-stuff",
        "year"   => "2015",
    ]
);

// Generate a URL with GET arguments (/show/products?id=1&name=Carrots)
echo $url->get(
    "show/products",
    [
        "id"    => 1,
        "name"  => "Carrots",
    ]
);

// Generate an absolute URL by setting the third parameter as false.
echo $url->get(
    "https://phalconphp.com/",
    null,
    false
);
```

public **getStatic** ([*mixed* \$uri])

Generates a URL for a static resource

```
<?php

// Generate a URL for a static resource
echo $url->getStatic("img/logo.png");

// Generate a URL for a static predefined route
echo $url->getStatic(
    [
        "for" => "logo-cdn",
    ]
);
```

public **path** ([*mixed* \$path])

Generates a local path

Class `Phalcon\Mvc\Url\Exception`

extends class `Phalcon\Exception`

implements `Throwable`

Methods

final private `Exception` **__clone** () inherited from `Exception`

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [`Exception` \$previous]) inherited from `Exception`

Exception constructor

public **__wakeup** () inherited from `Exception`

...

final public *string* **getMessage** () inherited from `Exception`

Gets the Exception message

final public *int* **getCode** () inherited from `Exception`

Gets the Exception code

final public *string* **getFile** () inherited from `Exception`

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from `Exception`

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from `Exception`

Gets the stack trace

final public `Exception` **getPrevious** () inherited from `Exception`

Returns previous Exception

final public `Exception` **getTraceAsString** () inherited from `Exception`

Gets the stack trace as a string

public *string* **__toString** () inherited from `Exception`

String representation of the exception

Class `Phalcon\Mvc\User\Component`

extends abstract class `Phalcon\Di\Injectable`

implements `Phalcon\Events\EventsAwareInterface`, `Phalcon\Di\InjectionAwareInterface`

Methods

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di\Injectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Di\Injectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Di\Injectable*

Returns the internal event manager

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method **__get**

Class *Phalcon\Mvc\User\Module*

extends abstract class *Phalcon\Di\Injectable*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\Di\InjectionAwareInterface*

Methods

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di\Injectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Di\Injectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Di\Injectable*

Returns the internal event manager

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method **__get**

Class *Phalcon\Mvc\User\Plugin*

extends abstract class *Phalcon\Di\Injectable*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\Di\InjectionAwareInterface*

Methods

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di\Injectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Di\Injectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Di\Injectable*

Returns the internal event manager

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method **__get**

Class Phalcon\Mvc\View

extends abstract class *Phalcon\Di\Injectable*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\Di\InjectionAwareInterface*, *Phalcon\Mvc\ViewInterface*, *Phalcon\Mvc\ViewBaseInterface*

Phalcon\Mvc\View is a class for working with the “view” portion of the model-view-controller pattern. That is, it exists to help keep the view script separate from the model and controller scripts. It provides a system of helpers, output filters, and variable escaping.

```
<?php
use Phalcon\Mvc\View;

$view = new View();

// Setting views directory
$view->setViewsDir("app/views/");

$view->start();

// Shows recent posts view (app/views/posts/recent.phtml)
$view->render("posts", "recent");
$view->finish();

// Printing views output
echo $view->getContent();
```

Constants

integer **LEVEL_MAIN_LAYOUT**

integer **LEVEL_AFTER_TEMPLATE**

integer **LEVEL_LAYOUT**

integer **LEVEL_BEFORE_TEMPLATE**

integer **LEVEL_ACTION_VIEW**

integer **LEVEL_NO_RENDER**

integer **CACHE_MODE_NONE**

integer **CACHE_MODE_INVERSE**

Methods

public **getRenderLevel** ()

...

public **getCurrentRenderLevel** ()

...

public **getRegisteredEngines** ()

public **__construct** ([array \$options])

Phalcon\Mvc\View constructor

final protected **_isAbsolutePath** (*mixed* \$path)

Checks if a path is absolute or not

public **setViewsDir** (*mixed* \$viewsDir)

Sets the views directory. Depending of your platform, always add a trailing slash or backslash

public **getViewsDir** ()

Gets views directory

public **setLayoutsDir** (*mixed* \$layoutsDir)

Sets the layouts sub-directory. Must be a directory under the views directory. Depending of your platform, always add a trailing slash or backslash

```
<?php
$view->setLayoutsDir("../common/layouts/");
```

public **getLayoutsDir** ()

Gets the current layouts sub-directory

public **setPartialsDir** (*mixed* \$partialsDir)

Sets a partials sub-directory. Must be a directory under the views directory. Depending of your platform, always add a trailing slash or backslash

```
<?php
$view->setPartialsDir("../common/partials/");
```

public **getPartialsDir** ()

Gets the current partials sub-directory

public **setBasePath** (*mixed* \$basePath)

Sets base path. Depending of your platform, always add a trailing slash or backslash

```
<?php

$view->setBasePath(__DIR__ . "/");
```

public **getBasePath** ()

Gets base path

public **setRenderLevel** (*mixed* \$level)

Sets the render level for the view

```
<?php

// Render the view related to the controller only
$this->view->setRenderLevel(
    View::LEVEL_LAYOUT
);
```

public **disableLevel** (*mixed* \$level)

Disables a specific level of rendering

```
<?php

// Render all levels except ACTION level
$this->view->disableLevel(
    View::LEVEL_ACTION_VIEW
);
```

public **setMainView** (*mixed* \$viewPath)

Sets default view name. Must be a file without extension in the views directory

```
<?php

// Renders as main view views-dir/base.phtml
$this->view->setMainView("base");
```

public **getMainView** ()

Returns the name of the main view

public **setLayout** (*mixed* \$layout)

Change the layout to be used instead of using the name of the latest controller name

```
<?php

$this->view->setLayout("main");
```

public **getLayout** ()

Returns the name of the main view

public **setTemplateBefore** (*mixed* \$templateBefore)

Sets a template before the controller layout

public **cleanTemplateBefore** ()

Resets any “template before” layouts

public **setTemplateAfter** (*mixed* \$templateAfter)

Sets a “template after” controller layout

public **cleanTemplateAfter** ()

Resets any template before layouts

public **setParamToView** (*mixed* \$key, *mixed* \$value)

Adds parameters to views (alias of setVar)

```
<?php
$this->view->setParamToView("products", $products);
```

public **setVars** (*array* \$params, [*mixed* \$merge])

Set all the render params

```
<?php
$this->view->setVars (
    [
        "products" => $products,
    ]
);
```

public **setVar** (*mixed* \$key, *mixed* \$value)

Set a single view parameter

```
<?php
$this->view->setVar("products", $products);
```

public **getVar** (*mixed* \$key)

Returns a parameter previously set in the view

public **getParamsToView** ()

Returns parameters to views

public **getControllerName** ()

Gets the name of the controller rendered

public **getActionName** ()

Gets the name of the action rendered

public **getParams** ()

Gets extra parameters of the action rendered

public **start** ()

Starts rendering process enabling the output buffering

protected **_loadTemplateEngines** ()

Loads registered template engines, if none is registered it will use Phalcon\Mvc\View\Engine\Php

protected **_engineRender** (*array* \$engines, *string* \$viewPath, *boolean* \$silence, *boolean* \$mustClean, [*Phalcon\Cache\BackendInterface* \$cache])

Checks whether view exists on registered extensions and render it

public **registerEngines** (*array* \$engines)

Register templating engines

```
<?php
$this->view->registerEngines(
    [
        ".phtml" => "Phalcon\\Mvc\\View\\Engine\\Php",
        ".volt"   => "Phalcon\\Mvc\\View\\Engine\\Volt",
        ".mhtml"  => "MyCustomEngine",
    ]
);
```

public **exists** (*mixed* \$view)

Checks whether view exists

public **render** (*string* \$controllerName, *string* \$actionName, [*array* \$params])

Executes render process from dispatching data

```
<?php
// Shows recent posts view (app/views/posts/recent.phtml)
$view->start()->render("posts", "recent")->finish();
```

public **pick** (*mixed* \$renderView)

Choose a different view to render instead of last-controller/last-action

```
<?php
use Phalcon\Mvc\Controller;

class ProductsController extends Controller
{
    public function saveAction()
    {
        // Do some save stuff...

        // Then show the list view
        $this->view->pick("products/list");
    }
}
```

public **getPartial** (*mixed* \$partialPath, [*mixed* \$params])

Renders a partial view

```
<?php
// Retrieve the contents of a partial
echo $this->getPartial("shared/footer");
```

```
<?php

// Retrieve the contents of a partial with arguments
echo $this->getPartial(
    "shared/footer",
    [
        "content" => $html,
    ]
);
```

public **partial** (*mixed* \$partialPath, [*mixed* \$params])

Renders a partial view

```
<?php

// Show a partial inside another view
$this->partial("shared/footer");
```

```
<?php

// Show a partial inside another view with parameters
$this->partial(
    "shared/footer",
    [
        "content" => $html,
    ]
);
```

public *string* **getRender** (*string* \$controllerName, *string* \$actionName, [*array* \$params], [*mixed* \$configCallback])

Perform the automatic rendering returning the output as a string

```
<?php

$template = $this->view->getRender(
    "products",
    "show",
    [
        "products" => $products,
    ]
);
```

public **finish** ()

Finishes the render process by stopping the output buffering

protected **_createCache** ()

Create a Phalcon\Cache based on the internal cache options

public **isCaching** ()

Check if the component is currently caching the output content

public **getCache** ()

Returns the cache instance used to cache

public **cache** ([*mixed* \$options])

Cache the actual view render to certain level

```
<?php

$this->view->cache (
    [
        "key"      => "my-key",
        "lifetime" => 86400,
    ]
);
```

public **setContent** (*mixed* \$content)

Externally sets the view content

```
<?php

$this->view->setContent ("<h1>hello</h1>");
```

public **getContent** ()

Returns cached output from another view stage

public **getActiveRenderPath** ()

Returns the path (or paths) of the views that are currently rendered

public **disable** ()

Disables the auto-rendering process

public **enable** ()

Enables the auto-rendering process

public **reset** ()

Resets the view component to its factory default values

public **__set** (*mixed* \$key, *mixed* \$value)

Magic method to pass variables to the views

```
<?php

$this->view->products = $products;
```

public **__get** (*mixed* \$key)

Magic method to retrieve a variable passed to the view

```
<?php

echo $this->view->products;
```

public **isDisabled** ()

Whether automatic rendering is enabled

public **__isset** (*mixed* \$key)

Magic method to retrieve if a variable is set in the view

```
<?php

echo isset ($this->view->products);
```

protected **getViewsDirs** ()

Gets views directories

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\DiInjectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\DiInjectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\DiInjectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\DiInjectable*

Returns the internal event manager

Abstract class **Phalcon\Mvc\View\Engine**

extends abstract class *Phalcon\DiInjectable*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\DiInjectionAwareInterface*, *Phalcon\Mvc\View\EngineInterface*

All the template engine adapters must inherit this class. This provides basic interfacing between the engine and the *Phalcon\Mvc\View* component.

Methods

public **__construct** (*Phalcon\Mvc\View\BaseInterface* \$view, [*Phalcon\DiInterface* \$dependencyInjector])

Phalcon\Mvc\View\Engine constructor

public **getContent** ()

Returns cached output on another view stage

public *string* **partial** (*string* \$partialPath, [*array* \$params])

Renders a partial inside another view

public **getView** ()

Returns the view component related to the adapter

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\DiInjectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\DiInjectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\DiInjectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\DiInjectable*

Returns the internal event manager

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method **__get**

abstract public **render** (*mixed* \$path, *mixed* \$params, [*mixed* \$mustClean]) inherited from *Phalcon\Mvc\View\EngineInterface*

...

Class *Phalcon\Mvc\View\Engine\Php*

extends abstract class *Phalcon\Mvc\View\Engine*

implements *Phalcon\Mvc\View\EngineInterface*, *Phalcon\Di\InjectionAwareInterface*, *Phalcon\Events\EventsAwareInterface*

Adapter to use PHP itself as templating engine

Methods

public **render** (*mixed* \$path, *mixed* \$params, [*mixed* \$mustClean])

Renders a view using the template engine

public **__construct** (*Phalcon\Mvc\ViewBaseInterface* \$view, [*Phalcon\DiInterface* \$dependencyInjector]) inherited from *Phalcon\Mvc\View\Engine*

Phalcon\Mvc\View\Engine constructor

public **getContent** () inherited from *Phalcon\Mvc\View\Engine*

Returns cached output on another view stage

public *string* **partial** (*string* \$partialPath, [*array* \$params]) inherited from *Phalcon\Mvc\View\Engine*

Renders a partial inside another view

public **getView** () inherited from *Phalcon\Mvc\View\Engine*

Returns the view component related to the adapter

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di\Injectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Di\Injectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Di\Injectable*

Returns the internal event manager

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method **__get**

Class `Phalcon\Mvc\View\Engine\Volt`

extends abstract class `Phalcon\Mvc\View\Engine`

implements `Phalcon\Mvc\View\EngineInterface`, `Phalcon\Di\InjectionAwareInterface`, `Phalcon\Events\EventsAwareInterface`

Designer friendly and fast template engine for PHP written in Zephir/C

Methods

public **setOptions** (*array* \$options)

Set Volt's options

public **getOptions** ()

Return Volt's options

public **getCompiler** ()

Returns the Volt's compiler

public **render** (*mixed* \$templatePath, *mixed* \$params, [*mixed* \$mustClean])

Renders a view using the template engine

public **length** (*mixed* \$item)

Length filter. If an array/object is passed a count is performed otherwise a strlen/mb_strlen

public **isIncluded** (*mixed* \$needle, *mixed* \$haystack)

Checks if the needle is included in the haystack

public **convertEncoding** (*mixed* \$text, *mixed* \$from, *mixed* \$to)

Performs a string conversion

public **slice** (*mixed* \$value, [*mixed* \$start], [*mixed* \$end])

Extracts a slice from a string/array/traversable object value

public **sort** (*array* \$value)

Sorts an array

public **callMacro** (*mixed* \$name, [*array* \$arguments])

Checks if a macro is defined and calls it

public **__construct** (*Phalcon\Mvc\ViewBaseInterface* \$view, [*Phalcon\DiInterface* \$dependencyInjector]) inherited from *Phalcon\Mvc\View\Engine*

Phalcon\Mvc\View\Engine constructor

public **getContent** () inherited from *Phalcon\Mvc\View\Engine*

Returns cached output on another view stage

public *string* **partial** (*string* \$partialPath, [*array* \$params]) inherited from *Phalcon\Mvc\View\Engine*

Renders a partial inside another view

public **getView** () inherited from *Phalcon\Mvc\View\Engine*

Returns the view component related to the adapter

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di\Injectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Di\Injectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Di\Injectable*

Returns the internal event manager

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method __get

Class *Phalcon\Mvc\View\Engine\Volt\Compiler*

implements Phalcon\Di\InjectionAwareInterface

This class reads and compiles Volt templates into PHP plain code

```
<?php

$compiler = new \Phalcon\Mvc\View\Engine\Volt\Compiler();

$compiler->compile("views/partials/header.volt");

require $compiler->getCompiledTemplatePath();
```

Methods

public **__construct** (*Phalcon\Mvc\ViewBaseInterface* \$view)

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injector

public **getDI** ()

Returns the internal dependency injector

public **setOptions** (array \$options)

Sets the compiler options

public **setOption** (string \$option, *mixed* \$value)

Sets a single compiler option

public string **getOption** (string \$option)

Returns a compiler's option

public **getOptions** ()

Returns the compiler options

final public *mixed* **fireExtensionEvent** (string \$name, [array \$arguments])

Fires an event to registered extensions

public **addExtension** (*mixed* \$extension)

Registers a Volt's extension

public **getExtensions** ()

Returns the list of extensions registered in Volt

public **addFunction** (*mixed* \$name, *mixed* \$definition)

Register a new function in the compiler

public **getFunctions** ()

Register the user registered functions

public **addFilter** (*mixed* \$name, *mixed* \$definition)

Register a new filter in the compiler

public **getFilters** ()

Register the user registered filters

public **setUniquePrefix** (*mixed* \$prefix)

Set a unique prefix to be used as prefix for compiled variables

public **getUniquePrefix** ()

Return a unique prefix to be used as prefix for compiled variables and contexts

public **attributeReader** (*array* \$expr)

Resolves attribute reading

public **functionCall** (*array* \$expr)

Resolves function intermediate code into PHP function calls

public **resolveTest** (*array* \$test, *mixed* \$left)

Resolves filter intermediate code into a valid PHP expression

final protected **resolveFilter** (*array* \$filter, *mixed* \$left)

Resolves filter intermediate code into PHP function calls

final public **expression** (*array* \$expr)

Resolves an expression node in an AST volt tree

final protected *string* | *array* **_statementListOrExtends** (*array* \$statements)

Compiles a block of statements

public **compileForeach** (*array* \$statement, [*mixed* \$extendsMode])

Compiles a “foreach” intermediate code representation into plain PHP code

public **compileForElse** ()

Generates a ‘forelse’ PHP code

public **compileIf** (*array* \$statement, [*mixed* \$extendsMode])

Compiles a ‘if’ statement returning PHP code

public **compileElseIf** (*array* \$statement)

Compiles a “elseif” statement returning PHP code

public **compileCache** (*array* \$statement, [*mixed* \$extendsMode])

Compiles a “cache” statement returning PHP code

public **compileSet** (*array* \$statement)

Compiles a “set” statement returning PHP code

public **compileDo** (*array* \$statement)

Compiles a “do” statement returning PHP code

public **compileReturn** (*array* \$statement)

Compiles a “return” statement returning PHP code

public **compileAutoEscape** (*array* \$statement, *mixed* \$extendsMode)

Compiles a “autoescape” statement returning PHP code

public *string* **compileEcho** (*array* \$statement)

Compiles a ‘{{ ‘ ’}}’ statement returning PHP code

public **compileInclude** (*array* \$statement)

Compiles a ‘include’ statement returning PHP code

public **compileMacro** (*array* \$statement, *mixed* \$extendsMode)

Compiles macros

public *string* **compileCall** (*array* \$statement, *boolean* \$extendsMode)

Compiles calls to macros

final protected **_statementList** (*array* \$statements, [*mixed* \$extendsMode])

Traverses a statement list compiling each of its nodes

protected **_compileSource** (*mixed* \$viewCode, [*mixed* \$extendsMode])

Compiles a Volt source code returning a PHP plain version

public **compileString** (*mixed* \$viewCode, [*mixed* \$extendsMode])

Compiles a template into a string

```
<?php
echo $compiler->compileString('{{ "hello world" }}');
```

public *string* | *array* **compileFile** (*string* \$path, *string* \$compiledPath, [*boolean* \$extendsMode])

Compiles a template into a file forcing the destination path

```
<?php
$compiler->compile("views/layouts/main.volt", "views/layouts/main.volt.php");
```

public **compile** (*mixed* \$templatePath, [*mixed* \$extendsMode])

Compiles a template into a file applying the compiler options This method does not return the compiled path if the template was not compiled

```
<?php

$compiler->compile("views/layouts/main.volt");

require $compiler->getCompiledTemplatePath();
```

public **getTemplatePath** ()

Returns the path that is currently being compiled

public **getCompiledTemplatePath** ()

Returns the path to the last compiled template

public array **parse** (string \$viewCode)

Parses a Volt template returning its intermediate representation

```
<?php

print_r(
    $compiler->parse("{ 3 + 2 }")
);
```

protected **getFinalPath** (mixed \$path)

Gets the final path with VIEW

Class **Phalcon\Mvc\View\Engine\Volt\Exception**

*extends class **Phalcon\Mvc\View\Exception***

*implements **Throwable***

Methods

final private **Exception** **__clone** () inherited from **Exception**

Clone the exception

public **__construct** ([string \$message], [int \$code], [**Exception** \$previous]) inherited from **Exception**

Exception constructor

public **__wakeup** () inherited from **Exception**

...

final public string **getMessage** () inherited from **Exception**

Gets the Exception message

final public int **getCode** () inherited from **Exception**

Gets the Exception code

final public string **getFile** () inherited from **Exception**

Gets the file in which the exception occurred

final public int **getLine** () inherited from **Exception**

Gets the line in which the exception occurred

final public array **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public string **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Mvc\View\Exception](#)

extends class [Phalcon\Exception](#)

implements [Throwable](#)

Methods

final private [Exception](#) **__clone** () inherited from [Exception](#)

Clone the exception

public **__construct** ([string \$message], [int \$code], [[Exception](#) \$previous]) inherited from [Exception](#)

Exception constructor

public **__wakeup** () inherited from [Exception](#)

...

final public string **getMessage** () inherited from [Exception](#)

Gets the Exception message

final public int **getCode** () inherited from [Exception](#)

Gets the Exception code

final public string **getFile** () inherited from [Exception](#)

Gets the file in which the exception occurred

final public int **getLine** () inherited from [Exception](#)

Gets the line in which the exception occurred

final public array **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public string **__toString** () inherited from [Exception](#)

String representation of the exception

Class Phalcon\Mvc\View\Simple

extends abstract class *Phalcon\DI\Injectable*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\DI\InjectionAwareInterface*, *Phalcon\Mvc\View\ManagerInterface*

This component allows to render views without hierarchical levels

```
<?php

use Phalcon\Mvc\View\Simple as View;

$view = new View();

// Render a view
echo $view->render(
    "templates/my-view",
    [
        "some" => $param,
    ]
);

// Or with filename with extension
echo $view->render(
    "templates/my-view.volt",
    [
        "parameter" => $here,
    ]
);
```

Methods

public **getRegisteredEngines** ()

public **__construct** ([array \$options])

Phalcon\Mvc\View\Simple constructor

public **setViewsDir** (mixed \$viewsDir)

Sets views directory. Depending of your platform, always add a trailing slash or backslash

public **getViewsDir** ()

Gets views directory

public **registerEngines** (array \$engines)

Register templating engines

```
<?php

$this->view->registerEngines(
    [
        ".phtml" => "Phalcon\\Mvc\\View\\Engine\\Php",
        ".volt"  => "Phalcon\\Mvc\\View\\Engine\\Volt",
        ".mhtml" => "MyCustomEngine",
    ]
);
```

protected *array* **_loadTemplateEngines** ()

Loads registered template engines, if none is registered it will use Phalcon\Mvc\View\Engine\Php

final protected **_internalRender** (*string* \$path, *array* \$params)

Tries to render the view with every engine registered in the component

public **render** (*string* \$path, [*array* \$params])

Renders a view

public **partial** (*mixed* \$partialPath, [*mixed* \$params])

Renders a partial view

```
<?php
// Show a partial inside another view
$this->partial("shared/footer");
```

```
<?php
// Show a partial inside another view with parameters
$this->partial(
    "shared/footer",
    [
        "content" => $html,
    ]
);
```

public **setCacheOptions** (*array* \$options)

Sets the cache options

public *array* **getCacheOptions** ()

Returns the cache options

protected **_createCache** ()

Create a Phalcon\Cache based on the internal cache options

public **getCache** ()

Returns the cache instance used to cache

public **cache** ([*mixed* \$options])

Cache the actual view render to certain level

```
<?php
$this->view->cache(
    [
        "key"          => "my-key",
        "lifetime"     => 86400,
    ]
);
```

public **setParamToView** (*mixed* \$key, *mixed* \$value)

Adds parameters to views (alias of setVar)

```
<?php
$this->view->setParamToView("products", $products);
```

public **setVars** (array \$params, [*mixed* \$merge])

Set all the render params

```
<?php
$this->view->setVars (
    [
        "products" => $products,
    ]
);
```

public **setVar** (*mixed* \$key, *mixed* \$value)

Set a single view parameter

```
<?php
$this->view->setVar("products", $products);
```

public **getVar** (*mixed* \$key)

Returns a parameter previously set in the view

public array **getParamsToView** ()

Returns parameters to views

public **setContent** (*mixed* \$content)

Externally sets the view content

```
<?php
$this->view->setContent("<h1>hello</h1>");
```

public **getContent** ()

Returns cached output from another view stage

public string **getActiveRenderPath** ()

Returns the path of the view that is currently rendered

public **__set** (*mixed* \$key, *mixed* \$value)

Magic method to pass variables to the views

```
<?php
$this->view->products = $products;
```

public **__get** (*mixed* \$key)

Magic method to retrieve a variable passed to the view

```
<?php
echo $this->view->products;
```

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di\Injectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Di\Injectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Di\Injectable*

Returns the internal event manager

Abstract class **Phalcon\Paginator\Adapter**

implements Phalcon\Paginator\AdapterInterface

Methods

public **setCurrentPage** (*mixed* \$page)

Set the current page number

public **setLimit** (*mixed* \$limitRows)

Set current rows limit

public **getLimit** ()

Get current rows limit

abstract public **getPaginate** () inherited from *Phalcon\Paginator\AdapterInterface*

...

Class **Phalcon\Paginator\Adapter\Model**

extends abstract class *Phalcon\Paginator\Adapter*

implements Phalcon\Paginator\AdapterInterface

This adapter allows to paginate data using a *Phalcon\Mvc\Model* resultset as a base.

```
<?php
use Phalcon\Paginator\Adapter\Model;

$paginator = new Model(
[
    "data" => Robots::find(),
    "limit" => 25,
    "page" => $currentPage,
]
);
```

```
$paginate = $paginator->getPaginate();
```

Methods

public **__construct** (array \$config)

Phalcon\Paginator\Adapter\Model constructor

public **getPaginate** ()

Returns a slice of the resultset to show in the pagination

public **setCurrentPage** (mixed \$page) inherited from *Phalcon\Paginator\Adapter*

Set the current page number

public **setLimit** (mixed \$limitRows) inherited from *Phalcon\Paginator\Adapter*

Set current rows limit

public **getLimit** () inherited from *Phalcon\Paginator\Adapter*

Get current rows limit

Class Phalcon\Paginator\Adapter\NativeArray

extends abstract class *Phalcon\Paginator\Adapter*

implements *Phalcon\Paginator\AdapterInterface*

Pagination using a PHP array as source of data

```
<?php

use Phalcon\Paginator\Adapter\NativeArray;

$paginator = new NativeArray(
    [
        "data" => [
            ["id" => 1, "name" => "Artichoke"],
            ["id" => 2, "name" => "Carrots"],
            ["id" => 3, "name" => "Beet"],
            ["id" => 4, "name" => "Lettuce"],
            ["id" => 5, "name" => ""],
        ],
        "limit" => 2,
        "page" => $currentPage,
    ]
);
```

Methods

public **__construct** (array \$config)

Phalcon\Paginator\Adapter\NativeArray constructor

public **getPaginate** ()

Returns a slice of the resultset to show in the pagination

public **setCurrentPage** (*mixed* \$page) inherited from *Phalcon\Paginator\Adapter*

Set the current page number

public **setLimit** (*mixed* \$limitRows) inherited from *Phalcon\Paginator\Adapter*

Set current rows limit

public **getLimit** () inherited from *Phalcon\Paginator\Adapter*

Get current rows limit

Class *Phalcon\Paginator\Adapter\QueryBuilder*

extends abstract class *Phalcon\Paginator\Adapter*

implements *Phalcon\Paginator\AdapterInterface*

Pagination using a PHQL query builder as source of data

```
<?php

use Phalcon\Paginator\Adapter\QueryBuilder;

$builder = $this->modelsManager->createBuilder()
    ->columns("id, name")
    ->from("Robots")
    ->orderBy("name");

$paginator = new QueryBuilder(
    [
        "builder" => $builder,
        "limit"   => 20,
        "page"    => 1,
    ]
);
```

Methods

public **__construct** (*array* \$config)

public **getCurrentPage** ()

Get the current page number

public **setQueryBuilder** (*Phalcon\Mvc\Model\Query\Builder* \$builder)

Set query builder object

public **getQueryBuilder** ()

Get query builder object

public **getPaginate** ()

Returns a slice of the resultset to show in the pagination

public **setCurrentPage** (*mixed* \$page) inherited from *Phalcon\Paginator\Adapter*

Set the current page number

public **setLimit** (*mixed* \$limitRows) inherited from *Phalcon\Paginator\Adapter*

Set current rows limit

public **getLimit** () inherited from *Phalcon\Paginator\Adapter*

Get current rows limit

Class *Phalcon\Paginator\Exception*

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*

Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*

String representation of the exception

Class Phalcon\Queue\Beanstalk

Class to access the beanstalk queue service. Partially implements the protocol version 1.2

```

<?php

use Phalcon\Queue\Beanstalk;

$queue = new Beanstalk(
    [
        "host"      => "127.0.0.1",
        "port"      => 11300,
        "persistent" => true,
    ]
);

```

Constants

integer **DEFAULT_DELAY**

integer **DEFAULT_PRIORITY**

integer **DEFAULT_TTR**

string **DEFAULT_TUBE**

string **DEFAULT_HOST**

integer **DEFAULT_PORT**

Methods

public **__construct** ([array \$parameters])

public **connect** ()

Makes a connection to the Beanstalkd server

public **put** (*mixed* \$data, [array \$options])

Puts a job on the queue using specified tube.

public **reserve** ([*mixed* \$timeout])

Reserves/locks a ready job from the specified tube.

public **choose** (*mixed* \$tube)

Change the active tube. By default the tube is “default”.

public **watch** (*mixed* \$tube)

The watch command adds the named tube to the watch list for the current connection.

public **ignore** (*mixed* \$tube)

It removes the named tube from the watch list for the current connection.

public **pauseTube** (*mixed* \$tube, *mixed* \$delay)

Can delay any new job being reserved for a given time.

public **kick** (*mixed* \$bound)

The kick command applies only to the currently used tube.

public **stats** ()

Gives statistical information about the system as a whole.

public **statsTube** (*mixed* \$tube)

Gives statistical information about the specified tube if it exists.

public **listTubes** ()

Returns a list of all existing tubes.

public **listTubeUsed** ()

Returns the tube currently being used by the client.

public **listTubesWatched** ()

Returns a list tubes currently being watched by the client.

public **peekReady** ()

Inspect the next ready job.

public **peekBuried** ()

Return the next job in the list of buried jobs.

public **peekDelayed** ()

Return the next job in the list of buried jobs.

public **jobPeek** (*mixed* \$id)

The peek commands let the client inspect a job in the system.

final public **readStatus** ()

Reads the latest status from the Beanstalkd server

final public **readYaml** ()

Fetch a YAML payload from the Beanstalkd server

public **read** (*mixed* \$length)

Reads a packet from the socket. Prior to reading from the socket will check for availability of the connection.

protected **write** (*mixed* \$data)

Writes data to the socket. Performs a connection if none is available

public **disconnect** ()

Closes the connection to the beanstalk server.

public **quit** ()

Simply closes the connection.

Class **Phalcon\Queue\Beanstalk\Exception**

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private [Exception](#) **__clone** () inherited from [Exception](#)

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [[Exception](#) \$previous]) inherited from [Exception](#)

Exception constructor

public **__wakeup** () inherited from [Exception](#)

...

final public *string* **getMessage** () inherited from [Exception](#)

Gets the Exception message

final public *int* **getCode** () inherited from [Exception](#)

Gets the Exception code

final public *string* **getFile** () inherited from [Exception](#)

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from [Exception](#)

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Queue\Beanstalk\Job](#)

Represents a job in a beanstalk queue

Methods

public **getId** ()

public **getBody** ()

public **__construct** ([Phalcon\Queue\Beanstalk](#) \$queue, *mixed* \$id, *mixed* \$body)

public **delete** ()

Removes a job from the server entirely

public **release** ([*mixed* \$priority], [*mixed* \$delay])

The release command puts a reserved job back into the ready queue (and marks its state as “ready”) to be run by any client. It is normally used when the job fails because of a transitory error.

public **bury** ([*mixed* \$priority])

The bury command puts a job into the “buried” state. Buried jobs are put into a FIFO linked list and will not be touched by the server again until a client kicks them with the “kick” command.

public **touch** ()

The *touch* command allows a worker to request more time to work on a job. This is useful for jobs that potentially take a long time, but you still want the benefits of a TTR pulling a job away from an unresponsive worker. A worker may periodically tell the server that it’s still alive and processing a job (e.g. it may do this on *DEADLINE_SOON*). The command postpones the auto release of a reserved job until TTR seconds from when the command is issued.

public **kick** ()

Move the job to the ready queue if it is delayed or buried.

public **stats** ()

Gives statistical information about the specified job if it exists.

public **__wakeup** ()

Checks if the job has been modified after unserializing the object

Final class **Phalcon\Registry**

implements [ArrayAccess](#), [Countable](#), [Iterator](#), [Traversable](#)

A registry is a container for storing objects and values in the application space. By storing the value in a registry, the same object is always available throughout your application.

```
<?php

$registry = new \Phalcon\Registry();

// Set value
$registry->something = "something";
// or
$registry["something"] = "something";

// Get value
$value = $registry->something;
// or
$value = $registry["something"];

// Check if the key exists
$exists = isset($registry->something);
// or
$exists = isset($registry["something"]);

// Unset
unset($registry->something);
// or
unset($registry["something"]);
```

In addition to [ArrayAccess](#), [Phalcon\Registry](#) also implements [Countable](#) (`count($registry)` will return the number of elements in the registry), [Serializable](#) and [Iterator](#) (you can iterate over the registry using a `foreach` loop) interfaces. For PHP 5.4 and higher, [JsonSerializable](#) interface is implemented.

Phalcon\Registry is very fast (it is typically faster than any userspace implementation of the registry); however, this comes at a price: Phalcon\Registry is a final class and cannot be inherited from.

Though Phalcon\Registry exposes methods like `__get()`, `offsetGet()`, `count()` etc, it is not recommended to invoke them manually (these methods exist mainly to match the interfaces the registry implements): `$registry->__get("property")` is several times slower than `$registry->property`.

Internally all the magic methods (and interfaces except `JsonSerializable`) are implemented using object handlers or similar techniques: this allows to bypass relatively slow method calls.

Methods

final public **__construct** ()

Registry constructor

final public **offsetExists** (*mixed* \$offset)

Checks if the element is present in the registry

final public **offsetGet** (*mixed* \$offset)

Returns an index in the registry

final public **offsetSet** (*mixed* \$offset, *mixed* \$value)

Sets an element in the registry

final public **offsetUnset** (*mixed* \$offset)

Unsets an element in the registry

final public **count** ()

Checks how many elements are in the register

final public **next** ()

Moves cursor to next row in the registry

final public **key** ()

Gets pointer number of active row in the registry

final public **rewind** ()

Rewinds the registry cursor to its beginning

public **valid** ()

Checks if the iterator is valid

public **current** ()

Obtains the current value in the internal iterator

final public **__set** (*mixed* \$key, *mixed* \$value)

Sets an element in the registry

final public **__get** (*mixed* \$key)

Returns an index in the registry

final public **__isset** (*mixed* \$key)

...

```
final public __unset (mixed $key)
```

```
...
```

Class Phalcon\Security

implements Phalcon\Di\InjectionAwareInterface

This component provides a set of functions to improve the security in Phalcon applications

```
<?php

$login    = $this->request->getPost ("login");
$password = $this->request->getPost ("password");

$user = Users::findFirstByLogin($login);

if ($user) {
    if ($this->security->checkHash($password, $user->password)) {
        // The password is valid
    }
}
```

Constants

integer **CRYPT_DEFAULT**

integer **CRYPT_STD_DES**

integer **CRYPT_EXT_DES**

integer **CRYPT_MD5**

integer **CRYPT_BLOWFISH**

integer **CRYPT_BLOWFISH_A**

integer **CRYPT_BLOWFISH_X**

integer **CRYPT_BLOWFISH_Y**

integer **CRYPT_SHA256**

integer **CRYPT_SHA512**

Methods

```
public setWorkFactor (mixed $workFactor)
```

```
...
```

```
public getWorkFactor ()
```

```
...
```

```
public __construct ()
```

Phalcon\Security constructor

```
public setDI (Phalcon\DiInterface $dependencyInjector)
```


Sets the dependency injector

public **getDI** ()

Returns the internal dependency injector

public **setRandomBytes** (*mixed* \$randomBytes)

Sets a number of bytes to be generated by the openssl pseudo random generator

public **getRandomBytes** ()

Returns a number of bytes to be generated by the openssl pseudo random generator

public **getRandom** ()

Returns a secure random number generator instance

public **getSaltBytes** (*mixed* \$numberBytes)

Generate a >22-length pseudo random string to be used as salt for passwords

public **hash** (*mixed* \$password, [*mixed* \$workFactor])

Creates a password hash using bcrypt with a pseudo random salt

public **checkHash** (*mixed* \$password, *mixed* \$passwordHash, [*mixed* \$maxPassLength])

Checks a plain text password and its hash version to check if the password matches

public **isLegacyHash** (*mixed* \$passwordHash)

Checks if a password hash is a valid bcrypt's hash

public **getTokenKey** ()

Generates a pseudo random token key to be used as input's name in a CSRF check

public **getToken** ()

Generates a pseudo random token value to be used as input's value in a CSRF check

public **checkToken** ([*mixed* \$tokenKey], [*mixed* \$tokenValue], [*mixed* \$destroyIfValid])

Check if the CSRF token sent in the request is the same that the current in session

public **getSessionToken** ()

Returns the value of the CSRF token in session

public **destroyToken** ()

Removes the value of the CSRF token and key from session

public **computeHmac** (*mixed* \$data, *mixed* \$key, *mixed* \$algo, [*mixed* \$raw])

Computes a HMAC

public **setDefaultHash** (*mixed* \$defaultHash)

Sets the default hash

public **getDefaultHash** ()

Returns the default hash

public **hasLibreSsl** ()

Testing for LibreSSL

public **getSslVersionNumber** ()

Getting OpenSSL or LibreSSL version Parse OPENSSL_VERSION_TEXT because
OPENSSL_VERSION_NUMBER is no use for LibreSSL.

```
<?php

if ($security->getSslVersionNumber() >= 20105) {
    // ...
}
```

Class Phalcon\Security\Exception

extends class [Phalcon\Exception](#)

implements [Throwable](#)

Methods

final private [Exception](#) **__clone** () inherited from [Exception](#)

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [[Exception](#) \$previous]) inherited from [Exception](#)

Exception constructor

public **__wakeup** () inherited from [Exception](#)

...

final public *string* **getMessage** () inherited from [Exception](#)

Gets the Exception message

final public *int* **getCode** () inherited from [Exception](#)

Gets the Exception code

final public *string* **getFile** () inherited from [Exception](#)

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from [Exception](#)

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class Phalcon\Security\Random

Secure random number generator class.

Provides secure random number generator which is suitable for generating session key in HTTP cookies, etc.

It supports following secure random number generators:

- random_bytes (PHP 7)
- libsodium
- openssl, libressl
- /dev/urandom

Phalcon\Security\Random could be mainly useful for:

- Key generation (e.g. generation of complicated keys)
- Generating random passwords for new user accounts
- Encryption systems

```
<?php

$random = new \Phalcon\Security\Random();

// Random binary string
$bytes = $random->bytes();

// Random hex string
echo $random->hex(10); // a29f470508d5ccb8e289
echo $random->hex(10); // 533c2f08d5eee750e64a
echo $random->hex(11); // f362ef96cb9ffef150c9cd
echo $random->hex(12); // 95469d667475125208be45c4
echo $random->hex(13); // 05475e8af4a34f8f743ab48761

// Random base64 string
echo $random->base64(12); // XfIN81jGGuKkcE1E
echo $random->base64(12); // 3rcq39QzGK9fUqh8
echo $random->base64(); // DRcfbngL/iOo9hGGvy1TcQ==
echo $random->base64(16); // SvdhPcIHDZFad838Bb0Swg==

// Random URL-safe base64 string
echo $random->base64Safe(); // PcV6jGbJ6vfVw7hfKIFDGA
echo $random->base64Safe(); // GD8JoJhzSTrqX7Q8J6uug
echo $random->base64Safe(8); // mGyy0evy3ok
echo $random->base64Safe(null, true); // DRrAgOFkS4rvRiVHFefcQ==

// Random UUID
echo $random->uuid(); // db082997-2572-4e2c-a046-5eefe97b1235
echo $random->uuid(); // da2aa0e2-b4d0-4e3c-99f5-f5ef62c57fe2
echo $random->uuid(); // 75e6b628-c562-4117-bb76-61c4153455a9
echo $random->uuid(); // dc446df1-0848-4d05-b501-4af3c220c13d

// Random number between 0 and $len
echo $random->number(256); // 84
echo $random->number(256); // 79
echo $random->number(100); // 29
echo $random->number(300); // 40
```

```
// Random base58 string
echo $random->base58(); // 4kUgL2pdQMSCQtjE
echo $random->base58(); // Umjxqf7ZPwh765yR
echo $random->base58(24); // qoXcgmw4A9dys26HaNEdCRj9
echo $random->base58(7); // 774SJD3vgP
```

This class partially borrows SecureRandom library from Ruby

Methods

public **bytes** ([mixed \$len])

Generates a random binary string The *Random::bytes* method returns a string and accepts as input an int representing the length in bytes to be returned. If \$len is not specified, 16 is assumed. It may be larger in future. The result may contain any byte: “x00” - “xFF”.

```
<?php

$random = new \Phalcon\Security\Random();

$bytes = $random->bytes();
var_dump(bin2hex($bytes));
// Possible output: string(32) "00f6c04b144b41fad6a59111c126e1ee"
```

public **hex** ([mixed \$len])

Generates a random hex string If \$len is not specified, 16 is assumed. It may be larger in future. The length of the result string is usually greater of \$len.

```
<?php

$random = new \Phalcon\Security\Random();

echo $random->hex(10); // a29f470508d5ccb8e289
```

public **base58** ([mixed \$n])

Generates a random base58 string If \$len is not specified, 16 is assumed. It may be larger in future. The result may contain alphanumeric characters except 0, O, I and l. It is similar to Base64 but has been modified to avoid both non-alphanumeric characters and letters which might look ambiguous when printed.

```
<?php

$random = new \Phalcon\Security\Random();

echo $random->base58(); // 4kUgL2pdQMSCQtjE
```

public **base64** ([mixed \$len])

Generates a random base64 string If \$len is not specified, 16 is assumed. It may be larger in future. The length of the result string is usually greater of \$len. Size formula: $4 * (\$len / 3)$ and this need to be rounded up to a multiple of 4.

```
<?php

$random = new \Phalcon\Security\Random();

echo $random->base64(12); // 3rcq39QzGK9fUqh8
```

public **base64Safe** (*mixed* \$len, [*mixed* \$padding])

Generates a random URL-safe base64 string If \$len is not specified, 16 is assumed. It may be larger in future. The length of the result string is usually greater of \$len. By default, padding is not generated because “=” may be used as a URL delimiter. The result may contain A-Z, a-z, 0-9, “-” and “_”. “=” is also used if \$padding is true. See RFC 3548 for the definition of URL-safe base64.

```
<?php
$random = new \Phalcon\Security\Random();

echo $random->base64Safe(); // GD8JojhzSTrqX7Q8J6uug
```

public **uuid** ()

Generates a v4 random UUID (Universally Unique Identifier) The version 4 UUID is purely random (except the version). It doesn’t contain meaningful information such as MAC address, time, etc. See RFC 4122 for details of UUID. This algorithm sets the version number (4 bits) as well as two reserved bits. All other bits (the remaining 122 bits) are set using a random or pseudorandom data source. Version 4 UUIDs have the form xxxxxxxx-xxxx-4xxx-xxxx-xxxxxxxxxxxx where x is any hexadecimal digit and y is one of 8, 9, A, or B (e.g., f47ac10b-58cc-4372-a567-0e02b2c3d479).

```
<?php
$random = new \Phalcon\Security\Random();

echo $random->uuid(); // 1378c906-64bb-4f81-a8d6-4ae1bfcdec22
```

public **number** (*mixed* \$len)

Generates a random number between 0 and \$len Returns an integer: 0 <= result <= \$len.

```
<?php
$random = new \Phalcon\Security\Random();

echo $random->number(16); // 8
```

Abstract class **Phalcon\Session\Adapter**

implements *Phalcon\Session\AdapterInterface*

Base class for Phalcon\Session adapters

Constants

integer **SESSION_ACTIVE**

integer **SESSION_NONE**

integer **SESSION_DISABLED**

Methods

public **__construct** ([*array* \$options])

Phalcon\Session\Adapter constructor

public **start** ()

Starts the session (if headers are already sent the session will not be started)

public **setOptions** (*array* \$options)

Sets session's options

```
<?php

$session->setOptions (
    [
        "uniqueId" => "my-private-app",
    ]
);
```

public **getOptions** ()

Get internal options

public **setName** (*mixed* \$name)

Set session name

public **getName** ()

Get session name

public **regenerateId** ([*mixed* \$deleteOldSession])

public **get** (*mixed* \$index, [*mixed* \$defaultValue], [*mixed* \$remove])

Gets a session variable from an application context

```
<?php

$session->get ("auth", "yes");
```

public **set** (*mixed* \$index, *mixed* \$value)

Sets a session variable in an application context

```
<?php

$session->set ("auth", "yes");
```

public **has** (*mixed* \$index)

Check whether a session variable is set in an application context

```
<?php

var_dump (
    $session->has ("auth")
);
```

public **remove** (*mixed* \$index)

Removes a session variable from an application context

```
<?php
$session->remove("auth");
```

public **getId** ()

Returns active session id

```
<?php
echo $session->getId();
```

public **setId** (mixed \$id)

Set the current session id

```
<?php
$session->setId($id);
```

public **isStarted** ()

Check whether the session has been started

```
<?php
var_dump (
    $session->isStarted()
);
```

public **destroy** ([mixed \$removeData])

Destroys the active session

```
<?php
var_dump (
    $session->destroy()
);

var_dump (
    $session->destroy(true)
);
```

public **status** ()

Returns the status of the current session.

```
<?php
var_dump (
    $session->status()
);

if ($session->status() !== $session::SESSION_ACTIVE) {
    $session->start();
}
```

public **__get** (mixed \$index)

Alias: Gets a session variable from an application context

public **__set** (*mixed* \$index, *mixed* \$value)

Alias: Sets a session variable in an application context

public **__isset** (*mixed* \$index)

Alias: Check whether a session variable is set in an application context

public **__unset** (*mixed* \$index)

Alias: Removes a session variable from an application context

public **__destruct** ()

...

Class `Phalcon\Session\Adapter\Files`

extends abstract class `Phalcon\Session\Adapter`

implements `Phalcon\Session\AdapterInterface`

Constants

integer **SESSION_ACTIVE**

integer **SESSION_NONE**

integer **SESSION_DISABLED**

Methods

public **__construct** ([*array* \$options]) inherited from `Phalcon\Session\Adapter`

Phalcon\Session\Adapter constructor

public **start** () inherited from `Phalcon\Session\Adapter`

Starts the session (if headers are already sent the session will not be started)

public **setOptions** (*array* \$options) inherited from `Phalcon\Session\Adapter`

Sets session's options

```
<?php
$session->setOptions (
    [
        "uniqueId" => "my-private-app",
    ]
);
```

public **getOptions** () inherited from `Phalcon\Session\Adapter`

Get internal options

public **setName** (*mixed* \$name) inherited from `Phalcon\Session\Adapter`

Set session name

public **getName** () inherited from *Phalcon\Session\Adapter*

Get session name

public **regenerateId** ([*mixed* \$deleteOldSession]) inherited from *Phalcon\Session\Adapter*

public **get** (*mixed* \$index, [*mixed* \$defaultValue], [*mixed* \$remove]) inherited from *Phalcon\Session\Adapter*

Gets a session variable from an application context

```
<?php
$session->get("auth", "yes");
```

public **set** (*mixed* \$index, *mixed* \$value) inherited from *Phalcon\Session\Adapter*

Sets a session variable in an application context

```
<?php
$session->set("auth", "yes");
```

public **has** (*mixed* \$index) inherited from *Phalcon\Session\Adapter*

Check whether a session variable is set in an application context

```
<?php
var_dump(
    $session->has("auth")
);
```

public **remove** (*mixed* \$index) inherited from *Phalcon\Session\Adapter*

Removes a session variable from an application context

```
<?php
$session->remove("auth");
```

public **getId** () inherited from *Phalcon\Session\Adapter*

Returns active session id

```
<?php
echo $session->getId();
```

public **setId** (*mixed* \$id) inherited from *Phalcon\Session\Adapter*

Set the current session id

```
<?php
$session->setId($id);
```

public **isStarted** () inherited from *Phalcon\Session\Adapter*

Check whether the session has been started

```
<?php
var_dump (
    $session->isStarted()
);
```

public **destroy** ([mixed \$removeData]) inherited from *Phalcon\Session\Adapter*

Destroys the active session

```
<?php
var_dump (
    $session->destroy()
);

var_dump (
    $session->destroy(true)
);
```

public **status** () inherited from *Phalcon\Session\Adapter*

Returns the status of the current session.

```
<?php
var_dump (
    $session->status()
);

if ($session->status() !== $session::SESSION_ACTIVE) {
    $session->start();
}
```

public **__get** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Gets a session variable from an application context

public **__set** (mixed \$index, mixed \$value) inherited from *Phalcon\Session\Adapter*

Alias: Sets a session variable in an application context

public **__isset** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Check whether a session variable is set in an application context

public **__unset** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Removes a session variable from an application context

public **__destruct** () inherited from *Phalcon\Session\Adapter*

...

Class *Phalcon\Session\Adapter\Libmemcached*

extends abstract class *Phalcon\Session\Adapter*

implements *Phalcon\Session\AdapterInterface*

This adapter store sessions in libmemcached

```

<?php

use Phalcon\Session\Adapter\Libmemcached;

$session = new Libmemcached(
    [
        "servers" => [
            [
                "host"    => "localhost",
                "port"    => 11211,
                "weight" => 1,
            ],
        ],
        "client" => [
            \Memcached::OPT_HASH           => \Memcached::HASH_MD5,
            \Memcached::OPT_PREFIX_KEY => "prefix.",
        ],
        "lifetime" => 3600,
        "prefix"   => "my_",
    ]
);

$session->start();

$session->set("var", "some-value");

echo $session->get("var");

```

Constants

integer **SESSION_ACTIVE**

integer **SESSION_NONE**

integer **SESSION_DISABLED**

Methods

public **getLibmemcached** ()

...

public **getLifetime** ()

...

public **__construct** (array \$options)

Phalcon\Session\Adapter\Libmemcached constructor

public **open** ()

...

public **close** ()

...

public **read** (mixed \$sessionId)

public **write** (*mixed* \$sessionId, *mixed* \$data)

public **destroy** ([*mixed* \$sessionId])

public **gc** ()

public **start** () inherited from *Phalcon\Session\Adapter*

Starts the session (if headers are already sent the session will not be started)

public **setOptions** (array \$options) inherited from *Phalcon\Session\Adapter*

Sets session's options

```
<?php

$session->setOptions (
    [
        "uniqueId" => "my-private-app",
    ]
);
```

public **getOptions** () inherited from *Phalcon\Session\Adapter*

Get internal options

public **setName** (*mixed* \$name) inherited from *Phalcon\Session\Adapter*

Set session name

public **getName** () inherited from *Phalcon\Session\Adapter*

Get session name

public **regenerateId** ([*mixed* \$deleteOldSession]) inherited from *Phalcon\Session\Adapter*

public **get** (*mixed* \$index, [*mixed* \$defaultValue], [*mixed* \$remove]) inherited from *Phalcon\Session\Adapter*

Gets a session variable from an application context

```
<?php

$session->get ("auth", "yes");
```

public **set** (*mixed* \$index, *mixed* \$value) inherited from *Phalcon\Session\Adapter*

Sets a session variable in an application context

```
<?php

$session->set ("auth", "yes");
```

public **has** (*mixed* \$index) inherited from *Phalcon\Session\Adapter*

Check whether a session variable is set in an application context

```
<?php

var_dump (
    $session->has ("auth")
);
```

public **remove** (*mixed* \$index) inherited from *Phalcon\Session\Adapter*

Removes a session variable from an application context

```
<?php
$session->remove("auth");
```

public **getId** () inherited from *Phalcon\Session\Adapter*

Returns active session id

```
<?php
echo $session->getId();
```

public **setId** (mixed \$id) inherited from *Phalcon\Session\Adapter*

Set the current session id

```
<?php
$session->setId($id);
```

public **isStarted** () inherited from *Phalcon\Session\Adapter*

Check whether the session has been started

```
<?php
var_dump(
    $session->isStarted()
);
```

public **status** () inherited from *Phalcon\Session\Adapter*

Returns the status of the current session.

```
<?php
var_dump(
    $session->status()
);

if ($session->status() !== $session::SESSION_ACTIVE) {
    $session->start();
}
```

public **__get** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Gets a session variable from an application context

public **__set** (mixed \$index, mixed \$value) inherited from *Phalcon\Session\Adapter*

Alias: Sets a session variable in an application context

public **__isset** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Check whether a session variable is set in an application context

public **__unset** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Removes a session variable from an application context

public **__destruct** () inherited from *Phalcon\Session\Adapter*

...

Class Phalcon\Session\Adapter\Memcache

extends abstract class *Phalcon\Session\Adapter*

implements *Phalcon\Session\AdapterInterface*

This adapter store sessions in memcache

```
<?php

use Phalcon\Session\Adapter\Memcache;

$session = new Memcache (
    [
        "uniqueId" => "my-private-app",
        "host"      => "127.0.0.1",
        "port"      => 11211,
        "persistent" => true,
        "lifetime"  => 3600,
        "prefix"    => "my_",
    ]
);

$session->start ();

$session->set ("var", "some-value");

echo $session->get ("var");
```

Constants

integer **SESSION_ACTIVE**

integer **SESSION_NONE**

integer **SESSION_DISABLED**

Methods

public **getMemcache** ()

...

public **getLifetime** ()

...

public **__construct** ([array \$options])

Phalcon\Session\Adapter\Memcache constructor

public **open** ()

...

public **close** ()

...

public **read** (*mixed* \$sessionId)

public **write** (*mixed* \$sessionId, *mixed* \$data)

public **destroy** ([*mixed* \$sessionId])

public **gc** ()

public **start** () inherited from *Phalcon\Session\Adapter*

Starts the session (if headers are already sent the session will not be started)

public **setOptions** (*array* \$options) inherited from *Phalcon\Session\Adapter*

Sets session's options

```
<?php

$session->setOptions (
    [
        "uniqueId" => "my-private-app",
    ]
);
```

public **getOptions** () inherited from *Phalcon\Session\Adapter*

Get internal options

public **setName** (*mixed* \$name) inherited from *Phalcon\Session\Adapter*

Set session name

public **getName** () inherited from *Phalcon\Session\Adapter*

Get session name

public **regenerateId** ([*mixed* \$deleteOldSession]) inherited from *Phalcon\Session\Adapter*

public **get** (*mixed* \$index, [*mixed* \$defaultValue], [*mixed* \$remove]) inherited from *Phalcon\Session\Adapter*

Gets a session variable from an application context

```
<?php

$session->get ("auth", "yes");
```

public **set** (*mixed* \$index, *mixed* \$value) inherited from *Phalcon\Session\Adapter*

Sets a session variable in an application context

```
<?php

$session->set ("auth", "yes");
```

public **has** (*mixed* \$index) inherited from *Phalcon\Session\Adapter*

Check whether a session variable is set in an application context

```
<?php

var_dump (
    $session->has ("auth")
);
```

public **remove** (*mixed* \$index) inherited from *Phalcon\Session\Adapter*

Removes a session variable from an application context

```
<?php
$session->remove("auth");
```

public **getId** () inherited from *Phalcon\Session\Adapter*

Returns active session id

```
<?php
echo $session->getId();
```

public **setId** (mixed \$id) inherited from *Phalcon\Session\Adapter*

Set the current session id

```
<?php
$session->setId($id);
```

public **isStarted** () inherited from *Phalcon\Session\Adapter*

Check whether the session has been started

```
<?php
var_dump(
    $session->isStarted()
);
```

public **status** () inherited from *Phalcon\Session\Adapter*

Returns the status of the current session.

```
<?php
var_dump(
    $session->status()
);

if ($session->status() !== $session::SESSION_ACTIVE) {
    $session->start();
}
```

public **__get** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Gets a session variable from an application context

public **__set** (mixed \$index, mixed \$value) inherited from *Phalcon\Session\Adapter*

Alias: Sets a session variable in an application context

public **__isset** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Check whether a session variable is set in an application context

public **__unset** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Removes a session variable from an application context

public **__destruct** () inherited from *Phalcon\Session\Adapter*

...

Class Phalcon\Session\Adapter\Redis

extends abstract class *Phalcon\Session\Adapter*

implements *Phalcon\Session\AdapterInterface*

This adapter store sessions in Redis

```

<?php

use Phalcon\Session\Adapter\Redis;

$session = new Redis (
    [
        "uniqueId" => "my-private-app",
        "host"      => "localhost",
        "port"      => 6379,
        "auth"      => "foobared",
        "persistent" => false,
        "lifetime"  => 3600,
        "prefix"    => "my",
        "index"     => 1,
    ]
);

$session->start ();

$session->set ("var", "some-value");

echo $session->get ("var");

```

Constants

integer **SESSION_ACTIVE**

integer **SESSION_NONE**

integer **SESSION_DISABLED**

Methods

public **getRedis** ()

...

public **getLifetime** ()

...

public **__construct** ([array \$options])

Phalcon\Session\Adapter\Redis constructor

public **open** ()

public **close** ()

public **read** (*mixed* \$sessionId)

public **write** (*mixed* \$sessionId, *mixed* \$data)

public **destroy** ([*mixed* \$sessionId])

public **gc** ()

public **start** () inherited from *Phalcon\Session\Adapter*

Starts the session (if headers are already sent the session will not be started)

public **setOptions** (*array* \$options) inherited from *Phalcon\Session\Adapter*

Sets session's options

```
<?php
$session->setOptions (
    [
        "uniqueId" => "my-private-app",
    ]
);
```

public **getOptions** () inherited from *Phalcon\Session\Adapter*

Get internal options

public **setName** (*mixed* \$name) inherited from *Phalcon\Session\Adapter*

Set session name

public **getName** () inherited from *Phalcon\Session\Adapter*

Get session name

public **regenerateId** ([*mixed* \$deleteOldSession]) inherited from *Phalcon\Session\Adapter*

public **get** (*mixed* \$index, [*mixed* \$defaultValue], [*mixed* \$remove]) inherited from *Phalcon\Session\Adapter*

Gets a session variable from an application context

```
<?php
$session->get ("auth", "yes");
```

public **set** (*mixed* \$index, *mixed* \$value) inherited from *Phalcon\Session\Adapter*

Sets a session variable in an application context

```
<?php
$session->set ("auth", "yes");
```

public **has** (*mixed* \$index) inherited from *Phalcon\Session\Adapter*

Check whether a session variable is set in an application context

```
<?php
var_dump (
    $session->has ("auth")
);
```

public **remove** (*mixed* \$index) inherited from *Phalcon\Session\Adapter*

Removes a session variable from an application context

```
<?php
$session->remove("auth");
```

public **getId** () inherited from *Phalcon\Session\Adapter*

Returns active session id

```
<?php
echo $session->getId();
```

public **setId** (mixed \$id) inherited from *Phalcon\Session\Adapter*

Set the current session id

```
<?php
$session->setId($id);
```

public **isStarted** () inherited from *Phalcon\Session\Adapter*

Check whether the session has been started

```
<?php
var_dump(
    $session->isStarted()
);
```

public **status** () inherited from *Phalcon\Session\Adapter*

Returns the status of the current session.

```
<?php
var_dump(
    $session->status()
);

if ($session->status() !== $session::SESSION_ACTIVE) {
    $session->start();
}
```

public **__get** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Gets a session variable from an application context

public **__set** (mixed \$index, mixed \$value) inherited from *Phalcon\Session\Adapter*

Alias: Sets a session variable in an application context

public **__isset** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Check whether a session variable is set in an application context

public **__unset** (mixed \$index) inherited from *Phalcon\Session\Adapter*

Alias: Removes a session variable from an application context

public **__destruct** () inherited from *Phalcon\Session\Adapter*

...

Class Phalcon\Session\Bag

implements *Phalcon\Di\InjectionAwareInterface*, *Phalcon\Session\BagInterface*, *IteratorAggregate*, *Traversable*, *ArrayAccess*, *Countable*

This component helps to separate session data into “namespaces”. Working by this way you can easily create groups of session variables into the application

```
<?php
$user = new \Phalcon\Session\Bag("user");

$user->name = "Kimbra Johnson";
$user->age  = 22;
```

Methods

public **__construct** (*mixed* \$name)

Phalcon\Session\Bag constructor

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the DependencyInjector container

public **getDI** ()

Returns the DependencyInjector container

public **initialize** ()

Initializes the session bag. This method must not be called directly, the class calls it when its internal data is accessed

public **destroy** ()

Destroys the session bag

```
<?php
$user->destroy();
```

public **set** (*mixed* \$property, *mixed* \$value)

Sets a value in the session bag

```
<?php
$user->set("name", "Kimbra");
```

public **__set** (*mixed* \$property, *mixed* \$value)

Magic setter to assign values to the session bag

```
<?php
$user->name = "Kimbra";
```

public **get** (*mixed* \$property, [*mixed* \$defaultValue])

Obtains a value from the session bag optionally setting a default value

```
<?php
echo $user->get("name", "Kimbra");
```

public **__get** (*mixed* \$property)

Magic getter to obtain values from the session bag

```
<?php
echo $user->name;
```

public **has** (*mixed* \$property)

Check whether a property is defined in the internal bag

```
<?php
var_dump (
    $user->has("name")
);
```

public **__isset** (*mixed* \$property)

Magic isset to check whether a property is defined in the bag

```
<?php
var_dump (
    isset($user["name"])
);
```

public **remove** (*mixed* \$property)

Removes a property from the internal bag

```
<?php
$user->remove("name");
```

public **__unset** (*mixed* \$property)

Magic unset to remove items using the array syntax

```
<?php
unset($user["name"]);
```

final public **count** ()

Return length of bag

```
<?php
echo $user->count();
```

final public **getIterator** ()

Returns the bag iterator

final public **offsetSet** (*mixed* \$property, *mixed* \$value)

...

final public **offsetExists** (*mixed* \$property)

...

final public **offsetUnset** (*mixed* \$property)

...

final public **offsetGet** (*mixed* \$property)

...

Class Phalcon\Session\Exception

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*
Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*
Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*
Gets the Exception message

final public *int* **getCode** () inherited from *Exception*
Gets the Exception code

final public *string* **getFile** () inherited from *Exception*
Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*
Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*
Gets the stack trace

final public *Exception* **getPrevious** () inherited from *Exception*
Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from *Exception*
Gets the stack trace as a string

public *string* **__toString** () inherited from *Exception*
String representation of the exception

Class Phalcon\Tag

Phalcon\Tag is designed to simplify building of HTML tags. It provides a set of helpers to generate HTML in a dynamic way. This component is an abstract class that you can extend to add more helpers.

Constants

integer **HTML32**

integer **HTML401_STRICT**

integer **HTML401_TRANSITIONAL**

integer **HTML401_FRAMESET**

integer **HTML5**

integer **XHTML10_STRICT**

integer **XHTML10_TRANSITIONAL**

integer **XHTML10_FRAMESET**

integer **XHTML11**

integer **XHTML20**

integer **XHTML5**

Methods

public static *EscaperInterface* **getEscaper** (*array* \$params)

Obtains the 'escaper' service if required

public static **renderAttributes** (*mixed* \$code, *array* \$attributes)

Renders parameters keeping order in their HTML attributes

public static **setDI** (*Phalcon\DiInterface* \$dependencyInjector)

Sets the dependency injector container.

public static **getDI** ()

Internally gets the request dispatcher

public static **getUrlService** ()

Returns a URL service from the default DI

public static **getEscaperService** ()

Returns an Escaper service from the default DI

public static **setAutoescape** (*mixed* \$autoescape)

Set autoescape mode in generated html

public static **setDefault** (*string* \$id, *string* \$value)

Assigns default values to generated tags by helpers

```
<?php

// Assigning "peter" to "name" component
Phalcon\Tag::setDefault("name", "peter");

// Later in the view
echo Phalcon\Tag::textField("name"); // Will have the value "peter" by default
```

public static **setDefault** (array \$values, [mixed \$merge])

Assigns default values to generated tags by helpers

```
<?php

// Assigning "peter" to "name" component
Phalcon\Tag::setDefault(
    [
        "name" => "peter",
    ]
);

// Later in the view
echo Phalcon\Tag::textField("name"); // Will have the value "peter" by default
```

public static **displayTo** (string \$id, string \$value)

Alias of Phalcon\Tag::setDefault

public static *boolean* **hasValue** (string \$name)

Check if a helper has a default value set using Phalcon\Tag::setDefault or value from \$_POST

public static *mixed* **getValue** (string \$name, [array \$params])

Every helper calls this function to check whether a component has a predefined value using Phalcon\Tag::setDefault or value from \$_POST

public static **resetInput** ()

Resets the request and internal values to avoid those fields will have any default value.

public static **linkTo** (array | string \$parameters, [string \$text], [boolean \$local])

Builds a HTML A tag using framework conventions

```
<?php

echo Phalcon\Tag::linkTo("signup/register", "Register Here!");

echo Phalcon\Tag::linkTo(
    [
        "signup/register",
        "Register Here!"
    ]
);

echo Phalcon\Tag::linkTo(
    [
        "signup/register",
        "Register Here!",
        "class" => "btn-primary",
    ]
);
```



```

    ]
);

echo Phalcon\Tag::linkTo("http://phalconphp.com/", "Phalcon", false);

echo Phalcon\Tag::linkTo(
[
    "http://phalconphp.com/",
    "Phalcon Home",
    false,
]
);

echo Phalcon\Tag::linkTo(
[
    "http://phalconphp.com/",
    "Phalcon Home",
    "local" => false,
]
);

echo Phalcon\Tag::linkTo(
[
    "action" => "http://phalconphp.com/",
    "text"    => "Phalcon Home",
    "local"   => false,
    "target"  => "_new"
]
);

```

final protected static *string* **_inputField** (*string* \$type, *array* \$parameters, [*boolean* \$asValue])

Builds generic INPUT tags

final protected static *string* **_inputFieldChecked** (*string* \$type, *array* \$parameters)

Builds INPUT tags that implements the checked attribute

public static *string* **colorField** (*array* \$parameters)

Builds a HTML input[type="color"] tag

public static *string* **textField** (*array* \$parameters)

Builds a HTML input[type="text"] tag

```

<?php
echo Phalcon\Tag::textField(
[
    "name",
    "size" => 30,
]
);

```

public static *string* **numericField** (*array* \$parameters)

Builds a HTML input[type="number"] tag

```

<?php

```

```
echo Phalcon\Tag::numericField(  
    [  
        "price",  
        "min" => "1",  
        "max" => "5",  
    ]  
);
```

public static *string* **rangeField** (*array* \$parameters)

Builds a HTML input[type="range"] tag

public static *string* **emailField** (*array* \$parameters)

Builds a HTML input[type="email"] tag

```
<?php  
echo Phalcon\Tag::emailField("email");
```

public static *string* **dateField** (*array* \$parameters)

Builds a HTML input[type="date"] tag

```
<?php  
echo Phalcon\Tag::dateField(  
    [  
        "born",  
        "value" => "14-12-1980",  
    ]  
);
```

public static *string* **dateTimeField** (*array* \$parameters)

Builds a HTML input[type="datetime"] tag

public static *string* **dateTimeLocalField** (*array* \$parameters)

Builds a HTML input[type="datetime-local"] tag

public static *string* **monthField** (*array* \$parameters)

Builds a HTML input[type="month"] tag

public static *string* **timeField** (*array* \$parameters)

Builds a HTML input[type="time"] tag

public static *string* **weekField** (*array* \$parameters)

Builds a HTML input[type="week"] tag

public static *string* **passwordField** (*array* \$parameters)

Builds a HTML input[type="password"] tag

```
<?php  
echo Phalcon\Tag::passwordField(  
    [  
        "name",  
        "size" => 30,
```

```
    ]
);
```

public static *string* **hiddenField** (*array* \$parameters)

Builds a HTML input[type="hidden"] tag

```
<?php

echo Phalcon\Tag::hiddenField(
    [
        "name",
        "value" => "mike",
    ]
);
```

public static *string* **fileField** (*array* \$parameters)

Builds a HTML input[type="file"] tag

```
<?php

echo Phalcon\Tag::fileField("file");
```

public static *string* **searchField** (*array* \$parameters)

Builds a HTML input[type="search"] tag

public static *string* **telField** (*array* \$parameters)

Builds a HTML input[type="tel"] tag

public static *string* **urlField** (*array* \$parameters)

Builds a HTML input[type="url"] tag

public static *string* **checkField** (*array* \$parameters)

Builds a HTML input[type="checkbox"] tag

```
<?php

echo Phalcon\Tag::checkField(
    [
        "terms",
        "value" => "Y",
    ]
);
```

Volt syntax:

```
<?php

{{ check_field("terms") }}
```

public static *string* **radioField** (*array* \$parameters)

Builds a HTML input[type="radio"] tag

```
<?php
```

```
echo Phalcon\Tag::radioField(  
    [  
        "weather",  
        "value" => "hot",  
    ]  
);
```

Volt syntax:

```
<?php  
  
{{ radio_field("Save") }}
```

public static *string* **imageInput** (*array* \$parameters)

Builds a HTML input[type="image"] tag

```
<?php  
  
echo Phalcon\Tag::imageInput(  
    [  
        "src" => "/img/button.png",  
    ]  
);
```

Volt syntax:

```
<?php  
  
{{ image_input("src": "/img/button.png") }}
```

public static *string* **submitButton** (*array* \$parameters)

Builds a HTML input[type="submit"] tag

```
<?php  
  
echo Phalcon\Tag::submitButton("Save")
```

Volt syntax:

```
<?php  
  
{{ submit_button("Save") }}
```

public static *string* **selectStatic** (*array* \$parameters, [*array* \$data])

Builds a HTML SELECT tag using a PHP array for options

```
<?php  
  
echo Phalcon\Tag::selectStatic(  
    "status",  
    [  
        "A" => "Active",  
        "I" => "Inactive",  
    ]  
);
```

public static *string* **select** (array \$parameters, [array \$data])

Builds a HTML SELECT tag using a Phalcon\Mvc\Model resultset as options

```
<?php
echo Phalcon\Tag::select (
    [
        "robotId",
        Robots::find("type = \"mechanical\""),
        "using" => ["id", "name"],
    ]
);
```

Volt syntax:

```
<?php
{{ select("robotId", robots, "using": ["id", "name"]) }}
```

public static *string* **textArea** (array \$parameters)

Builds a HTML TEXTAREA tag

```
<?php
echo Phalcon\Tag::textArea (
    [
        "comments",
        "cols" => 10,
        "rows" => 4,
    ]
);
```

Volt syntax:

```
<?php
{{ text_area("comments", "cols": 10, "rows": 4) }}
```

public static *string* **form** (array \$parameters)

Builds a HTML FORM tag

```
<?php
echo Phalcon\Tag::form("posts/save");

echo Phalcon\Tag::form(
    [
        "posts/save",
        "method" => "post",
    ]
);
```

Volt syntax:

```
<?php
```

```
{{ form("posts/save") }}
{{ form("posts/save", "method": "post") }}
```

public static **endForm** ()

Builds a HTML close FORM tag

public static **setTitle** (*mixed* \$title)

Set the title of view content

```
<?php
Phalcon\Tag::setTitle("Welcome to my Page");
```

public static **setTitleSeparator** (*mixed* \$titleSeparator)

Set the title separator of view content

```
<?php
Phalcon\Tag::setTitleSeparator("-");
```

public static **appendTitle** (*mixed* \$title)

Appends a text to current document title

public static **prependTitle** (*mixed* \$title)

Prepends a text to current document title

public static **getTitle** ([*mixed* \$tags])

Gets the current document title. The title will be automatically escaped.

```
<?php
echo Phalcon\Tag::getTitle();
```

```
<?php
{{ get_title() }}
```

public static **getTitleSeparator** ()

Gets the current document title separator

```
<?php
echo Phalcon\Tag::getTitleSeparator();
```

```
<?php
{{ get_title_separator() }}
```

public static *string* **stylesheetLink** ([*array* \$parameters], [*boolean* \$local])

Builds a LINK[rel="stylesheet"] tag

```
<?php

echo Phalcon\Tag::stylesheetLink("http://fonts.googleapis.com/css?family=Rosario",
↪false);
echo Phalcon\Tag::stylesheetLink("css/style.css");
```

Volt Syntax:

```
<?php

{{ stylesheet_link("http://fonts.googleapis.com/css?family=Rosario", false) }}
{{ stylesheet_link("css/style.css") }}
```

public static *string* **javascriptInclude** ([array \$parameters], [boolean \$local])

Builds a SCRIPT[type="javascript"] tag

```
<?php

echo Phalcon\Tag::javascriptInclude("http://ajax.googleapis.com/ajax/libs/jquery/2.2.
↪3/jquery.min.js", false);
echo Phalcon\Tag::javascriptInclude("javascript/jquery.js");
```

Volt syntax:

```
<?php

{{ javascript_include("http://ajax.googleapis.com/ajax/libs/jquery/2.2.3/jquery.min.js
↪", false) }}
{{ javascript_include("javascript/jquery.js") }}
```

public static *string* **image** ([array \$parameters], [boolean \$local])

Builds HTML IMG tags

```
<?php

echo Phalcon\Tag::image("img/bg.png");

echo Phalcon\Tag::image(
    [
        "img/photo.jpg",
        "alt" => "Some Photo",
    ]
);
```

Volt Syntax:

```
<?php

{{ image("img/bg.png") }}
{{ image("img/photo.jpg", "alt": "Some Photo") }}
{{ image("http://static.mywebsite.com/img/bg.png", false) }}
```

public static **friendlyTitle** (*mixed* \$text, [*mixed* \$separator], [*mixed* \$lowercase], [*mixed* \$replace])

Converts texts into URL-friendly titles

```
<?php
echo Phalcon\Tag::friendlyTitle("These are big important news", "-")
```

public static **setDocType** (*mixed* \$doctype)

Set the document type of content

public static **getDocType** ()

Get the document type declaration of content

public static **tagHtml** (*mixed* \$tagName, [*mixed* \$parameters], [*mixed* \$selfClose], [*mixed* \$onlyStart], [*mixed* \$useEol])

Builds a HTML tag

public static **tagHtmlClose** (*mixed* \$tagName, [*mixed* \$useEol])

Builds a HTML tag closing tag

```
<?php
echo Phalcon\Tag::tagHtmlClose("script", true);
```

Class Phalcon\Tag\Exception

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Abstract class [Phalcon\Tag\Select](#)

Generates a SELECT html tag using a static array of values or a [Phalcon\Mvc\Model](#) resultset

Methods

public static **selectField** (*array* \$parameters, [*array* \$data])

Generates a SELECT tag

private static **_optionsFromResultset** ([Phalcon\Mvc\Model\Resultset](#) \$resultset, *array* \$using, *mixed* \$value, *string* \$closeOption)

Generate the OPTION tags based on a resultset

private static **_optionsFromArray** (*array* \$data, *mixed* \$value, *string* \$closeOption)

Generate the OPTION tags based on an array

Abstract class [Phalcon\Text](#)

Provides utilities to work with texts

Constants

integer **RANDOM_ALNUM**

integer **RANDOM_ALPHA**

integer **RANDOM_HEXDEC**

integer **RANDOM_NUMERIC**

integer **RANDOM_NOZERO**

Methods

public static **camelize** (*mixed* \$str, [*mixed* \$delimiter])

Converts strings to camelize style

```
<?php

echo Phalcon\Text::camelize("coco_bongo"); // CocoBongo
echo Phalcon\Text::camelize("co_co-bon_go", "-"); // Co_coBon_go
echo Phalcon\Text::camelize("co_co-bon_go", "_-"); // CoCoBonGo
```

public static **uncamelize** (*mixed* \$str, [*mixed* \$delimiter])

Uncamelize strings which are camelized

```
<?php

echo Phalcon\Text::uncamelize("CocoBongo"); // coco_bongo
echo Phalcon\Text::uncamelize("CocoBongo", "-"); // coco-bongo
```

public static **increment** (*mixed* \$str, [*mixed* \$separator])

Adds a number to a string or increment that number if it already is defined

```
<?php

echo Phalcon\Text::increment("a"); // "a_1"
echo Phalcon\Text::increment("a_1"); // "a_2"
```

public static **random** ([*mixed* \$type], [*mixed* \$length])

Generates a random string based on the given type. Type is one of the RANDOM_* constants

```
<?php

// "aloiwkqz"
echo Phalcon\Text::random(
    Phalcon\Text::RANDOM_ALNUM
);
```

public static **startsWith** (*mixed* \$str, *mixed* \$start, [*mixed* \$ignoreCase])

Check if a string starts with a given string

```
<?php

echo Phalcon\Text::startsWith("Hello", "He"); // true
echo Phalcon\Text::startsWith("Hello", "he", false); // false
echo Phalcon\Text::startsWith("Hello", "he"); // true
```

public static **endsWith** (*mixed* \$str, *mixed* \$end, [*mixed* \$ignoreCase])

Check if a string ends with a given string

```
<?php

echo Phalcon\Text::endsWith("Hello", "llo"); // true
echo Phalcon\Text::endsWith("Hello", "LLO", false); // false
echo Phalcon\Text::endsWith("Hello", "LLO"); // true
```

public static **lower** (*mixed* \$str, [*mixed* \$encoding])

Lowercases a string, this function makes use of the mbstring extension if available

```
<?php
echo Phalcon\Text::lower("HELLO"); // hello
```

public static **upper** (*mixed* \$str, [*mixed* \$encoding])

Uppercases a string, this function makes use of the mbstring extension if available

```
<?php
echo Phalcon\Text::upper("hello"); // HELLO
```

public static **reduceSlashes** (*mixed* \$str)

Reduces multiple slashes in a string to single slashes

```
<?php
echo Phalcon\Text::reduceSlashes("foo//bar/baz"); // foo/bar/baz
echo Phalcon\Text::reduceSlashes("http://foo.bar//baz/buz"); // http://foo.bar/baz/
↪buz
```

public static **concat** ()

Concatenates strings using the separator only once without duplication in places concatenation

```
<?php
$str = Phalcon\Text::concat(
    "/",
    "/tmp/",
    "/folder_1/",
    "/folder_2",
    "folder_3/"
);

// /tmp/folder_1/folder_2/folder_3/
echo $str;
```

public static **dynamic** (*mixed* \$text, [*mixed* \$leftDelimiter], [*mixed* \$rightDelimiter], [*mixed* \$separator])

Generates random text in accordance with the template

```
<?php

// Hi my name is a Bob
echo Phalcon\Text::dynamic("{Hi|Hello}, my name is a {Bob|Mark|Jon}!");

// Hi my name is a Jon
echo Phalcon\Text::dynamic("{Hi|Hello}, my name is a {Bob|Mark|Jon}!");

// Hello my name is a Bob
echo Phalcon\Text::dynamic("{Hi|Hello}, my name is a {Bob|Mark|Jon}!");

// Hello my name is a Zyxep
echo Phalcon\Text::dynamic("[Hi|Hello], my name is a [Zyxep|Mark]!", "[", "]", "/");
```

public static **underscore** (*mixed* \$text)

Makes a phrase underscored instead of spaced

```
<?php

echo Phalcon\Text::underscore("look behind"); // "look_behind"
echo Phalcon\Text::underscore("Awesome Phalcon"); // "Awesome_Phalcon"
```

public static **humanize** (*mixed* \$text)

Makes an underscored or dashed phrase human-readable

```
<?php

echo Phalcon\Text::humanize("start-a-horse"); // "start a horse"
echo Phalcon\Text::humanize("five_cats"); // "five cats"
```

Abstract class Phalcon\Translate

Abstract class Phalcon\Translate\Adapter

implements Phalcon\Translate\AdapterInterface

Base class for Phalcon\Translate adapters

Methods

public **__construct** (*array* \$options)

...

public **setInterpolator** (*Phalcon\Translate\InterpolatorInterface* \$interpolator)

...

public *string* **t** (*string* \$translateKey, [*array* \$placeholders])

Returns the translation string of the given key

public *string* **_** (*string* \$translateKey, [*array* \$placeholders])

Returns the translation string of the given key (alias of method 't')

public **offsetSet** (*string* \$offset, *string* \$value)

Sets a translation value

public **offsetExists** (*mixed* \$translateKey)

Check whether a translation key exists

public **offsetUnset** (*string* \$offset)

Unsets a translation from the dictionary

public *string* **offsetGet** (*string* \$translateKey)

Returns the translation related to the given key

protected **replacePlaceholders** (*mixed* \$translation, [*mixed* \$placeholders])

Replaces placeholders by the values passed

abstract public **query** (*mixed* \$index, [*mixed* \$placeholders]) inherited from *Phalcon\Translate\AdapterInterface*

...

abstract public **exists** (*mixed* \$index) inherited from *Phalcon\Translate\AdapterInterface*

...

Class *Phalcon\Translate\Adapter\Csv*

extends abstract class *Phalcon\Translate\Adapter*

implements *Phalcon\Translate\AdapterInterface*, *ArrayAccess*

Allows to define translation lists using CSV file

Methods

public **__construct** (*array* \$options)

Phalcon\Translate\Adapter\Csv constructor

private **_load** (*string* \$file, *int* \$length, *string* \$delimiter, *string* \$enclosure)

Load translates from file

public **query** (*mixed* \$index, [*mixed* \$placeholders])

Returns the translation related to the given key

public **exists** (*mixed* \$index)

Check whether is defined a translation key in the internal array

public **setInterpolator** (*Phalcon\Translate\InterpolatorInterface* \$interpolator) inherited from *Phalcon\Translate\Adapter*

...

public *string* **t** (*string* \$translateKey, [*array* \$placeholders]) inherited from *Phalcon\Translate\Adapter*

Returns the translation string of the given key

public *string* **_** (*string* \$translateKey, [*array* \$placeholders]) inherited from *Phalcon\Translate\Adapter*

Returns the translation string of the given key (alias of method 't')

public **offsetSet** (*string* \$offset, *string* \$value) inherited from *Phalcon\Translate\Adapter*

Sets a translation value

public **offsetExists** (*mixed* \$translateKey) inherited from *Phalcon\Translate\Adapter*

Check whether a translation key exists

public **offsetUnset** (*string* \$offset) inherited from *Phalcon\Translate\Adapter*

Unsets a translation from the dictionary

public *string* **offsetGet** (*string* \$translateKey) inherited from *Phalcon\Translate\Adapter*

Returns the translation related to the given key

protected **replacePlaceholders** (*mixed* \$translation, [*mixed* \$placeholders]) inherited from *Phalcon\Translate\Adapter*

Replaces placeholders by the values passed

Class Phalcon\Translate\Adapter\Gettext

extends abstract class *Phalcon\Translate\Adapter*

implements *Phalcon\Translate\AdapterInterface*, *ArrayAccess*

```
<?php
use Phalcon\Translate\Adapter\Gettext;

$adapter = new Gettext(
    [
        "locale"          => "de_DE.UTF-8",
        "defaultDomain"    => "translations",
        "directory"        => "/path/to/application/locales",
        "category"         => LC_MESSAGES,
    ]
);
```

Allows translate using gettext

Methods

public **getDirectory** ()

public **getDefaultDomain** ()

public **getLocale** ()

public **getCategory** ()

public **__construct** (array \$options)

Phalcon\Translate\Adapter\Gettext constructor

public **query** (mixed \$index, [mixed \$placeholders])

Returns the translation related to the given key.

```
<?php
$translator->query(" %name%", ["name" => "Phalcon"]);
```

public **exists** (mixed \$index)

Check whether is defined a translation key in the internal array

public **nquery** (mixed \$msgid1, mixed \$msgid2, mixed \$count, [mixed \$placeholders], [mixed \$domain])

The plural version of gettext(). Some languages have more than one form for plural messages dependent on the count.

public **setDomain** (mixed \$domain)

Changes the current domain (i.e. the translation file)

public **resetDomain** ()

Sets the default domain

public **setDefaultDomain** (mixed \$domain)

Sets the domain default to search within when calls are made to gettext()

public **setDirectory** (mixed \$directory)

Sets the path for a domain

```
<?php

// Set the directory path
$gettext->setDirectory("/path/to/the/messages");

// Set the domains and directories path
$gettext->setDirectory(
    [
        "messages" => "/path/to/the/messages",
        "another"   => "/path/to/the/another",
    ]
);
```

public **setLocale** (*mixed* \$category, *mixed* \$locale)

Sets locale information

```
<?php

// Set locale to Dutch
$gettext->setLocale(LC_ALL, "nl_NL");

// Try different possible locale names for german
$gettext->setLocale(LC_ALL, "de_DE@euro", "de_DE", "de", "ge");
```

protected **prepareOptions** (*array* \$options)

Validator for constructor

protected **getOptionsDefault** ()

Gets default options

public **setInterpolator** (*Phalcon\Translate\InterpolatorInterface* \$interpolator) inherited from *Phalcon\Translate\Adapter*

...

public *string* **t** (*string* \$translateKey, [*array* \$placeholders]) inherited from *Phalcon\Translate\Adapter*

Returns the translation string of the given key

public *string* **_** (*string* \$translateKey, [*array* \$placeholders]) inherited from *Phalcon\Translate\Adapter*

Returns the translation string of the given key (alias of method 't')

public **offsetSet** (*string* \$offset, *string* \$value) inherited from *Phalcon\Translate\Adapter*

Sets a translation value

public **offsetExists** (*mixed* \$translateKey) inherited from *Phalcon\Translate\Adapter*

Check whether a translation key exists

public **offsetUnset** (*string* \$offset) inherited from *Phalcon\Translate\Adapter*

Unsets a translation from the dictionary

public *string* **offsetGet** (*string* \$translateKey) inherited from *Phalcon\Translate\Adapter*

Returns the translation related to the given key

protected **replacePlaceholders** (*mixed* \$translation, [*mixed* \$placeholders]) inherited from *Phalcon\Translate\Adapter*

Replaces placeholders by the values passed

Class `Phalcon\Translate\Adapter\NativeArray`

extends abstract class `Phalcon\Translate\Adapter`

implements `Phalcon\Translate\AdapterInterface`, `ArrayAccess`

Allows to define translation lists using PHP arrays

Methods

public **__construct** (*array* \$options)

`Phalcon\Translate\Adapter\NativeArray` constructor

public **query** (*mixed* \$index, [*mixed* \$placeholders])

Returns the translation related to the given key

public **exists** (*mixed* \$index)

Check whether is defined a translation key in the internal array

public **setInterpolator** (*Phalcon\Translate\InterpolatorInterface* \$interpolator) inherited from *Phalcon\Translate\Adapter*

...

public *string* **t** (*string* \$translateKey, [*array* \$placeholders]) inherited from *Phalcon\Translate\Adapter*

Returns the translation string of the given key

public *string* **_** (*string* \$translateKey, [*array* \$placeholders]) inherited from *Phalcon\Translate\Adapter*

Returns the translation string of the given key (alias of method 't')

public **offsetSet** (*string* \$offset, *string* \$value) inherited from *Phalcon\Translate\Adapter*

Sets a translation value

public **offsetExists** (*mixed* \$translateKey) inherited from *Phalcon\Translate\Adapter*

Check whether a translation key exists

public **offsetUnset** (*string* \$offset) inherited from *Phalcon\Translate\Adapter*

Unsets a translation from the dictionary

public *string* **offsetGet** (*string* \$translateKey) inherited from *Phalcon\Translate\Adapter*

Returns the translation related to the given key

protected **replacePlaceholders** (*mixed* \$translation, [*mixed* \$placeholders]) inherited from *Phalcon\Translate\Adapter*

Replaces placeholders by the values passed

Class `Phalcon\Translate\Exception`

extends class `Phalcon\Exception`

implements `Throwable`

Methods

final private [Exception](#) **__clone** () inherited from [Exception](#)

Clone the exception

public **__construct** ([*string* \$message], [*int* \$code], [[Exception](#) \$previous]) inherited from [Exception](#)

Exception constructor

public **__wakeup** () inherited from [Exception](#)

...

final public *string* **getMessage** () inherited from [Exception](#)

Gets the Exception message

final public *int* **getCode** () inherited from [Exception](#)

Gets the Exception code

final public *string* **getFile** () inherited from [Exception](#)

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from [Exception](#)

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from [Exception](#)

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Translate\Interpolator\AssociativeArray](#)

implements [Phalcon\Translate\InterpolatorInterface](#)

Methods

public **replacePlaceholders** (*mixed* \$translation, [*mixed* \$placeholders])

Replaces placeholders by the values passed

Class [Phalcon\Translate\Interpolator\IndexedArray](#)

implements [Phalcon\Translate\InterpolatorInterface](#)

Methods

public **replacePlaceholders** (*mixed* \$translation, [*mixed* \$placeholders])

Replaces placeholders by the values passed

Class Phalcon\Validation

extends abstract class *Phalcon\DI\Injectable*

implements *Phalcon\Events\EventsAwareInterface*, *Phalcon\DI\InjectionAwareInterface*, *Phalcon\ValidationInterface*

Allows to validate data using custom or built-in validators

Methods

public **getData** ()

...

public **setValidators** (*mixed* \$validators)

...

public **__construct** ([*array* \$validators])

Phalcon\Validation constructor

public *Phalcon\Validation\Message\Group* **validate** ([*array* | *object* \$data], [*object* \$entity])

Validate a set of data according to a set of rules

public **add** (*mixed* \$field, *Phalcon\Validation\ValidatorInterface* \$validator)

Adds a validator to a field

public **rule** (*mixed* \$field, *Phalcon\Validation\ValidatorInterface* \$validator)

Alias of *add* method

public **rules** (*mixed* \$field, *array* \$validators)

Adds the validators to a field

public *Phalcon\Validation* **setFilters** (*string* \$field, *array* | *string* \$filters)

Adds filters to the field

public *mixed* **getFilters** ([*string* \$field])

Returns all the filters or a specific one

public **getValidators** ()

Returns the validators added to the validation

public **setEntity** (*object* \$entity)

Sets the bound entity

public *object* **getEntity** ()

Returns the bound entity

public **setDefaultMessages** ([*array* \$messages])

Adds default messages to validators

public **getDefaultMessage** (*mixed* \$type)

Get default message for validator type

public **getMessages** ()

Returns the registered validators

public **setLabels** (*array* \$labels)

Adds labels for fields

public *string* **getLabel** (*string* \$field)

Get label for field

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message)

Appends a message to the messages list

public *Phalcon\Validation* **bind** (*object* \$entity, *array* | *object* \$data)

Assigns the data to an entity The entity is used to obtain the validation values

public *mixed* **getValue** (*string* \$field)

Gets the a value to validate in the array/object data source

protected **preChecking** (*mixed* \$field, *Phalcon\Validation\ValidatorInterface* \$validator)

Internal validations, if it returns true, then skip the current validator

public **setDI** (*Phalcon\DiInterface* \$dependencyInjector) inherited from *Phalcon\Di\Injectable*

Sets the dependency injector

public **getDI** () inherited from *Phalcon\Di\Injectable*

Returns the internal dependency injector

public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager) inherited from *Phalcon\Di\Injectable*

Sets the event manager

public **getEventManager** () inherited from *Phalcon\Di\Injectable*

Returns the internal event manager

public **__get** (*mixed* \$propertyName) inherited from *Phalcon\Di\Injectable*

Magic method __get

Abstract class Phalcon\Validation\CombinedFieldsValidator

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Methods

public **__construct** ([array \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (mixed \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (mixed \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (mixed \$key, [mixed \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (mixed \$key, mixed \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

abstract public **validate** (*Phalcon\Validation* \$validation, mixed \$attribute) inherited from *Phalcon\Validation\Validator*

Executes the validation

Class Phalcon\Validation\Exception

extends class *Phalcon\Exception*

implements *Throwable*

Methods

final private *Exception* **__clone** () inherited from *Exception*

Clone the exception

public **__construct** ([string \$message], [int \$code], [*Exception* \$previous]) inherited from *Exception*

Exception constructor

public **__wakeup** () inherited from *Exception*

...

final public *string* **getMessage** () inherited from *Exception*

Gets the Exception message

final public *int* **getCode** () inherited from *Exception*

Gets the Exception code

final public *string* **getFile** () inherited from *Exception*

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from *Exception*

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from *Exception*

Gets the stack trace

final public [Exception](#) **getPrevious** () inherited from [Exception](#)

Returns previous Exception

final public [Exception](#) **getTraceAsString** () inherited from [Exception](#)

Gets the stack trace as a string

public *string* **__toString** () inherited from [Exception](#)

String representation of the exception

Class [Phalcon\Validation\Message](#)

implements [Phalcon\Validation\MessageInterface](#)

Encapsulates validation info generated in the validation process

Methods

public **__construct** (*mixed* \$message, [*mixed* \$field], [*mixed* \$type], [*mixed* \$code])

[Phalcon\Validation\Message](#) constructor

public **setType** (*mixed* \$type)

Sets message type

public **getType** ()

Returns message type

public **setMessage** (*mixed* \$message)

Sets verbose message

public **getMessage** ()

Returns verbose message

public **setField** (*mixed* \$field)

Sets field name related to message

public *mixed* **getField** ()

Returns field name related to message

public **setCode** (*mixed* \$code)

Sets code for the message

public **getCode** ()

Returns the message code

public **__toString** ()

Magic **__toString** method returns verbose message

public static **__set_state** (*array* \$message)

Magic **__set_state** helps to recover messages from serialization

Class `Phalcon\Validation\Message\Group`

implements `Countable`, `ArrayAccess`, `Iterator`, `Traversable`

Represents a group of validation messages

Methods

public **__construct** ([array \$messages])

Phalcon\Validation\Message\Group constructor

public *Phalcon\Validation\Message* **offsetGet** (int \$index)

Gets an attribute a message using the array syntax

```
<?php  
  
print_r(  
    $messages[0]  
);
```

public **offsetSet** (int \$index, *Phalcon\Validation\Message* \$message)

Sets an attribute using the array-syntax

```
<?php  
  
$messages[0] = new \Phalcon\Validation\Message("This is a message");
```

public *boolean* **offsetExists** (int \$index)

Checks if an index exists

```
<?php  
  
var_dump(  
    isset($message["database"])  
);
```

public **offsetUnset** (*mixed* \$index)

Removes a message from the list

```
<?php  
  
unset($message["database"]);
```

public **appendMessage** (*Phalcon\Validation\MessageInterface* \$message)

Appends a message to the group

```
<?php  
  
$messages->appendMessage(  
    new \Phalcon\Validation\Message("This is a message")  
);
```

public **appendMessages** (*Phalcon\Validation\MessageInterface*[] \$messages)

Appends an array of messages to the group

```
<?php
$messages->appendMessages ($messagesArray) ;
```

public array **filter** (*string* \$fieldName)

Filters the message group by field name

public **count** ()

Returns the number of messages in the list

public **rewind** ()

Rewinds the internal iterator

public **current** ()

Returns the current message in the iterator

public **key** ()

Returns the current position/key in the iterator

public **next** ()

Moves the internal iteration pointer to the next position

public **valid** ()

Check if the current message in the iterator is valid

public static *Phalcon\Validation\Message\Group* **__set_state** (*array* \$group)

Magic __set_state helps to re-build messages variable when exporting

Abstract class *Phalcon\Validation\Validator*

implements Phalcon\Validation\ValidatorInterface

This is a base class for validators

Methods

public **__construct** ([*array* \$options])

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key)

Checks if an option has been defined

public **hasOption** (*mixed* \$key)

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue])

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value)

Sets an option in the validator

abstract public **validate** (*Phalcon\Validation* \$validation, *mixed* \$attribute)

Executes the validation

Class *Phalcon\Validation\Validator\Alnum*

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Check for alphanumeric character(s)

```
<?php

use Phalcon\Validation\Validator\Alnum as AlnumValidator;

$validator->add(
    "username",
    new AlnumValidator(
        [
            "message" => ":field must contain only alphanumeric characters",
        ]
    )
);

$validator->add(
    [
        "username",
        "name",
    ],
    new AlnumValidator(
        [
            "message" => [
                "username" => "username must contain only alphanumeric characters",
                "name"      => "name must contain only alphanumeric characters",
            ],
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class Phalcon\Validation\Validator\Alpha

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Check for alphabetic character(s)

```
<?php

use Phalcon\Validation\Validator\Alpha as AlphaValidator;

$validator->add(
    "username",
    new AlphaValidator(
        [
            "message" => ":field must contain only letters",
        ]
    )
);

$validator->add(
    [
        "username",
        "name",
    ],
    new AlphaValidator(
        [
            "message" => [
                "username" => "username must contain only letters",
                "name"      => "name must contain only letters",
            ],
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class *Phalcon\Validation\Validator\Between*

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Validates that a value is between an inclusive range of two values. For a value x, the test is passed if minimum<=x<=maximum.

```
<?php

use Phalcon\Validation\Validator\Between;

$validator->add(
    "price",
    new Between(
        [
            "minimum" => 0,
            "maximum" => 100,
            "message" => "The price must be between 0 and 100",
        ]
    )
);

$validator->add(
    [
        "price",
        "amount",
    ],
    new Between(
        [
            "minimum" => [
                "price" => 0,
                "amount" => 0,
            ],
            "maximum" => [
                "price" => 100,
                "amount" => 50,
            ],
            "message" => [
                "price" => "The price must be between 0 and 100",
                "amount" => "The amount must be between 0 and 50",
            ],
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([array \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class Phalcon\Validation\Validator\Confirmation

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Checks that two values have the same value

```
<?php
use Phalcon\Validation\Validator\Confirmation;

$validator->add(
    "password",
    new Confirmation(
        [
            "message" => "Password doesn't match confirmation",
            "with"     => "confirmPassword",
        ]
    )
);

$validator->add(
    [
        "password",
        "email",
    ],
    new Confirmation(
        [
            "message" => [
                "password" => "Password doesn't match confirmation",
                "email"     => "Email doesn't match confirmation",
            ],
            "with" => [
                "password" => "confirmPassword",
                "email"     => "confirmEmail",
            ],
        ]
    )
);
```

```
        ],  
    ]  
)  
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

final protected **compare** (*mixed* \$a, *mixed* \$b)

Compare strings

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class Phalcon\Validation\Validator\CreditCard

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Checks if a value has a valid credit card number

```
<?php  
  
use Phalcon\Validation\Validator\CreditCard as CreditCardValidator;  
  
$validator->add(  
    "creditCard",  
    new CreditCardValidator(  
        [  
            "message" => "The credit card number is not valid",  
        ]  
    )  
);  
  
$validator->add(  
    [  
        "creditCard",  
        "secondCreditCard",
```

```

    ],
    new CreditCardValidator(
        [
            "message" => [
                "creditCard" => "The credit card number is not valid",
                "secondCreditCard" => "The second credit card number is not valid",
            ],
        ]
    )
);

```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

private *boolean* **verifyByLuhnAlgorithm** (*string* \$number)

is a simple checksum formula used to validate a variety of identification numbers

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class Phalcon\Validation\Validator\Date

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Checks if a value is a valid date

```

<?php

use Phalcon\Validation\Validator\Date as DateValidator;

$validator->add(
    "date",
    new DateValidator(
        [
            "format" => "d-m-Y",
            "message" => "The date is invalid",
        ]
    )
);

```

```
    )
);

$validator->add(
    [
        "date",
        "anotherDate",
    ],
    new DateValidator(
        [
            "format" => [
                "date" => "d-m-Y",
                "anotherDate" => "Y-m-d",
            ],
            "message" => [
                "date" => "The date is invalid",
                "anotherDate" => "The another date is invalid",
            ],
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

private **checkDate** (*mixed* \$value, *mixed* \$format)

...

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class Phalcon\Validation\Validator\Digit

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Check for numeric character(s)

```

<?php

use Phalcon\Validation\Validator\Digit as DigitValidator;

$validator->add(
    "height",
    new DigitValidator(
        [
            "message" => ":field must be numeric",
        ]
    )
);

$validator->add(
    [
        "height",
        "width",
    ],
    new DigitValidator(
        [
            "message" => [
                "height" => "height must be numeric",
                "width" => "width must be numeric",
            ],
        ]
    )
);

```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class Phalcon\Validation\Validator\Email

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Checks if a value has a correct e-mail format

```
<?php

use Phalcon\Validation\Validator\Email as EmailValidator;

$validator->add(
    "email",
    new EmailValidator(
        [
            "message" => "The e-mail is not valid",
        ]
    )
);

$validator->add(
    [
        "email",
        "anotherEmail",
    ],
    new EmailValidator(
        [
            "message" => [
                "email" => "The e-mail is not valid",
                "anotherEmail" => "The another e-mail is not valid",
            ],
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class Phalcon\Validation\Validator\ExclusionIn

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Check if a value is not included into a list of values

```
<?php

use Phalcon\Validation\Validator\ExclusionIn;

$validator->add(
    "status",
    new ExclusionIn(
        [
            "message" => "The status must not be A or B",
            "domain" => [
                "A",
                "B",
            ],
        ]
    )
);

$validator->add(
    [
        "status",
        "type",
    ],
    new ExclusionIn(
        [
            "message" => [
                "status" => "The status must not be A or B",
                "type" => "The type must not be 1 or "
            ],
            "domain" => [
                "status" => [
                    "A",
                    "B",
                ],
                "type" => [1, 2],
            ],
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([array \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class *Phalcon\Validation\Validator\File*

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Checks if a value has a correct file

```
<?php

use Phalcon\Validation\Validator\File as FileValidator;

$validator->add(
    "file",
    new FileValidator(
        [
            "maxSize"           => "2M",
            "messageSize"       => ":field exceeds the max filesize (:max)",
            "allowedTypes"      => [
                "image/jpeg",
                "image/png",
            ],
            "messageType"       => "Allowed file types are :types",
            "maxResolution"     => "800x600",
            "messageMaxResolution" => "Max resolution of :field is :max",
        ]
    )
);

$validator->add(
    [
        "file",
        "anotherFile",
    ],
    new FileValidator(
        [
            "maxSize" => [
                "file"       => "2M",
                "anotherFile" => "4M",
            ],
            "messageSize" => [
                "file"       => "file exceeds the max filesize 2M",
                "anotherFile" => "anotherFile exceeds the max filesize 4M",
            ],
            "allowedTypes" => [
                "file"       => [
                    "image/jpeg",
                    "image/png",
                ],
                "anotherFile" => [
                    "image/gif",
                    "image/bmp",
                ],
            ],
        ]
    )
);
```

```

        ],
        "messageType" => [
            "file" => "Allowed file types are image/jpeg and image/png",
            "anotherFile" => "Allowed file types are image/gif and image/bmp",
        ],
        "maxResolution" => [
            "file" => "800x600",
            "anotherFile" => "1024x768",
        ],
        "messageMaxResolution" => [
            "file" => "Max resolution of file is 800x600",
            "anotherFile" => "Max resolution of file is 1024x768",
        ],
    ],
)
);

```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **isEmpty** (*Phalcon\Validation* \$validation, *mixed* \$field)

Check on empty

public **__construct** ([array \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class Phalcon\Validation\Validator\Identical

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Checks if a value is identical to other

```

<?php

use Phalcon\Validation\Validator\Identical;

```

```
$validator->add(
    "terms",
    new Identical(
        [
            "accepted" => "yes",
            "message" => "Terms and conditions must be accepted",
        ]
    )
);

$validator->add(
    [
        "terms",
        "anotherTerms",
    ],
    new Identical(
        [
            "accepted" => [
                "terms" => "yes",
                "anotherTerms" => "yes",
            ],
            "message" => [
                "terms" => "Terms and conditions must be accepted",
                "anotherTerms" => "Another terms must be accepted",
            ],
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class Phalcon\Validation\Validator\InclusionIn

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Check if a value is included into a list of values

```
<?php
use Phalcon\Validation\Validator\InclusionIn;

$validator->add(
    "status",
    new InclusionIn(
        [
            "message" => "The status must be A or B",
            "domain"  => ["A", "B"],
        ]
    )
);

$validator->add(
    [
        "status",
        "type",
    ],
    new InclusionIn(
        [
            "message" => [
                "status" => "The status must be A or B",
                "type"   => "The status must be 1 or 2",
            ],
            "domain" => [
                "status" => ["A", "B"],
                "type"   => [1, 2],
            ]
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([array \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class `Phalcon\Validation\Validator\Numericality`

extends abstract class `Phalcon\Validation\Validator`

implements `Phalcon\Validation\ValidatorInterface`

Check for a valid numeric value

```
<?php

use Phalcon\Validation\Validator\Numericality;

$validator->add(
    "price",
    new Numericality(
        [
            "message" => ":field is not numeric",
        ]
    )
);

$validator->add(
    [
        "price",
        "amount",
    ],
    new Numericality(
        [
            "message" => [
                "price" => "price is not numeric",
                "amount" => "amount is not numeric",
            ]
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class **Phalcon\Validation\Validator\PresenceOf**

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Validates that a value is not null or empty string

```
<?php

use Phalcon\Validation\Validator\PresenceOf;

$validator->add(
    "name",
    new PresenceOf (
        [
            "message" => "The name is required",
        ]
    )
);

$validator->add(
    [
        "name",
        "email",
    ],
    new PresenceOf (
        [
            "message" => [
                "name" => "The name is required",
                "email" => "The email is required",
            ],
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class *Phalcon\Validation\Validator\Regex*

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Allows validate if the value of a field matches a regular expression

```
<?php

use Phalcon\Validation\Validator\Regex as RegexValidator;

$validator->add(
    "created_at",
    new RegexValidator(
        [
            "pattern" => "/^[0-9]{4}[-\/] (0[1-9]|1[12]) [-\/] (0[1-9]|1[12][0-9]|3[01])$/",
            "message" => "The creation date is invalid",
        ]
    )
);

$validator->add(
    [
        "created_at",
        "name",
    ],
    new RegexValidator(
        [
            "pattern" => [
                "created_at" => "/^[0-9]{4}[-\/] (0[1-9]|1[12]) [-\/] (0[1-9]|1[12][0-9]|3[01])$/",
                "name" => "/^[a-z]$/",
            ],
            "message" => [
                "created_at" => "The creation date is invalid",
                "name" => "The name is invalid",
            ]
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class *Phalcon\Validation\Validator\StringLength*

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Validates that a string has the specified maximum and minimum constraints The test is passed if for a string's length L , $\min \leq L \leq \max$, i.e. L must be at least \min , and at most \max .

```
<?php

use Phalcon\Validation\Validator\StringLength as StringLength;

$validation->add(
    "name_last",
    new StringLength(
        [
            "max"           => 50,
            "min"           => 2,
            "messageMaximum" => "We don't like really long names",
            "messageMinimum" => "We want more than just their initials",
        ]
    )
);

$validation->add(
    [
        "name_last",
        "name_first",
    ],
    new StringLength(
        [
            "max" => [
                "name_last" => 50,
                "name_first" => 40,
            ],
            "min" => [
                "name_last" => 2,
                "name_first" => 4,
            ],
            "messageMaximum" => [
                "name_last" => "We don't like really long last names",
                "name_first" => "We don't like really long first names",
            ],
            "messageMinimum" => [
```

```
        "name_last" => "We don't like too short last names",
        "name_first" => "We don't like too short first names",
    ]
}
)
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([array \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class Phalcon\Validation\Validator\Uniqueness

extends abstract class *Phalcon\Validation\CombinedFieldsValidator*

implements *Phalcon\Validation\ValidatorInterface*

Check that a field is unique in the related table

```
<?php

use Phalcon\Validation\Validator\Uniqueness as UniquenessValidator;

$validator->add(
    "username",
    new UniquenessValidator(
        [
            "model" => new Users(),
            "message" => ":field must be unique",
        ]
    )
);
```

Different attribute from the field:

```
<?php

$validator->add(
    "username",
    new UniquenessValidator(
        [
            "model" => new Users(),
            "attribute" => "nick",
        ]
    )
);
```

In model:

```
<?php

$validator->add(
    "username",
    new UniquenessValidator()
);
```

Combination of fields in model:

```
<?php

$validator->add(
    [
        "firstName",
        "lastName",
    ],
    new UniquenessValidator()
);
```

It is possible to convert values before validation. This is useful in situations where values need to be converted to do the database lookup:

```
<?php

$validator->add(
    "username",
    new UniquenessValidator(
        [
            "convert" => function (array $values) {
                $values["username"] = strtolower($values["username"]);

                return $values;
            }
        ]
    )
);
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

protected **isUniqueness** (*Phalcon\Validation* \$validation, *mixed* \$field)

...

protected **getColumnNameReal** (*mixed* \$record, *mixed* \$field)

The column map is used in the case to get real column name

protected **isUniquenessModel** (*mixed* \$record, *array* \$field, *array* \$values)

Uniqueness method used for model

protected **isUniquenessCollection** (*mixed* \$record, *array* \$field, *array* \$values)

Uniqueness method used for collection

public **__construct** ([*array* \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class *Phalcon\Validation\Validator\Url*

extends abstract class *Phalcon\Validation\Validator*

implements *Phalcon\Validation\ValidatorInterface*

Checks if a value has a url format

```
<?php

use Phalcon\Validation\Validator\Url as UrlValidator;

$validator->add(
    "url",
    new UrlValidator(
        [
            "message" => ":field must be a url",
        ]
    )
);

$validator->add(
    [
        "url",
        "homepage",
    ],
    new UrlValidator(
        [
```

```
        "message" => [
            "url"      => "url must be a url",
            "homepage" => "homepage must be a url",
        ]
    )
};
```

Methods

public **validate** (*Phalcon\Validation* \$validation, *mixed* \$field)

Executes the validation

public **__construct** ([array \$options]) inherited from *Phalcon\Validation\Validator*

Phalcon\Validation\Validator constructor

public **isSetOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option has been defined

public **hasOption** (*mixed* \$key) inherited from *Phalcon\Validation\Validator*

Checks if an option is defined

public **getOption** (*mixed* \$key, [*mixed* \$defaultValue]) inherited from *Phalcon\Validation\Validator*

Returns an option in the validator's options Returns null if the option hasn't set

public **setOption** (*mixed* \$key, *mixed* \$value) inherited from *Phalcon\Validation\Validator*

Sets an option in the validator

Class Phalcon\Version

This class allows to get the installed version of the framework

Constants

integer **VERSION_MAJOR**

integer **VERSION_MEDIUM**

integer **VERSION_MINOR**

integer **VERSION_SPECIAL**

integer **VERSION_SPECIAL_NUMBER**

Methods

protected static **__getVersion** ()

Area where the version number is set. The format is as follows: ABBCCDE A - Major version B - Med version (two digits) C - Min version (two digits) D - Special release: 1 = Alpha, 2 = Beta, 3 = RC, 4 = Stable E - Special release version i.e. RC1, Beta2 etc.

final protected static **_getSpecial** (*mixed* \$special)

Translates a number to a special release If Special release = 1 this function will return ALPHA

public static **get** ()

Returns the active version (string)

```
<?php
echo Phalcon\Version::get();
```

public static **getId** ()

Returns the numeric active version

```
<?php
echo Phalcon\Version::getId();
```

public static **getPart** (*mixed* \$part)

Returns a specific part of the version. If the wrong parameter is passed it will return the full version

```
<?php
echo Phalcon\Version::getPart(
    Phalcon\Version::VERSION_MAJOR
);
```

Interface Phalcon\Acl\AdapterInterface

Methods

abstract public **setDefaultAction** (*mixed* \$defaultAccess)

...

abstract public **getDefaultAction** ()

...

abstract public **setNoArgumentsDefaultAction** (*mixed* \$defaultAccess)

...

abstract public **getNoArgumentsDefaultAction** ()

...

abstract public **addRole** (*mixed* \$role, [*mixed* \$accessInherits])

...

abstract public **addInherit** (*mixed* \$roleName, *mixed* \$roleToInherit)

...

abstract public **isRole** (*mixed* \$roleName)

...

abstract public **isResource** (*mixed* \$resourceName)

```
...
abstract public addResource (mixed $resourceObject, mixed $accessList)
...
abstract public addResourceAccess (mixed $resourceName, mixed $accessList)
...
abstract public dropResourceAccess (mixed $resourceName, mixed $accessList)
...
abstract public allow (mixed $roleName, mixed $resourceName, mixed $access, [mixed $func])
...
abstract public deny (mixed $roleName, mixed $resourceName, mixed $access, [mixed $func])
...
abstract public isAllowed (mixed $roleName, mixed $resourceName, mixed $access, [array $parameters])
...
abstract public getActiveRole ()
...
abstract public getActiveResource ()
...
abstract public getActiveAccess ()
...
abstract public getRoles ()
...
abstract public getResources ()
...
```

Interface Phalcon\Acl\ResourceAware

Methods

```
abstract public getResourceName ()
...
```

Interface Phalcon\Acl\ResourceInterface

Methods

```
abstract public getName ()
...
abstract public getDescription ()
...
```

abstract public **__toString** ()

...

Interface Phalcon\Acl\RoleAware

Methods

abstract public **getRoleName** ()

...

Interface Phalcon\Acl\RoleInterface

Methods

abstract public **getName** ()

...

abstract public **getDescription** ()

...

abstract public **__toString** ()

...

Interface Phalcon\Annotations\AdapterInterface

Methods

abstract public **setReader** (*Phalcon\Annotations\ReaderInterface* \$reader)

...

abstract public **getReader** ()

...

abstract public **get** (*mixed* \$className)

...

abstract public **getMethods** (*mixed* \$className)

...

abstract public **getMethod** (*mixed* \$className, *mixed* \$methodName)

...

abstract public **getProperties** (*mixed* \$className)

...

abstract public **getProperty** (*mixed* \$className, *mixed* \$propertyName)

...

Interface Phalcon\Annotations\ReaderInterface

Methods

abstract public **parse** (*mixed* \$className)

...

abstract public static **parseDocBlock** (*mixed* \$docBlock, [*mixed* \$file], [*mixed* \$line])

...

Interface Phalcon\Assets\FilterInterface

Methods

abstract public **filter** (*mixed* \$content)

...

Interface Phalcon\Cache\BackendInterface

Methods

abstract public **start** (*mixed* \$keyName, [*mixed* \$lifetime])

...

abstract public **stop** ([*mixed* \$stopBuffer])

...

abstract public **getFrontend** ()

...

abstract public **getOptions** ()

...

abstract public **isFresh** ()

...

abstract public **isStarted** ()

...

abstract public **setLastKey** (*mixed* \$lastKey)

...

abstract public **getLastKey** ()

...

abstract public **get** (*mixed* \$keyName, [*mixed* \$lifetime])

...

abstract public **save** ([*mixed* \$keyName], [*mixed* \$content], [*mixed* \$lifetime], [*mixed* \$stopBuffer])

...

```
abstract public delete (mixed $keyName)
...
abstract public queryKeys ([mixed $prefix])
...
abstract public exists ([mixed $keyName], [mixed $lifetime])
...
```

Interface Phalcon\Cache\FrontendInterface

Methods

```
abstract public getLifetime ()
...
abstract public isBuffering ()
...
abstract public start ()
...
abstract public getContent ()
...
abstract public stop ()
...
abstract public beforeStore (mixed $data)
...
abstract public afterRetrieve (mixed $data)
...
```

Interface Phalcon\Cli\DispatcherInterface

implements Phalcon\DispatcherInterface

Methods

```
abstract public setTaskSuffix (mixed $taskSuffix)
...
abstract public setDefaultTask (mixed $taskName)
...
abstract public setTaskName (mixed $taskName)
...
abstract public getTaskName ()
```

...

abstract public **getLastTask** ()

...

abstract public **getActiveTask** ()

...

abstract public **setActionSuffix** (*mixed* \$actionSuffix) inherited from *Phalcon\DispatcherInterface*

...

abstract public **getActionSuffix** () inherited from *Phalcon\DispatcherInterface*

...

abstract public **setDefaultNamespace** (*mixed* \$defaultNamespace) inherited from *Phalcon\DispatcherInterface*

...

abstract public **setDefaultAction** (*mixed* \$actionName) inherited from *Phalcon\DispatcherInterface*

...

abstract public **setNamespaceName** (*mixed* \$namespaceName) inherited from *Phalcon\DispatcherInterface*

...

abstract public **setModuleName** (*mixed* \$moduleName) inherited from *Phalcon\DispatcherInterface*

...

abstract public **setActionName** (*mixed* \$actionName) inherited from *Phalcon\DispatcherInterface*

...

abstract public **getActionName** () inherited from *Phalcon\DispatcherInterface*

...

abstract public **setParams** (*mixed* \$params) inherited from *Phalcon\DispatcherInterface*

...

abstract public **getParams** () inherited from *Phalcon\DispatcherInterface*

...

abstract public **setParam** (*mixed* \$param, *mixed* \$value) inherited from *Phalcon\DispatcherInterface*

...

abstract public **getParam** (*mixed* \$param, [*mixed* \$filters]) inherited from *Phalcon\DispatcherInterface*

...

abstract public **hasParam** (*mixed* \$param) inherited from *Phalcon\DispatcherInterface*

...

abstract public **isFinished** () inherited from *Phalcon\DispatcherInterface*

...

abstract public **getReturnedValue** () inherited from *Phalcon\DispatcherInterface*

...

abstract public **dispatch** () inherited from *Phalcon\DispatcherInterface*

...

abstract public **forward** (*mixed* \$forward) inherited from *Phalcon\DispatcherInterface*

...

Interface Phalcon\Cli\RouterInterface

Methods

abstract public **setDefaultModule** (*mixed* \$moduleName)

...

abstract public **setDefaultTask** (*mixed* \$taskName)

...

abstract public **setDefaultAction** (*mixed* \$actionName)

...

abstract public **setDefaults** (*array* \$defaults)

...

abstract public **handle** ([*mixed* \$arguments])

...

abstract public **add** (*mixed* \$pattern, [*mixed* \$paths])

...

abstract public **getModuleName** ()

...

abstract public **getTaskName** ()

...

abstract public **getActionName** ()

...

abstract public **getParams** ()

...

abstract public **getMatchedRoute** ()

...

abstract public **getMatches** ()

...

abstract public **wasMatched** ()

...

abstract public **getRoutes** ()

...

abstract public **getRouteById** (*mixed* \$id)

...

abstract public **getRouteByName** (*mixed* \$name)

...

Interface Phalcon\Cli\Router\RouteInterface

Methods

abstract public **compilePattern** (*mixed* \$pattern)

...

abstract public **reConfigure** (*mixed* \$pattern, [*mixed* \$paths])

...

abstract public **getName** ()

...

abstract public **setName** (*mixed* \$name)

...

abstract public **getRouteId** ()

...

abstract public **getPattern** ()

...

abstract public **getCompiledPattern** ()

...

abstract public **getPaths** ()

...

abstract public **getReversedPaths** ()

...

abstract public static **reset** ()

...

Interface Phalcon\Cli\TaskInterface

Interface Phalcon\CryptInterface

Methods

abstract public **setCipher** (*mixed* \$cipher)

...

abstract public **getCipher** ()

...

```
abstract public setKey (mixed $key)
...
abstract public getKey ()
...
abstract public encrypt (mixed $text, [mixed $key])
...
abstract public decrypt (mixed $text, [mixed $key])
...
abstract public encryptBase64 (mixed $text, [mixed $key])
...
abstract public decryptBase64 (mixed $text, [mixed $key])
...
abstract public getAvailableCiphers ()
...
```

Interface Phalcon\Db\AdapterInterface

Methods

```
abstract public fetchOne (mixed $sqlQuery, [mixed $fetchMode], [mixed $placeholders])
...
abstract public fetchAll (mixed $sqlQuery, [mixed $fetchMode], [mixed $placeholders])
...
abstract public insert (mixed $table, array $values, [mixed $fields], [mixed $dataTypes])
...
abstract public update (mixed $table, mixed $fields, mixed $values, [mixed $whereCondition], [mixed $dataTypes])
...
abstract public delete (mixed $table, [mixed $whereCondition], [mixed $placeholders], [mixed $dataTypes])
...
abstract public getColumnList (mixed $columnList)
...
abstract public limit (mixed $sqlQuery, mixed $number)
...
abstract public tableExists (mixed $tableName, [mixed $schemaName])
...
abstract public viewExists (mixed $viewName, [mixed $schemaName])
...
```

```
abstract public forUpdate (mixed $sqlQuery)
...
abstract public sharedLock (mixed $sqlQuery)
...
abstract public createTable (mixed $tableName, mixed $schemaName, array $definition)
...
abstract public dropTable (mixed $tableName, [mixed $schemaName], [mixed $ifExists])
...
abstract public createView (mixed $viewName, array $definition, [mixed $schemaName])
...
abstract public dropView (mixed $viewName, [mixed $schemaName], [mixed $ifExists])
...
abstract public addColumn (mixed $tableName, mixed $schemaName, Phalcon\Db\ColumnInterface $column)
...
abstract public modifyColumn (mixed $tableName, mixed $schemaName, Phalcon\Db\ColumnInterface $column,
[Phalcon\Db\ColumnInterface $currentColumn])
...
abstract public dropColumn (mixed $tableName, mixed $schemaName, mixed $columnName)
...
abstract public addIndex (mixed $tableName, mixed $schemaName, Phalcon\Db/IndexInterface $index)
...
abstract public dropIndex (mixed $tableName, mixed $schemaName, mixed $indexName)
...
abstract public addPrimaryKey (mixed $tableName, mixed $schemaName, Phalcon\Db/IndexInterface $index)
...
abstract public dropPrimaryKey (mixed $tableName, mixed $schemaName)
...
abstract public addForeignKey (mixed $tableName, mixed $schemaName, Phalcon\Db\ReferenceInterface $reference)
...
abstract public dropForeignKey (mixed $tableName, mixed $schemaName, mixed $referenceName)
...
abstract public getColumnDefinition (Phalcon\Db\ColumnInterface $column)
...
abstract public listTables ([mixed $schemaName])
...
abstract public listViews ([mixed $schemaName])
```

```
...
abstract public getDescriptor ()
...
abstract public getConnectionId ()
...
abstract public getSQLStatement ()
...
abstract public getRealSQLStatement ()
...
abstract public getSQLVariables ()
...
abstract public getSQLBindTypes ()
...
abstract public getType ()
...
abstract public getDialectType ()
...
abstract public getDialect ()
...
abstract public connect ([array $descriptor])
...
abstract public query (mixed $sqlStatement, [mixed $placeholders], [mixed $dataTypes])
...
abstract public execute (mixed $sqlStatement, [mixed $placeholders], [mixed $dataTypes])
...
abstract public affectedRows ()
...
abstract public close ()
...
abstract public escapeIdentifier (mixed $identifier)
...
abstract public escapeString (mixed $str)
...
abstract public lastInsertId ([mixed $sequenceName])
...
abstract public begin ([mixed $nesting])
```



```
...
abstract public rollback ([mixed $nesting])
...
abstract public commit ([mixed $nesting])
...
abstract public isUnderTransaction ()
...
abstract public getInternalHandler ()
...
abstract public describeIndexes (mixed $table, [mixed $schema])
...
abstract public describeReferences (mixed $table, [mixed $schema])
...
abstract public tableOptions (mixed $tableName, [mixed $schemaName])
...
abstract public useExplicitIdValue ()
...
abstract public getDefaultIdValue ()
...
abstract public supportSequences ()
...
abstract public createSavepoint (mixed $name)
...
abstract public releaseSavepoint (mixed $name)
...
abstract public rollbackSavepoint (mixed $name)
...
abstract public setNestedTransactionsWithSavepoints (mixed $nestedTransactionsWithSavepoints)
...
abstract public isNestedTransactionsWithSavepoints ()
...
abstract public getNestedTransactionSavepointName ()
...
abstract public describeColumns (mixed $table, [mixed $schema])
...
```

Interface Phalcon\Db\ColumnInterface

Methods

abstract public **getSchemaName** ()

...

abstract public **getName** ()

...

abstract public **getType** ()

...

abstract public **getTypeReference** ()

...

abstract public **getTypeValues** ()

...

abstract public **getSize** ()

...

abstract public **getScale** ()

...

abstract public **isUnsigned** ()

...

abstract public **isNotNull** ()

...

abstract public **isPrimary** ()

...

abstract public **isAutoIncrement** ()

...

abstract public **isNumeric** ()

...

abstract public **isFirst** ()

...

abstract public **getAfterPosition** ()

...

abstract public **getBindType** ()

...

abstract public **getDefault** ()

...

abstract public **hasDefault** ()

...

abstract public static **__set_state** (*array* \$data)

...

Interface **Phalcon\Db\DialectInterface**

Methods

abstract public **limit** (*mixed* \$sqlQuery, *mixed* \$number)

...

abstract public **forUpdate** (*mixed* \$sqlQuery)

...

abstract public **sharedLock** (*mixed* \$sqlQuery)

...

abstract public **select** (*array* \$definition)

...

abstract public **getColumnList** (*array* \$columnList)

...

abstract public **getColumnDefinition** (*Phalcon\Db\ColumnInterface* \$column)

...

abstract public **addColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column)

...

abstract public **modifyColumn** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ColumnInterface* \$column, [*Phalcon\Db\ColumnInterface* \$currentColumn])

...

abstract public **dropColumn** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$columnName)

...

abstract public **addIndex** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db/IndexInterface* \$index)

...

abstract public **dropIndex** (*mixed* \$tableName, *mixed* \$schemaName, *mixed* \$indexName)

...

abstract public **addPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db/IndexInterface* \$index)

...

abstract public **dropPrimaryKey** (*mixed* \$tableName, *mixed* \$schemaName)

...

abstract public **addForeignKey** (*mixed* \$tableName, *mixed* \$schemaName, *Phalcon\Db\ReferenceInterface* \$reference)

...

```
abstract public dropForeignKey (mixed $tableName, mixed $schemaName, mixed $referenceName)
...
abstract public createTable (mixed $tableName, mixed $schemaName, array $definition)
...
abstract public createView (mixed $viewName, array $definition, [mixed $schemaName])
...
abstract public dropTable (mixed $tableName, mixed $schemaName)
...
abstract public dropView (mixed $viewName, [mixed $schemaName], [mixed $ifExists])
...
abstract public tableExists (mixed $tableName, [mixed $schemaName])
...
abstract public viewExists (mixed $viewName, [mixed $schemaName])
...
abstract public describeColumns (mixed $table, [mixed $schema])
...
abstract public listTables ([mixed $schemaName])
...
abstract public describeIndexes (mixed $table, [mixed $schema])
...
abstract public describeReferences (mixed $table, [mixed $schema])
...
abstract public tableOptions (mixed $table, [mixed $schema])
...
abstract public supportsSavepoints ()
...
abstract public supportsReleaseSavepoints ()
...
abstract public createSavepoint (mixed $name)
...
abstract public releaseSavepoint (mixed $name)
...
abstract public rollbackSavepoint (mixed $name)
...
```

Interface Phalcon\Db\IndexInterface

Methods

```
abstract public getName ()  
...  
abstract public getColumns ()  
...  
abstract public getType ()  
...  
abstract public static __set_state (array $data)  
...
```

Interface Phalcon\Db\ReferenceInterface

Methods

```
abstract public getName ()  
...  
abstract public getSchemaName ()  
...  
abstract public getReferencedSchema ()  
...  
abstract public getColumns ()  
...  
abstract public getReferencedTable ()  
...  
abstract public getReferencedColumns ()  
...  
abstract public getOnDelete ()  
...  
abstract public getOnUpdate ()  
...  
abstract public static __set_state (array $data)  
...
```

Interface Phalcon\Db\ResultInterface

Methods

abstract public **execute** ()

...

abstract public **fetch** ()

...

abstract public **fetchArray** ()

...

abstract public **fetchAll** ()

...

abstract public **numRows** ()

...

abstract public **dataSeek** (*mixed* \$number)

...

abstract public **setFetchMode** (*mixed* \$fetchMode)

...

abstract public **getInternalResult** ()

...

Interface Phalcon\DiInterface

implements [ArrayAccess](#)

Methods

abstract public **set** (*mixed* \$name, *mixed* \$definition, [*mixed* \$shared])

...

abstract public **setShared** (*mixed* \$name, *mixed* \$definition)

...

abstract public **remove** (*mixed* \$name)

...

abstract public **attempt** (*mixed* \$name, *mixed* \$definition, [*mixed* \$shared])

...

abstract public **get** (*mixed* \$name, [*mixed* \$parameters])

...

abstract public **getShared** (*mixed* \$name, [*mixed* \$parameters])

...

```
abstract public setRaw (mixed $name, Phalcon\Di\ServiceInterface $rawDefinition)
...
abstract public getRaw (mixed $name)
...
abstract public getService (mixed $name)
...
abstract public has (mixed $name)
...
abstract public wasFreshInstance ()
...
abstract public getServices ()
...
abstract public static setDefault (Phalcon\DiInterface $dependencyInjector)
...
abstract public static getDefault ()
...
abstract public static reset ()
...
abstract public offsetExists (mixed $offset) inherited from ArrayAccess
...
abstract public offsetGet (mixed $offset) inherited from ArrayAccess
...
abstract public offsetSet (mixed $offset, mixed $value) inherited from ArrayAccess
...
abstract public offsetUnset (mixed $offset) inherited from ArrayAccess
...
```

Interface *Phalcon\Di\InjectionAwareInterface*

Methods

```
abstract public setDI (Phalcon\DiInterface $dependencyInjector)
...
abstract public getDI ()
...
```

Interface Phalcon\Di\ServiceInterface

Methods

```
abstract public getName ()
...
abstract public setShared (mixed $shared)
...
abstract public isShared ()
...
abstract public setDefinition (mixed $definition)
...
abstract public getDefinition ()
...
abstract public resolve ([mixed $parameters], [Phalcon\DiInterface $dependencyInjector])
...
abstract public setParameter (mixed $position, array $parameter)
...
abstract public static __set_state (array $attributes)
...
```

Interface Phalcon\DispatcherInterface

Methods

```
abstract public setActionSuffix (mixed $actionSuffix)
...
abstract public getActionSuffix ()
...
abstract public setDefaultNamespace (mixed $defaultNamespace)
...
abstract public setDefaultAction (mixed $actionName)
...
abstract public setNamespaceName (mixed $namespaceName)
...
abstract public setModuleName (mixed $moduleName)
...
abstract public setActionName (mixed $actionName)
```


...

abstract public **getActionName** ()

...

abstract public **setParams** (*mixed* \$params)

...

abstract public **getParams** ()

...

abstract public **setParam** (*mixed* \$param, *mixed* \$value)

...

abstract public **getParam** (*mixed* \$param, [*mixed* \$filters])

...

abstract public **hasParam** (*mixed* \$param)

...

abstract public **isFinished** ()

...

abstract public **getReturnedValue** ()

...

abstract public **dispatch** ()

...

abstract public **forward** (*mixed* \$forward)

...

Interface Phalcon\EscaperInterface

Methods

abstract public **setEncoding** (*mixed* \$encoding)

...

abstract public **getEncoding** ()

...

abstract public **setHtmlQuoteType** (*mixed* \$quoteType)

...

abstract public **escapeHtml** (*mixed* \$text)

...

abstract public **escapeHtmlAttr** (*mixed* \$text)

...

abstract public **escapeCss** (*mixed* \$css)

...

abstract public **escapeJs** (*mixed* \$js)

...

abstract public **escapeUrl** (*mixed* \$url)

...

Interface Phalcon\Events\EventInterface

Methods

abstract public **getData** ()

...

abstract public **setData** ([*mixed* \$data])

...

abstract public **getType** ()

...

abstract public **setType** (*mixed* \$type)

...

abstract public **stop** ()

...

abstract public **isStopped** ()

...

abstract public **isCancelable** ()

...

Interface Phalcon\Events\EventsAwareInterface

Methods

abstract public **setEventManager** (*Phalcon\Events\ManagerInterface* \$eventsManager)

...

abstract public **getEventManager** ()

...

Interface Phalcon\Events\ManagerInterface

Methods

abstract public **attach** (*mixed* \$eventType, *mixed* \$handler)

...

```
abstract public detach (mixed $eventType, mixed $handler)
...
abstract public detachAll ([mixed $type])
...
abstract public fire (mixed $eventType, mixed $source, [mixed $data])
...
abstract public getListeners (mixed $type)
...
```

Interface Phalcon\FilterInterface

Methods

```
abstract public add (mixed $name, mixed $handler)
...
abstract public sanitize (mixed $value, mixed $filters)
...
abstract public getFilters ()
...
```

Interface Phalcon\Filter\UserFilterInterface

Methods

```
abstract public filter (mixed $value)
...
```

Interface Phalcon\FlashInterface

Methods

```
abstract public error (mixed $message)
...
abstract public notice (mixed $message)
...
abstract public success (mixed $message)
...
abstract public warning (mixed $message)
...
abstract public message (mixed $type, mixed $message)
```

...

Interface `Phalcon\Forms\ElementInterface`

Methods

abstract public **setForm** (*Phalcon\Forms\Form* \$form)

...

abstract public **getForm** ()

...

abstract public **setName** (*mixed* \$name)

...

abstract public **getName** ()

...

abstract public **setFilters** (*mixed* \$filters)

...

abstract public **addFilter** (*mixed* \$filter)

...

abstract public **getFilters** ()

...

abstract public **addValidators** (*array* \$validators, [*mixed* \$merge])

...

abstract public **addValidator** (*Phalcon\Validation\ValidatorInterface* \$validator)

...

abstract public **getValidators** ()

...

abstract public **prepareAttributes** ([*array* \$attributes], [*mixed* \$useChecked])

...

abstract public **setAttribute** (*mixed* \$attribute, *mixed* \$value)

...

abstract public **getAttribute** (*mixed* \$attribute, [*mixed* \$defaultValue])

...

abstract public **setAttributes** (*array* \$attributes)

...

abstract public **getAttributes** ()

...

abstract public **setUserOption** (*mixed* \$option, *mixed* \$value)

```
...
abstract public getUserOption (mixed $option, [mixed $defaultValue])
...
abstract public setUserOptions (array $options)
...
abstract public getUserOptions ()
...
abstract public setLabel (mixed $label)
...
abstract public getLabel ()
...
abstract public label ()
...
abstract public setDefault (mixed $value)
...
abstract public getDefault ()
...
abstract public getValue ()
...
abstract public getMessages ()
...
abstract public hasMessages ()
...
abstract public setMessages (Phalcon\Validation\Message\Group $group)
...
abstract public appendMessage (Phalcon\Validation\MessageInterface $message)
...
abstract public clear ()
...
abstract public render ([mixed $attributes])
...
```

Interface **Phalcon\Http\CookieInterface**

Methods

```
abstract public setValue (mixed $value)
```

```
...
abstract public getValue ([mixed $filters], [mixed $defaultValue])
...
abstract public send ()
...
abstract public delete ()
...
abstract public useEncryption (mixed $useEncryption)
...
abstract public isUsingEncryption ()
...
abstract public setExpiration (mixed $expire)
...
abstract public getExpiration ()
...
abstract public setPath (mixed $path)
...
abstract public getName ()
...
abstract public getPath ()
...
abstract public setDomain (mixed $domain)
...
abstract public getDomain ()
...
abstract public setSecure (mixed $secure)
...
abstract public getSecure ()
...
abstract public setHttpOnly (mixed $httpOnly)
...
abstract public getHttpOnly ()
...
```

Interface Phalcon\Http\RequestInterface

Methods

abstract public **get** ([*mixed* \$name], [*mixed* \$filters], [*mixed* \$defaultValue])

...

abstract public **getPost** ([*mixed* \$name], [*mixed* \$filters], [*mixed* \$defaultValue])

...

abstract public **getQuery** ([*mixed* \$name], [*mixed* \$filters], [*mixed* \$defaultValue])

...

abstract public **getServer** (*mixed* \$name)

...

abstract public **has** (*mixed* \$name)

...

abstract public **hasPost** (*mixed* \$name)

...

abstract public **hasPut** (*mixed* \$name)

...

abstract public **hasQuery** (*mixed* \$name)

...

abstract public **hasServer** (*mixed* \$name)

...

abstract public **getHeader** (*mixed* \$header)

...

abstract public **getScheme** ()

...

abstract public **isAjax** ()

...

abstract public **isSoapRequested** ()

...

abstract public **isSecureRequest** ()

...

abstract public **getRawBody** ()

...

abstract public **getServerAddress** ()

...

abstract public **getServerName** ()

```
...
abstract public getHttpHost ()
...
abstract public getPort ()
...
abstract public getClientAddress ([mixed $trustForwardedHeader])
...
abstract public getMethod ()
...
abstract public getUserAgent ()
...
abstract public isMethod (mixed $methods, [mixed $strict])
...
abstract public isPost ()
...
abstract public isGet ()
...
abstract public isPut ()
...
abstract public isHead ()
...
abstract public isDelete ()
...
abstract public isOptions ()
...
abstract public isPurge ()
...
abstract public isTrace ()
...
abstract public isConnect ()
...
abstract public hasFiles ([mixed $onlySuccessful])
...
abstract public getUploadedFiles ([mixed $onlySuccessful])
...
abstract public getHTTPReferer ()
```



```
...
abstract public getAcceptableContent ()
...
abstract public getBestAccept ()
...
abstract public getClientCharsets ()
...
abstract public getBestCharset ()
...
abstract public getLanguages ()
...
abstract public getBestLanguage ()
...
abstract public getBasicAuth ()
...
abstract public getDigestAuth ()
...
```

Interface Phalcon\Http\Request\FileInterface

Methods

```
abstract public getSize ()
...
abstract public getName ()
...
abstract public getTempName ()
...
abstract public getType ()
...
abstract public getRealType ()
...
abstract public moveTo (mixed $destination)
...
```

Interface Phalcon\Http\ResponseInterface

Methods

abstract public **setStatusCode** (*mixed* \$code, [*mixed* \$message])

...

abstract public **getHeaders** ()

...

abstract public **setHeader** (*mixed* \$name, *mixed* \$value)

...

abstract public **setRawHeader** (*mixed* \$header)

...

abstract public **resetHeaders** ()

...

abstract public **setExpires** ([DateTime](#) \$datetime)

...

abstract public **setNotModified** ()

...

abstract public **setContentType** (*mixed* \$contentType, [*mixed* \$charset])

...

abstract public **setContentLength** (*mixed* \$contentLength)

...

abstract public **redirect** ([*mixed* \$location], [*mixed* \$externalRedirect], [*mixed* \$statusCode])

...

abstract public **setContent** (*mixed* \$content)

...

abstract public **setJsonContent** (*mixed* \$content)

...

abstract public **appendContent** (*mixed* \$content)

...

abstract public **getContent** ()

...

abstract public **sendHeaders** ()

...

abstract public **sendCookies** ()

...

abstract public **send** ()

...

abstract public **setFileToSend** (*mixed* \$filePath, [*mixed* \$attachmentName])

...

Interface Phalcon\Http\Response\CookiesInterface

Methods

abstract public **useEncryption** (*mixed* \$useEncryption)

...

abstract public **isUsingEncryption** ()

...

abstract public **set** (*mixed* \$name, [*mixed* \$value], [*mixed* \$expire], [*mixed* \$path], [*mixed* \$secure], [*mixed* \$domain], [*mixed* \$httpOnly])

...

abstract public **get** (*mixed* \$name)

...

abstract public **has** (*mixed* \$name)

...

abstract public **delete** (*mixed* \$name)

...

abstract public **send** ()

...

abstract public **reset** ()

...

Interface Phalcon\Http\Response\HeadersInterface

Methods

abstract public **set** (*mixed* \$name, *mixed* \$value)

...

abstract public **get** (*mixed* \$name)

...

abstract public **setRaw** (*mixed* \$header)

...

abstract public **send** ()

...

abstract public **reset** ()

...

abstract public static **__set_state** (*array* \$data)

...

Interface `Phalcon\Image\AdapterInterface`

Methods

abstract public **resize** (*mixed* \$width, *mixed* \$height, *mixed* \$master)

...

abstract public **crop** (*mixed* \$width, *mixed* \$height, *mixed* \$offsetX, *mixed* \$offsetY)

...

abstract public **rotate** (*mixed* \$degrees)

...

abstract public **flip** (*mixed* \$direction)

...

abstract public **sharpen** (*mixed* \$amount)

...

abstract public **reflection** (*mixed* \$height, *mixed* \$opacity, *mixed* \$fadeIn)

...

abstract public **watermark** (*Phalcon\Image\Adapter* \$watermark, *mixed* \$offsetX, *mixed* \$offsetY, *mixed* \$opacity)

...

abstract public **text** (*mixed* \$text, *mixed* \$offsetX, *mixed* \$offsetY, *mixed* \$opacity, *mixed* \$color, *mixed* \$size, *mixed* \$fontfile)

...

abstract public **mask** (*Phalcon\Image\Adapter* \$watermark)

...

abstract public **background** (*mixed* \$color, *mixed* \$opacity)

...

abstract public **blur** (*mixed* \$radius)

...

abstract public **pixelate** (*mixed* \$amount)

...

abstract public **save** (*mixed* \$file, *mixed* \$quality)

...

abstract public **render** (*mixed* \$ext, *mixed* \$quality)

...

Interface Phalcon\Logger\AdapterInterface

Methods

abstract public **setFormatter** (*Phalcon\Logger\FormatterInterface* \$formatter)

...

abstract public **getFormatter** ()

...

abstract public **setLogLevel** (*mixed* \$level)

...

abstract public **getLogLevel** ()

...

abstract public **log** (*mixed* \$type, [*mixed* \$message], [*array* \$context])

...

abstract public **begin** ()

...

abstract public **commit** ()

...

abstract public **rollback** ()

...

abstract public **close** ()

...

abstract public **debug** (*mixed* \$message, [*array* \$context])

...

abstract public **error** (*mixed* \$message, [*array* \$context])

...

abstract public **info** (*mixed* \$message, [*array* \$context])

...

abstract public **notice** (*mixed* \$message, [*array* \$context])

...

abstract public **warning** (*mixed* \$message, [*array* \$context])

...

abstract public **alert** (*mixed* \$message, [*array* \$context])

...

abstract public **emergency** (*mixed* \$message, [*array* \$context])

...

Interface Phalcon\Logger\FormatterInterface

Methods

abstract public **format** (*mixed* \$message, *mixed* \$type, *mixed* \$timestamp, [*mixed* \$context])

...

Interface Phalcon\Mvc\CollectionInterface

Methods

abstract public **setId** (*mixed* \$id)

...

abstract public **getId** ()

...

abstract public **getReservedAttributes** ()

...

abstract public **getSource** ()

...

abstract public **setConnectionService** (*mixed* \$connectionService)

...

abstract public **getConnection** ()

...

abstract public **setDirtyState** (*mixed* \$dirtyState)

...

abstract public **getDirtyState** ()

...

abstract public static **cloneResult** (*Phalcon\Mvc\CollectionInterface* \$collection, *array* \$document)

...

abstract public **fireEvent** (*mixed* \$eventName)

...

abstract public **fireEventCancel** (*mixed* \$eventName)

...

abstract public **validationHasFailed** ()

...

abstract public **getMessages** ()

...

abstract public **appendMessage** (*Phalcon\Mvc\Model\MessageInterface* \$message)

```
...
abstract public save ()
...
abstract public static findById (mixed $id)
...
abstract public static findFirst ([array $parameters])
...
abstract public static find ([array $parameters])
...
abstract public static count ([array $parameters])
...
abstract public delete ()
...
```

Interface `Phalcon\Mvc\Collection\BehaviorInterface`

Methods

```
abstract public notify (mixed $type, Phalcon\Mvc\CollectionInterface $collection)
...
abstract public missingMethod (Phalcon\Mvc\CollectionInterface $collection, mixed $method, [mixed $arguments])
...
```

Interface `Phalcon\Mvc\Collection\ManagerInterface`

Methods

```
abstract public setCustomEventManager (Phalcon\Mvc\CollectionInterface $model, Phalcon\Events\ManagerInterface $eventsManager)
...
abstract public getCustomEventManager (Phalcon\Mvc\CollectionInterface $model)
...
abstract public initialize (Phalcon\Mvc\CollectionInterface $model)
...
abstract public isInitialized (mixed $modelName)
...
abstract public getLastInitialized ()
...
abstract public setConnectionService (Phalcon\Mvc\CollectionInterface $model, mixed $connectionService)
```

```
...
abstract public useImplicitObjectIds (Phalcon\Mvc\CollectionInterface $model, mixed $useImplicitObjectIds)
...
abstract public isUsingImplicitObjectIds (Phalcon\Mvc\CollectionInterface $model)
...
abstract public getConnection (Phalcon\Mvc\CollectionInterface $model)
...
abstract public notifyEvent (mixed $eventName, Phalcon\Mvc\CollectionInterface $model)
...
abstract public addBehavior (Phalcon\Mvc\CollectionInterface $model, Phalcon\Mvc\Collection\BehaviorInterface
$behavior)
...
```

Interface `Phalcon\Mvc\ControllerInterface`

Interface `Phalcon\Mvc\Controller\BindModelInterface`

Methods

```
abstract public static getModelName ()
...
```

Interface `Phalcon\Mvc\DispatcherInterface`

implements `Phalcon\DispatcherInterface`

Methods

```
abstract public setControllerSuffix (mixed $controllerSuffix)
...
abstract public setDefaultController (mixed $controllerName)
...
abstract public setControllerName (mixed $controllerName)
...
abstract public getControllerName ()
...
abstract public getLastController ()
...
abstract public getActiveController ()
...
```


abstract public **setActionSuffix** (*mixed* \$actionSuffix) inherited from *Phalcon\DispatcherInterface*
...
abstract public **getActionSuffix** () inherited from *Phalcon\DispatcherInterface*
...
abstract public **setDefaultNamespace** (*mixed* \$defaultNamespace) inherited from *Phalcon\DispatcherInterface*
...
abstract public **setDefaultAction** (*mixed* \$actionName) inherited from *Phalcon\DispatcherInterface*
...
abstract public **setNamespaceName** (*mixed* \$namespaceName) inherited from *Phalcon\DispatcherInterface*
...
abstract public **setModuleName** (*mixed* \$moduleName) inherited from *Phalcon\DispatcherInterface*
...
abstract public **setActionName** (*mixed* \$actionName) inherited from *Phalcon\DispatcherInterface*
...
abstract public **getActionName** () inherited from *Phalcon\DispatcherInterface*
...
abstract public **setParams** (*mixed* \$params) inherited from *Phalcon\DispatcherInterface*
...
abstract public **getParams** () inherited from *Phalcon\DispatcherInterface*
...
abstract public **setParam** (*mixed* \$param, *mixed* \$value) inherited from *Phalcon\DispatcherInterface*
...
abstract public **getParam** (*mixed* \$param, [*mixed* \$filters]) inherited from *Phalcon\DispatcherInterface*
...
abstract public **hasParam** (*mixed* \$param) inherited from *Phalcon\DispatcherInterface*
...
abstract public **isFinished** () inherited from *Phalcon\DispatcherInterface*
...
abstract public **getReturnedValue** () inherited from *Phalcon\DispatcherInterface*
...
abstract public **dispatch** () inherited from *Phalcon\DispatcherInterface*
...
abstract public **forward** (*mixed* \$forward) inherited from *Phalcon\DispatcherInterface*
...

Interface Phalcon\Mvc\EntityInterface

Methods

abstract public **readAttribute** (*mixed* \$attribute)
...
abstract public **writeAttribute** (*mixed* \$attribute, *mixed* \$value)
...

Interface Phalcon\Mvc\Micro\CollectionInterface

Methods

abstract public **setPrefix** (*mixed* \$prefix)
...
abstract public **getPrefix** ()
...
abstract public **getHandlers** ()
...
abstract public **setHandler** (*mixed* \$handler, [*mixed* \$lazy])
...
abstract public **setLazy** (*mixed* \$lazy)
...
abstract public **isLazy** ()
...
abstract public **getHandler** ()
...
abstract public **map** (*mixed* \$routeProvider, *mixed* \$handler, [*mixed* \$name])
...
abstract public **get** (*mixed* \$routeProvider, *mixed* \$handler, [*mixed* \$name])
...
abstract public **post** (*mixed* \$routeProvider, *mixed* \$handler, [*mixed* \$name])
...
abstract public **put** (*mixed* \$routeProvider, *mixed* \$handler, [*mixed* \$name])
...
abstract public **patch** (*mixed* \$routeProvider, *mixed* \$handler, [*mixed* \$name])
...
abstract public **head** (*mixed* \$routeProvider, *mixed* \$handler, [*mixed* \$name])

...

abstract public **delete** (*mixed* \$routePattern, *mixed* \$handler, [*mixed* \$name])

...

abstract public **options** (*mixed* \$routePattern, *mixed* \$handler, [*mixed* \$name])

...

Interface Phalcon\Mvc\Micro\MiddlewareInterface

Methods

abstract public **call** (*Phalcon\Mvc\Micro* \$application)

...

Interface Phalcon\Mvc\ModelInterface

Methods

abstract public **setTransaction** (*Phalcon\Mvc\Model\TransactionInterface* \$transaction)

...

abstract public **getSource** ()

...

abstract public **getSchema** ()

...

abstract public **setConnectionService** (*mixed* \$connectionService)

...

abstract public **setWriteConnectionService** (*mixed* \$connectionService)

...

abstract public **setReadConnectionService** (*mixed* \$connectionService)

...

abstract public **getReadConnectionService** ()

...

abstract public **getWriteConnectionService** ()

...

abstract public **getReadConnection** ()

...

abstract public **getWriteConnection** ()

...

abstract public **setDirtyState** (*mixed* \$dirtyState)

...

```
abstract public getDirtyState ()
...
abstract public assign (array $data, [mixed $dataColumnMap], [mixed $whiteList])
...
abstract public static cloneResultMap (mixed $base, array $data, mixed $columnMap, [mixed $dirtyState], [mixed $keepSnapshots])
...
abstract public static cloneResult (Phalcon\Mvc\ModelInterface $base, array $data, [mixed $dirtyState])
...
abstract public static cloneResultMapHydrate (array $data, mixed $columnMap, mixed $hydrationMode)
...
abstract public static find ([mixed $parameters])
...
abstract public static findFirst ([mixed $parameters])
...
abstract public static query ([Phalcon\DiInterface $dependencyInjector])
...
abstract public static count ([mixed $parameters])
...
abstract public static sum ([mixed $parameters])
...
abstract public static maximum ([mixed $parameters])
...
abstract public static minimum ([mixed $parameters])
...
abstract public static average ([mixed $parameters])
...
abstract public fireEvent (mixed $eventName)
...
abstract public fireEventCancel (mixed $eventName)
...
abstract public appendMessage (Phalcon\Mvc\Model\MessageInterface $message)
...
abstract public validationHasFailed ()
...
abstract public getMessages ()
```

```
...
abstract public save ([mixed $data], [mixed $whiteList])
...
abstract public create ([mixed $data], [mixed $whiteList])
...
abstract public update ([mixed $data], [mixed $whiteList])
...
abstract public delete ()
...
abstract public getOperationMade ()
...
abstract public refresh ()
...
abstract public skipOperation (mixed $skip)
...
abstract public getRelated (mixed $alias, [mixed $arguments])
...
abstract public setSnapshotData (array $data, [mixed $columnMap])
...
abstract public reset ()
...
```

Interface `Phalcon\Mvc\Model\BehaviorInterface`

Methods

```
abstract public notify (mixed $type, Phalcon\Mvc\ModelInterface $model)
...
abstract public missingMethod (Phalcon\Mvc\ModelInterface $model, mixed $method, [mixed $arguments])
...
```

Interface `Phalcon\Mvc\Model\CriteriaInterface`

Methods

```
abstract public setModelName (mixed $modelName)
...
abstract public getModelName ()
...
```

```
abstract public bind (array $bindParam)  
...  
abstract public bindTypes (array $bindTypes)  
...  
abstract public where (mixed $conditions)  
...  
abstract public conditions (mixed $conditions)  
...  
abstract public orderBy (mixed $orderColumns)  
...  
abstract public limit (mixed $limit, [mixed $offset])  
...  
abstract public forUpdate ([mixed $forUpdate])  
...  
abstract public sharedLock ([mixed $sharedLock])  
...  
abstract public andWhere (mixed $conditions, [mixed $bindParam], [mixed $bindTypes])  
...  
abstract public orWhere (mixed $conditions, [mixed $bindParam], [mixed $bindTypes])  
...  
abstract public betweenWhere (mixed $expr, mixed $minimum, mixed $maximum)  
...  
abstract public notBetweenWhere (mixed $expr, mixed $minimum, mixed $maximum)  
...  
abstract public inWhere (mixed $expr, array $values)  
...  
abstract public notInWhere (mixed $expr, array $values)  
...  
abstract public getWhere ()  
...  
abstract public getConditions ()  
...  
abstract public getLimit ()  
...  
abstract public getOrderBy ()  
...
```

abstract public **getParams** ()

...

abstract public **execute** ()

...

Interface **Phalcon\Mvc\Model\ManagerInterface**

Methods

abstract public **initialize** (*Phalcon\Mvc\ModelInterface* \$model)

...

abstract public **setModelSource** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$source)

...

abstract public **getModelSource** (*Phalcon\Mvc\ModelInterface* \$model)

...

abstract public **setModelSchema** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$schema)

...

abstract public **getModelSchema** (*Phalcon\Mvc\ModelInterface* \$model)

...

abstract public **setConnectionService** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$connectionService)

...

abstract public **setReadConnectionService** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$connectionService)

...

abstract public **getReadConnectionService** (*Phalcon\Mvc\ModelInterface* \$model)

...

abstract public **setWriteConnectionService** (*Phalcon\Mvc\ModelInterface* \$model, *mixed* \$connectionService)

...

abstract public **getWriteConnectionService** (*Phalcon\Mvc\ModelInterface* \$model)

...

abstract public **getReadConnection** (*Phalcon\Mvc\ModelInterface* \$model)

...

abstract public **getWriteConnection** (*Phalcon\Mvc\ModelInterface* \$model)

...

abstract public **isInitialized** (*mixed* \$modelName)

...

abstract public **getLastInitialized** ()

...

```
abstract public load (mixed $modelName, [mixed $newInstance])
...
abstract public addHasOne (Phalcon\Mvc\ModelInterface $model, mixed $fields, mixed $referencedModel, mixed
$referencedFields, [mixed $options])
...
abstract public addBelongsTo (Phalcon\Mvc\ModelInterface $model, mixed $fields, mixed $referencedModel, mixed
$referencedFields, [mixed $options])
...
abstract public addHasMany (Phalcon\Mvc\ModelInterface $model, mixed $fields, mixed $referencedModel, mixed
$referencedFields, [mixed $options])
...
abstract public existsBelongsTo (mixed $modelName, mixed $modelRelation)
...
abstract public existsHasMany (mixed $modelName, mixed $modelRelation)
...
abstract public existsHasOne (mixed $modelName, mixed $modelRelation)
...
abstract public getBelongsToRecords (mixed $method, mixed $modelName, mixed $modelRelation, Phal-
con\Mvc\ModelInterface $record, [mixed $parameters])
...
abstract public getHasManyRecords (mixed $method, mixed $modelName, mixed $modelRelation, Phal-
con\Mvc\ModelInterface $record, [mixed $parameters])
...
abstract public getHasOneRecords (mixed $method, mixed $modelName, mixed $modelRelation, Phal-
con\Mvc\ModelInterface $record, [mixed $parameters])
...
abstract public getBelongsTo (Phalcon\Mvc\ModelInterface $model)
...
abstract public getHasMany (Phalcon\Mvc\ModelInterface $model)
...
abstract public getHasOne (Phalcon\Mvc\ModelInterface $model)
...
abstract public getHasOneAndHasMany (Phalcon\Mvc\ModelInterface $model)
...
abstract public getRelations (mixed $modelName)
...
abstract public getRelationsBetween (mixed $first, mixed $second)
...
```



```
abstract public createQuery (mixed $sql)
...
abstract public executeQuery (mixed $sql, [mixed $placeholders])
...
abstract public createBuilder ([mixed $params])
...
abstract public addBehavior (Phalcon\Mvc\ModelInterface $model, Phalcon\Mvc\Model\BehaviorInterface $behavior)
...
abstract public notifyEvent (mixed $eventName, Phalcon\Mvc\ModelInterface $model)
...
abstract public missingMethod (Phalcon\Mvc\ModelInterface $model, mixed $eventName, mixed $data)
...
abstract public getLastQuery ()
...
abstract public getRelationByAlias (mixed $modelName, mixed $alias)
...
```

Interface **Phalcon\Mvc\Model\MessageInterface**

Methods

```
abstract public setType (mixed $type)
...
abstract public getType ()
...
abstract public setMessage (mixed $message)
...
abstract public getMessage ()
...
abstract public setField (mixed $field)
...
abstract public getField ()
...
abstract public __toString ()
...
abstract public static __set_state (array $message)
...
```

Interface Phalcon\Mvc\Model\MetaDataInterface

Methods

```
abstract public setStrategy (Phalcon\Mvc\Model\MetaData\StrategyInterface $strategy)
...
abstract public getStrategy ()
...
abstract public readMetaData (Phalcon\Mvc\ModelInterface $model)
...
abstract public readMetaDataIndex (Phalcon\Mvc\ModelInterface $model, mixed $index)
...
abstract public writeMetaDataIndex (Phalcon\Mvc\ModelInterface $model, mixed $index, mixed $data)
...
abstract public readColumnMap (Phalcon\Mvc\ModelInterface $model)
...
abstract public readColumnMapIndex (Phalcon\Mvc\ModelInterface $model, mixed $index)
...
abstract public getAttributes (Phalcon\Mvc\ModelInterface $model)
...
abstract public getPrimaryKeyAttributes (Phalcon\Mvc\ModelInterface $model)
...
abstract public getNonPrimaryKeyAttributes (Phalcon\Mvc\ModelInterface $model)
...
abstract public getNotNullAttributes (Phalcon\Mvc\ModelInterface $model)
...
abstract public getDataTypes (Phalcon\Mvc\ModelInterface $model)
...
abstract public getDataTypesNumeric (Phalcon\Mvc\ModelInterface $model)
...
abstract public getIdentityField (Phalcon\Mvc\ModelInterface $model)
...
abstract public getBindTypes (Phalcon\Mvc\ModelInterface $model)
...
abstract public getAutomaticCreateAttributes (Phalcon\Mvc\ModelInterface $model)
...
abstract public getAutomaticUpdateAttributes (Phalcon\Mvc\ModelInterface $model)
```

```
...
abstract public setAutomaticCreateAttributes (Phalcon\Mvc\ModelInterface $model, array $attributes)
...
abstract public setAutomaticUpdateAttributes (Phalcon\Mvc\ModelInterface $model, array $attributes)
...
abstract public setEmptyStringAttributes (Phalcon\Mvc\ModelInterface $model, array $attributes)
...
abstract public getEmptyStringAttributes (Phalcon\Mvc\ModelInterface $model)
...
abstract public getDefaultValues (Phalcon\Mvc\ModelInterface $model)
...
abstract public getColumnMap (Phalcon\Mvc\ModelInterface $model)
...
abstract public getReverseColumnMap (Phalcon\Mvc\ModelInterface $model)
...
abstract public hasAttribute (Phalcon\Mvc\ModelInterface $model, mixed $attribute)
...
abstract public isEmpty ()
...
abstract public reset ()
...
abstract public read (mixed $key)
...
abstract public write (mixed $key, mixed $data)
...
```

Interface *Phalcon\Mvc\Model\MetaData\StrategyInterface*

Methods

```
abstract public getMetaData (Phalcon\Mvc\ModelInterface $model, Phalcon\DiInterface $dependencyInjector)
...
abstract public getColumnMaps (Phalcon\Mvc\ModelInterface $model, Phalcon\DiInterface $dependencyInjector)
...
```

Interface Phalcon\Mvc\Model\QueryInterface

Methods

```
abstract public parse ()
...
abstract public cache (mixed $cacheOptions)
...
abstract public getCacheOptions ()
...
abstract public setUniqueRow (mixed $uniqueRow)
...
abstract public getUniqueRow ()
...
abstract public execute ([mixed $bindParam], [mixed $bindTypes])
...
```

Interface Phalcon\Mvc\Model\Query\BuilderInterface

Constants

```
string OPERATOR_OR
string OPERATOR_AND
```

Methods

```
abstract public columns (mixed $columns)
...
abstract public getColumns ()
...
abstract public from (mixed $models)
...
abstract public addFrom (mixed $model, [mixed $alias])
...
abstract public getFrom ()
...
abstract public join (mixed $model, [mixed $conditions], [mixed $alias])
...
abstract public innerJoin (mixed $model, [mixed $conditions], [mixed $alias])
```

```
...
abstract public leftJoin (mixed $model, [mixed $conditions], [mixed $alias])
...
abstract public rightJoin (mixed $model, [mixed $conditions], [mixed $alias])
...
abstract public getJoins ()
...
abstract public where (mixed $conditions, [mixed $bindParam], [mixed $bindTypes])
...
abstract public andWhere (mixed $conditions, [mixed $bindParam], [mixed $bindTypes])
...
abstract public orWhere (mixed $conditions, [mixed $bindParam], [mixed $bindTypes])
...
abstract public betweenWhere (mixed $expr, mixed $minimum, mixed $maximum, [mixed $operator])
...
abstract public notBetweenWhere (mixed $expr, mixed $minimum, mixed $maximum, [mixed $operator])
...
abstract public inWhere (mixed $expr, array $values, [mixed $operator])
...
abstract public notInWhere (mixed $expr, array $values, [mixed $operator])
...
abstract public getWhere ()
...
abstract public orderBy (mixed $orderBy)
...
abstract public getOrderBy ()
...
abstract public having (mixed $having)
...
abstract public getHaving ()
...
abstract public limit (mixed $limit, [mixed $offset])
...
abstract public getLimit ()
...
abstract public groupBy (mixed $group)
```

```
...
abstract public getGroupBy ()
...
abstract public getPhql ()
...
abstract public getQuery ()
...
```

Interface Phalcon\Mvc\Model\Query\StatusInterface

Methods

```
abstract public getModel ()
...
abstract public getMessages ()
...
abstract public success ()
...
```

Interface Phalcon\Mvc\Model\RelationInterface

Methods

```
abstract public setIntermediateRelation (mixed $intermediateFields, mixed $intermediateModel, mixed $intermediateReferencedFields)
...
abstract public isReusable ()
...
abstract public getType ()
...
abstract public getReferencedModel ()
...
abstract public getFields ()
...
abstract public getReferencedFields ()
...
abstract public getOptions ()
...
abstract public getOption (mixed $name)
```

...

abstract public **isForeignKey** ()

...

abstract public **getForeignKey** ()

...

abstract public **isThrough** ()

...

abstract public **getIntermediateFields** ()

...

abstract public **getIntermediateModel** ()

...

abstract public **getIntermediateReferencedFields** ()

...

Interface Phalcon\Mvc\Model\ResultInterface

Methods

abstract public **setDirtyState** (*mixed* \$dirtyState)

...

Interface Phalcon\Mvc\Model\ResultsetInterface

Methods

abstract public **getType** ()

...

abstract public **getFirst** ()

...

abstract public **getLast** ()

...

abstract public **setIsFresh** (*mixed* \$isFresh)

...

abstract public **isFresh** ()

...

abstract public **getCache** ()

...

abstract public **toArray** ()

...

Interface Phalcon\Mvc\Model\TransactionInterface

Methods

```
abstract public setTransactionManager (Phalcon\Mvc\Model\TransactionManagerInterface $manager)
...
abstract public begin ()
...
abstract public commit ()
...
abstract public rollback ([mixed $rollbackMessage], [mixed $rollbackRecord])
...
abstract public getConnection ()
...
abstract public setIsNewTransaction (mixed $isNew)
...
abstract public setRollbackOnAbort (mixed $rollbackOnAbort)
...
abstract public isManaged ()
...
abstract public getMessages ()
...
abstract public isValid ()
...
abstract public setRollbackedRecord (Phalcon\Mvc\ModelInterface $record)
...
```

Interface Phalcon\Mvc\Model\Transaction\ManagerInterface

Methods

```
abstract public has ()
...
abstract public get ([mixed $autoBegin])
...
abstract public rollbackPendent ()
...
abstract public commit ()
```


...

abstract public **rollback** ([*mixed* \$collect])

...

abstract public **notifyRollback** (*Phalcon\Mvc\Model\TransactionInterface* \$transaction)

...

abstract public **notifyCommit** (*Phalcon\Mvc\Model\TransactionInterface* \$transaction)

...

abstract public **collectTransactions** ()

...

Interface *Phalcon\Mvc\Model\ValidatorInterface*

Methods

abstract public **getMessages** ()

...

abstract public **validate** (*Phalcon\Mvc\EntityInterface* \$record)

...

Interface *Phalcon\Mvc\ModuleDefinitionInterface*

Methods

abstract public **registerAutoloaders** ([*Phalcon\DiInterface* \$dependencyInjector])

...

abstract public **registerServices** (*Phalcon\DiInterface* \$dependencyInjector)

...

Interface *Phalcon\Mvc\RouterInterface*

Methods

abstract public **setDefaultModule** (*mixed* \$moduleName)

...

abstract public **setDefaultController** (*mixed* \$controllerName)

...

abstract public **setDefaultAction** (*mixed* \$actionName)

...

abstract public **setDefaults** (array \$defaults)

...

```
abstract public handle ([mixed $uri])
...
abstract public add (mixed $pattern, [mixed $paths], [mixed $httpMethods])
...
abstract public addGet (mixed $pattern, [mixed $paths])
...
abstract public addPost (mixed $pattern, [mixed $paths])
...
abstract public addPut (mixed $pattern, [mixed $paths])
...
abstract public addPatch (mixed $pattern, [mixed $paths])
...
abstract public addDelete (mixed $pattern, [mixed $paths])
...
abstract public addOptions (mixed $pattern, [mixed $paths])
...
abstract public addHead (mixed $pattern, [mixed $paths])
...
abstract public addPurge (mixed $pattern, [mixed $paths])
...
abstract public addTrace (mixed $pattern, [mixed $paths])
...
abstract public addConnect (mixed $pattern, [mixed $paths])
...
abstract public mount (Phalcon\Mvc\Router\GroupInterface $group)
...
abstract public clear ()
...
abstract public getModuleName ()
...
abstract public getNamespaceName ()
...
abstract public getControllerName ()
...
abstract public getActionName ()
...
```

```
abstract public getParams ()
...
abstract public getMatchedRoute ()
...
abstract public getMatches ()
...
abstract public wasMatched ()
...
abstract public getRoutes ()
...
abstract public getRouteById (mixed $id)
...
abstract public getRouteByName (mixed $name)
...
```

Interface Phalcon\Mvc\Router\GroupInterface

Methods

```
abstract public setHostname (mixed $hostname)
...
abstract public getHostname ()
...
abstract public setPrefix (mixed $prefix)
...
abstract public getPrefix ()
...
abstract public beforeMatch (mixed $beforeMatch)
...
abstract public getBeforeMatch ()
...
abstract public setPaths (mixed $paths)
...
abstract public getPaths ()
...
abstract public getRoutes ()
...
```

abstract public **add** (*mixed* \$pattern, [*mixed* \$paths], [*mixed* \$httpMethods])

...

abstract public **addGet** (*mixed* \$pattern, [*mixed* \$paths])

...

abstract public **addPost** (*mixed* \$pattern, [*mixed* \$paths])

...

abstract public **addPut** (*mixed* \$pattern, [*mixed* \$paths])

...

abstract public **addPatch** (*mixed* \$pattern, [*mixed* \$paths])

...

abstract public **addDelete** (*mixed* \$pattern, [*mixed* \$paths])

...

abstract public **addOptions** (*mixed* \$pattern, [*mixed* \$paths])

...

abstract public **addHead** (*mixed* \$pattern, [*mixed* \$paths])

...

abstract public **clear** ()

...

Interface Phalcon\Mvc\Router\RouteInterface

Methods

abstract public **setHostname** (*mixed* \$hostname)

...

abstract public **getHostname** ()

...

abstract public **compilePattern** (*mixed* \$pattern)

...

abstract public **via** (*mixed* \$httpMethods)

...

abstract public **reConfigure** (*mixed* \$pattern, [*mixed* \$paths])

...

abstract public **getName** ()

...

abstract public **setName** (*mixed* \$name)

...

abstract public **setHttpMethods** (*mixed* \$httpMethods)

...

abstract public **getRouteId** ()

...

abstract public **getPattern** ()

...

abstract public **getCompiledPattern** ()

...

abstract public **getPaths** ()

...

abstract public **getReversedPaths** ()

...

abstract public **getHttpMethods** ()

...

abstract public static **reset** ()

...

Interface Phalcon\Mvc\UrlInterface

Methods

abstract public **setBaseUri** (*mixed* \$baseUri)

...

abstract public **getBaseUri** ()

...

abstract public **setBasePath** (*mixed* \$basePath)

...

abstract public **getBasePath** ()

...

abstract public **get** ([*mixed* \$uri], [*mixed* \$args], [*mixed* \$local])

...

abstract public **path** ([*mixed* \$path])

...

Interface Phalcon\Mvc\ViewBaseInterface

Methods

abstract public **setViewsDir** (*mixed* \$viewsDir)
...
abstract public **getViewsDir** ()
...
abstract public **setParamToView** (*mixed* \$key, *mixed* \$value)
...
abstract public **setVar** (*mixed* \$key, *mixed* \$value)
...
abstract public **getParamsToView** ()
...
abstract public **getCache** ()
...
abstract public **cache** ([*mixed* \$options])
...
abstract public **setContent** (*mixed* \$content)
...
abstract public **getContent** ()
...
abstract public **partial** (*mixed* \$partialPath, [*mixed* \$params])
...

Interface Phalcon\Mvc\ViewInterface

implements [Phalcon\Mvc\ViewBaseInterface](#)

Methods

abstract public **setLayoutsDir** (*mixed* \$layoutsDir)
...
abstract public **getLayoutsDir** ()
...
abstract public **setPartialsDir** (*mixed* \$partialsDir)
...
abstract public **getPartialsDir** ()
...

```
abstract public setBasePath (mixed $basePath)
...
abstract public getBasePath ()
...
abstract public setRenderLevel (mixed $level)
...
abstract public setMainView (mixed $viewPath)
...
abstract public getMainView ()
...
abstract public setLayout (mixed $layout)
...
abstract public getLayout ()
...
abstract public setTemplateBefore (mixed $templateBefore)
...
abstract public cleanTemplateBefore ()
...
abstract public setTemplateAfter (mixed $templateAfter)
...
abstract public cleanTemplateAfter ()
...
abstract public getControllerName ()
...
abstract public getActionName ()
...
abstract public getParams ()
...
abstract public start ()
...
abstract public registerEngines (array $engines)
...
abstract public render (mixed $controllerName, mixed $actionName, [mixed $params])
...
abstract public pick (mixed $renderView)
...
```

```
abstract public finish ()
...
abstract public getActiveRenderPath ()
...
abstract public disable ()
...
abstract public enable ()
...
abstract public reset ()
...
abstract public isDisabled ()
...
abstract public setViewsDir (mixed $viewsDir) inherited from Phalcon\Mvc\ViewBaseInterface
...
abstract public getViewsDir () inherited from Phalcon\Mvc\ViewBaseInterface
...
abstract public setParamToView (mixed $key, mixed $value) inherited from Phalcon\Mvc\ViewBaseInterface
...
abstract public setVar (mixed $key, mixed $value) inherited from Phalcon\Mvc\ViewBaseInterface
...
abstract public getParamsToView () inherited from Phalcon\Mvc\ViewBaseInterface
...
abstract public getCache () inherited from Phalcon\Mvc\ViewBaseInterface
...
abstract public cache ([mixed $options]) inherited from Phalcon\Mvc\ViewBaseInterface
...
abstract public setContent (mixed $content) inherited from Phalcon\Mvc\ViewBaseInterface
...
abstract public getContent () inherited from Phalcon\Mvc\ViewBaseInterface
...
abstract public partial (mixed $partialPath, [mixed $params]) inherited from Phalcon\Mvc\ViewBaseInterface
...
```


Interface Phalcon\Mvc\View\EngineInterface

Methods

abstract public **getContent** ()

...

abstract public **partial** (*mixed* \$partialPath, [*mixed* \$params])

...

abstract public **render** (*mixed* \$path, *mixed* \$params, [*mixed* \$mustClean])

...

Interface Phalcon\Paginator\AdapterInterface

Methods

abstract public **setCurrentPage** (*mixed* \$page)

...

abstract public **getPaginate** ()

...

abstract public **setLimit** (*mixed* \$limit)

...

abstract public **getLimit** ()

...

Interface Phalcon\Session\AdapterInterface

Methods

abstract public **start** ()

...

abstract public **setOptions** (*array* \$options)

...

abstract public **getOptions** ()

...

abstract public **get** (*mixed* \$index, [*mixed* \$defaultValue])

...

abstract public **set** (*mixed* \$index, *mixed* \$value)

...

abstract public **has** (*mixed* \$index)

...

```
abstract public remove (mixed $index)
...
abstract public getId ()
...
abstract public isStarted ()
...
abstract public destroy ([mixed $removeData])
...
abstract public regenerateId ([mixed $deleteOldSession])
...
abstract public setName (mixed $name)
...
abstract public getName ()
...
```

Interface Phalcon\Session\BagInterface

Methods

```
abstract public initialize ()
...
abstract public destroy ()
...
abstract public set (mixed $property, mixed $value)
...
abstract public get (mixed $property, [mixed $defaultValue])
...
abstract public has (mixed $property)
...
abstract public __set (mixed $property, mixed $value)
...
abstract public __get (mixed $property)
...
abstract public __isset (mixed $property)
...
```

Interface Phalcon\Translate\AdapterInterface

Methods

abstract public **t** (*mixed* \$translateKey, [*mixed* \$placeholders])

...

abstract public **query** (*mixed* \$index, [*mixed* \$placeholders])

...

abstract public **exists** (*mixed* \$index)

...

Interface Phalcon\Translate\InterpolatorInterface

Methods

abstract public **replacePlaceholders** (*mixed* \$translation, [*mixed* \$placeholders])

...

Interface Phalcon\ValidationInterface

Methods

abstract public **validate** ([*mixed* \$data], [*mixed* \$entity])

...

abstract public **add** (*mixed* \$field, *Phalcon\Validation\ValidatorInterface* \$validator)

...

abstract public **rule** (*mixed* \$field, *Phalcon\Validation\ValidatorInterface* \$validator)

...

abstract public **rules** (*mixed* \$field, array \$validators)

...

abstract public **setFilters** (*mixed* \$field, *mixed* \$filters)

...

abstract public **getFilters** ([*mixed* \$field])

...

abstract public **getValidators** ()

...

abstract public **getEntity** ()

...

abstract public **setDefaultMessages** ([array \$messages])

...

```
abstract public getDefaultMessage (mixed $type)
...
abstract public getMessages ()
...
abstract public setLabels (array $labels)
...
abstract public getLabel (mixed $field)
...
abstract public appendMessage (Phalcon\Validation\MessageInterface $message)
...
abstract public bind (mixed $entity, mixed $data)
...
abstract public getValue (mixed $field)
...
```

Interface **Phalcon\Validation\MessageInterface**

Methods

```
abstract public setType (mixed $type)
...
abstract public getType ()
...
abstract public setMessage (mixed $message)
...
abstract public getMessage ()
...
abstract public setField (mixed $field)
...
abstract public getField ()
...
abstract public __toString ()
...
abstract public static __set_state (array $message)
...
```

Interface Phalcon\Validation\ValidatorInterface

Methods

abstract public **hasOption** (*mixed* \$key)

...

abstract public **getOption** (*mixed* \$key, [*mixed* \$defaultValue])

...

abstract public **validate** (*Phalcon\Validation* \$validation, *mixed* \$attribute)

...

2.6 Legal

2.6.1 License

Phalcon is brought to you by the Phalcon Team! [[Twitter](#) - [Google Plus](#) - [Github](#)]

The Phalcon PHP Framework is released under the [new BSD license](#). Except where otherwise noted, content on this site is licensed under the [Creative Commons Attribution 3.0 License](#).

If you love Phalcon please return something to the community! :)

CHAPTER 3

Previous Versions

- 2.0.0 <<https://docs.phalconphp.com/fr/2.0.0/>>

CHAPTER 4

Autres formats

- PDF
- HTML in one Zip
- ePub