A Gentle Introduction to the Zend Framework

This tutorial provides an introduction to the Zend Framework. It assumes readers have experience in writing simple PHP scripts that provide web-access to a database.

1. First a digression – Smarty templates: the separation of code (model and control) and display (view)
2. Using simple Zend components in isolation
3. Zend_DB – help with data persistence
4. Model View Control and related issues
5. Zend's MVC framework

The examples have been built for an Ubuntu system with a local Apache server using NetBeans 6.9.1 and Zend 1.10. The NetBeans projects are available for download. Details of how to set up the environment are appended to this document along with links to tutorials at NetBeans and Oracle. Some examples use a MySQL server hosted on the same machine as the Apache server; others use a remote Oracle server accessed via Oracle's instant_client and its associated libraries.

1 Smarty

The first PHP program that you wrote probably achieved the supposed ideal of just a little code embedded in a HTML page, such as this variant of Ramus Lerdorf's example script for handling input from a form for entering a user's name and age:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>First PHP Program</title>
    </head>
    <body>
        <h1 align="center">Greetings from PHP</h1>
        <?php
        if(!empty($_POST['username'])) :
        ?>
        <p>Hello <?php echo $_POST['username'] ?>, you are <?php echo $_POST['age'] ?>
            years old.</p>
        <?php else : ?>
        <p>Well hello; maybe next time you could fill in the form.</p>
        <?php endif; ?>
    </body>
</html>
```

But when you started to create real PHP applications you were using large files with lots of PHP code containing output statements that generated the HTML, as illustrated in the following fragment of a script dealing with file uploads:

```
    }
}

function add_picture() {
    global $mysqli;
    $error = $_FILES["image"]["error"];
    if($error != UPLOAD_ERR_OK) {
        echo <<<NOLUCK
    <html>
        <head><title>File upload failed</title></head>
        <body>
            <h1>Upload failed</h1>
            <p>The upload attempt failed.  The error code was $error .</p>
        </body>
    </html>
NOLUCK;
        exit;
    }
    $filename = $_FILES['image']['tmp_name'];
    $title = $_POST["title"];
    $comment = $_POST["comment"];
    if(empty($title) || empty($comment)) {
        badinput(" need title and comment");
        exit();
    }
    // Accept letters, digits, whitespace, and some punctuation - including
    // both single and double quotes and dollars signs
    $pat = '/^[a-zA-Z0-9\s"\.;,:!?()]+$/';
    if(!preg_match($pat,$title)) {
        badinput("invalid data");
        exit;
    }
    if(!preg_match($pat,$comment)) {
        badinput("invalid data");
        exit;
    }
    $numbytes = filesize($filename);
```

Code for data processing just does not mix well with display. They really are separate issues.

Similar problems had arisen with other technologies, such as the Java web technologies. The

original Java servlets (~1997) were all code with data display handled via println statements – and as such they were not popular. It was difficult to envisage how pages would appear given just a collection of println statements. Java Server Pages (JSPs) were invented (~1999) to try to overcome the problems. JSPs (which are "compiled" into servlet classes) were an attempt to achieve the ideal of a little code embedded in a HTML document. The code in JSPs was initially as scriptlets – little fragments of embedded Java – later, with the Java Standard Tag Library (JSTL), XML style markup tags were used to define the code. (Although implemented quite differently, JSTL tags act a bit like macros that expand out to large chunks of Java code when the JSP is compiled to yield a servlet.)

But the problems remain. Little scraps of code (scriptlet or JSTL tag style) embedded in HTML work fine for "hello world" demonstrations, but with real applications it's more like little scraps of HTML embedded in lots of code.

Eventually, Java developers settled on a division of responsibility. Java code in servlets and application defined classes was used to select the data to be displayed; JSPs with scriptlet coding or preferably JSTL tags were used to display these data. The servlet that had been written by the developer ran first. All the complex coding is done in the servlets and helper classes. The servlet code builds data structures to hold the data that are to be output and has control code that determines which JSP display script is to run (there could be JSP scripts for successful outcomes and scripts that deal with various error reports). The servlet forwarded the structures with data to the selected JSP (actually, forwarded the data to another servlet that had been compiled from the JSP). Any code in the JSP would be very simple - "*display this data element here in this HTML div*", "*loop through this data collection outputting successive rows of a list or table*".

A similar division of responsibility can be achieved in PHP applications with PHP scripts and classes used to select the data that are then displayed using Smarty classes working with template files. Taking a slightly simplified view, the PHP script will create a hash-map of (name, value) pairs inside a "smarty" object and then invoke the display method of that object using a provided template.

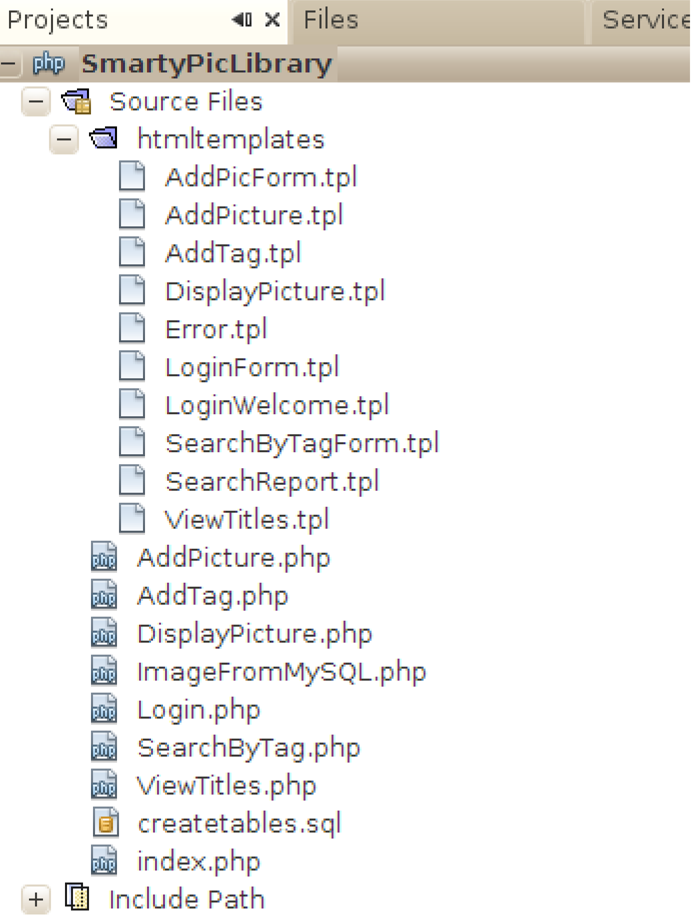
A Smarty template file consists of HTML markup and static content along with some limited scripting (in its own scripting language) that deals with tasks like "*display this data element here in this HTML div*", "*loop through this data collection outputting successive rows of a list or table*". Smarty template files are "compiled" into PHP classes (just like JSPs being compiled into servlets).

The main PHP script, and associated class files, are not encumbered with any print statements or blocks of predefined HTML. The Smarty template file is a pretty much standard HTML file that can be edited by a specialised HTML editor (provided the editor is configured to ignore the Smarty code fragments). PHP code is for control and data access; HTML markup and Smarty script fragments handle display. View is separated from control and model.

1.1 Smarty example – SmartyPicLibrary

The example application is available as the SmartyPicLibrary project in the download zip file. A conventional PHP version is described in the CSCI110 lecture notes; this new version uses Smarty for the form pages and response pages.

The application allows an owner to create a picture gallery, uploading pictures to a MySQL database. Pictures in the collection have title and comment information, and also "tags" held in a separate table. Uploading of pictures is restricted to the owner; other visitors can view the titles of pictures, search for pictures by tag, view pictures and add tags. The application is comprised of a number of PHP scripts; most handle "GET" requests by displaying a form, with input from that form being returned in "POST" requests for processing.



PHP scripts have two points of interaction with the Smarty system. There is an initial setup step where a "Smarty" object is configured; it needs a place to put the code that it will generate from the template HTML files:

```
<?php

require('/usr/local/lib/php/Smarty/Smarty.class.php');

// Global variables
$smarty = new Smarty();
$mysqli = 0;
$script = $_SERVER["PHP_SELF"];

function smartysetup() {
    global $smarty;
    $smarty->template_dir = '/home/nabg/SmartyStuff/Demo1/templates';
// The cache and templates_c directories need to be writeable
// by www-data (i.e. the Apache server process)
    $smarty->compile_dir = '/home/nabg/SmartyStuff/Demo1/templates_c';
    $smarty->cache_dir = '/home/nabg/SmartyStuff/Demo1/cache';
    $smarty->config_dir = '/home/nabg/SmartyStuff/Demo1/configs';
}
```

It is best to use a separate set of directories for each application that uses Smarty; these should be located separately from the htdocs directories. Obviously, the example code in the download files will need to be changed to reference a directory that you create on your own system. The cache and templates_c directories are for the code that the Smarty template engine creates and should be writeable by the Apache process. The other two directories are for more advanced uses, such as extending the set of templates that the Smarty engine employs.

```
nabg@info-bxsrr1s:~/SmartyStuff$ ls -l *
Demo1:
total 16
drwxr-xrwx 2 nabg nabg 4096 2011-04-13 09:15 cache
drwxr-xr-x 2 nabg nabg 4096 2011-04-13 09:15 configs
drwxr-xr-x 2 nabg nabg 4096 2011-04-13 09:15 templates
drwxr-xrwx 2 nabg nabg 4096 2011-04-13 09:15 templates_c

Demo2:
total 16
drwxr-xrwx 2 nabg nabg 4096 2011-04-13 09:16 cache
drwxr-xr-x 2 nabg nabg 4096 2011-04-13 09:16 configs
drwxr-xr-x 2 nabg nabg 4096 2011-04-13 09:16 templates
drwxr-xrwx 2 nabg nabg 4096 2011-04-13 09:16 templates_c
nabg@info-bxsrr1s:~/SmartyStuff$ pwd
/home/nabg/SmartyStuff
```

The other point of interaction between your PHP script and Smarty is where you forward the data that are to be displayed and start the Smarty display process.

The example Picture Library application has a script that allows a user to enter a tag, and that then uses this tag to retrieve the titles and identifiers that have been characterised with that tag. The results page from this search should be a table of links that will allow retrieval and display of the actual pictures. The PHP script runs the search request and assembles the resulting data for display in a Smarty template. These interactions are illustrated in the following code fragments:

```php
function display_search_form() {
    global $script;
    global $smarty;
    $smarty->assign('script', $script);
    $smarty->display('./htmltemplates/SearchByTagForm.tpl');
}

function dosearch() {
    global $mysqli;
    global $smarty;

    $usertag = $_POST["searchtag"];

    $stmt = $mysqli->Prepare("SELECT ident,title FROM nabg.Picys "
          . "where ident in (select Picid from nabg.PicyTags where Tagstr=?)");

    $stmt->bind_param('s', $usertag);
    $stmt->execute();
    $stmt->bind_result($id, $title);

    $matches = array();
    while ($stmt->fetch()) {
        $matches[] = array($id, $title);
    }

    $mysqli->close();
    $smarty->assign('matches', $matches);
    $smarty->assign('usertag', $usertag);
    $smarty->display('./htmltemplates/SearchReport.tpl');
}
```

```
smartysetup();

$method = $_SERVER["REQUEST_METHOD"];
if ($method == "POST") {
    connectToDatabase();
    dosearch();
} else {
    display_search_form();
}
```

The PHP script uses smarty->assign() to create name/value pairs in a hash map in the Smarty object, and smarty->display() which results in output using a class based on the template file supplied as an argument.

Smarty templates have their own scripting language with conditional constructs, loops, mechanisms for printing simple variables, array elements, or data members of an object. The template that displays the data on pictures with a given tag is:

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>My picture library</title>
    </head>
    <body>
        <h1>Pictures with tag {$usertag}</h1>
{if count($matches) gt 0}
        <table border="1">
            <thead>
                <tr>
                    <th>Title</th>
                </tr>
            </thead>
            <tbody>
            {foreach item=match from=$matches}
                <tr>
                    <td>
         <a href="./DisplayPicture.php?id={$match[0]}">{$match[1]}</a>
                   </td>
            </tr>
            {/foreach}
        </table>
{else}
        <p>There are no pictures currently tagged with {$usertag}.</p>
{/if}
   </body>
   </html>
```

The Smarty template has to print the tag that was used for the search; this was passed via the request $smarty->assign('usertag', $usertag) in the main script (names for variables used by Smarty don't have to be the same as the PHP script variables). In the template, the simple annotation {$usertag} will result in output of the assigned value.

The user could have requested a search specifying a tag that has not yet been employed with this picture library; in such a case, the collection of matches will be of zero length. This is dealt with in the template code using a Smarty {if *condition*} … {else} … {/if} construct. If there were no matching pictures, the response page will simply state this fact.

If there were some pictures in the library that had been tagged with the user's chosen tag, the search

will have resulted in a collection of two element arrays each containing a picture identifier, and its title. In this case, there will be elements in the $matches Smarty variable and so the "foreach loop" will be executed. In this loop, {foreach *item in collection*} … {/foreach}, the rows of a table are output to produce a response page like:

Pictures with tag Scotland

Title
Woods near Aviemore
Castle Stalker

Smarty supports mechanisms for modifying data that are to be printed. If you will be echoing data originally entered by users then you face the possibility of attempts at cross-site scripting attacks where a malicious user has entered Javascript etc. Smarty makes light work of this problem. You simply request that all strings output to the final response page have any HTML significant characters escaped – that's a one liner : $smarty->default_modifiers = array('escape:"html"').

1.2 The Smarty advantage

You can learn more about Smarty at the Smarty site. The PHP 5 Unleashed site has a detailed section on Smarty.

Learning how to use Smarty, and thereby separate code and display, is a worthwhile step toward the use of a more complete Model-View-Control system such as Zend. For many of your simpler sites, there would be no need to adopt the full complexities of an MVC system; but all such sites could be improved using Smarty rather than having simply PHP scripts with embedded HTML output.

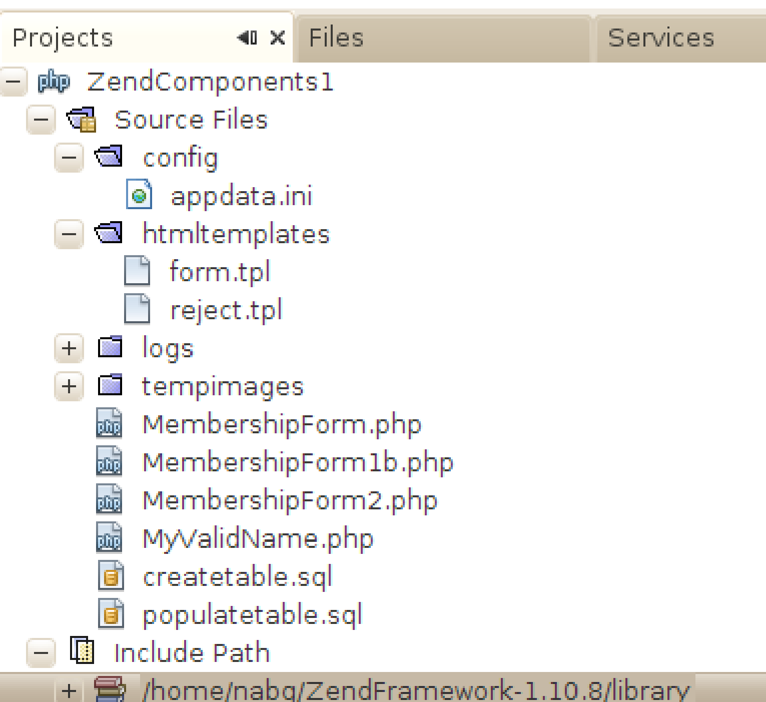
Smarty templates are used for display in the following examples that illustrate the use of Zend components. They can be used with the full Zend MVC framework as substitutes for Zend's own classes. Complicated response pages are better handled using a combination of Zend's Layout and View classes.

2 Using simple Zend components in isolation

The Zend Framework is not a monolithic entity. There are some core classes that serve as the basis of its “model-view-control” structure; but there are many other classes in the Zend Framework libraries that can be utilised quite independently of the MVC parts. The MVC framework is moderately complex and has many programming conventions and specific ways of organising the files that make up an application; these conventions are supported through a host of specialised shell scripts for creating stub files etc. The Zend Framework is powerful – but intimidating to beginners.

It is easier to start by just adopting some of the useful Zend classes and gradually learning the Zend programming styles.

2.1 ZendComponents1 – MembershipForm: Mostly data validation!



The “PictureLibrary” site presented in the “Smarty” section above obviously needs to be up-graded if it is to prove useful. Rather than have a single owner who can post pictures, one would want a system that had various classes of membership – an owner who can decide who else joins, members who can create galleries, visitors with the right to comment on galleries, and casual visitors who can simply view pictures. There also has to be a way of applying for membership – a form with fields for email, chosen member name, etc. If our new site proves popular, it will unfortunately attract spammers who will try to use scripts to create fake accounts for spam entry. A reasonably sophisticated application form, with some form of CAPTCHA device, is needed; all inputs should be validated as far as is practical e.g. is that email address for real.

Apply for Membership

Personal details

Your name	
Your postcode (ZIP code)	
Your country	Afghanistan
Year of birth	
Gender	Male
	Female

Your membership

Preferred user name (A single word made up from letters and digits, at least 6, not more than 10 characters.	
Your email address	

CAPTCHA

(A CAPTCHA is a device intended to prevent membership applications by scripted 'robots'. Robots are used to spam web-sites with junk advertis

This recognition test is supposed to make scripted applications less easy. You must enter the text string that is shown

Your guess	

Action

Apply for membership

This form will be returned in response to a GET request to the MembershipForm.php script; the completed form will be POSTed back to the script and will result in a final response page either rejecting the application or promising eventual consideration of the application by the site's owner.

The first notable feature is a CAPTCHA – the common version with a mutilated text string. The entire CAPTCHA scheme is handled by an instance of a Zend framework class as illustrated in the following script. (The image based CAPTCHA system requires that your PHP installation includes the GD graphics module.)

```
<?php

require('/usr/local/lib/php/Smarty/Smarty.class.php');
require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance();

// Global variables
$smarty = new Smarty();
$captcha = 0;

function display_member_form() {
    global $captcha;
    global $smarty;
    $id = $captcha->generate();
    $script = $_SERVER['PHP_SELF'];
    $smarty->assign('script', $script);
    $smarty->assign('id', $id);
    $smarty->display('./htmltemplates/form.tpl');
}

function rejectCandidate($msg1, $msg2) { … }
```

```
function handle_member_application() { … }

function smartysetup() { … }

smartysetup();
//REMEMBER TO MAKE THE ./tempimages DIRECTORY WRITEABLE BY WEB-SERVER!
$captcha = new Zend_Captcha_Image(array(
     'timeOut' => 200,
     'wordLen' => 6,
     'font' => '/usr/share/fonts/truetype/ubuntu-font-family/Ubuntu-R.ttf',
     'imgDir' => './tempimages/'
 ));
$method = $_SERVER["REQUEST_METHOD"];
if ($method == "POST") {
    handle_member_application();
} else {
    display_member_form();
}
?>
```

You should configure your NetBeans environment so that the Zend Framework libraries are added to the default PHP library path (I left the library in the directory where I had unpacked the original download). Any script using Zend library classes should then start with the lines:

```
require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance();
```

Subsequently, all references to specific classes, such as Zend_Captcha_Image will be sorted out automatically; one doesn't need to list all the requires clauses.

The application needs an instance of the Zend_Captcha_Image class; this has to be configured with various details such as the TrueType font that is to be used and the name of a directory where a temporary image file can be created; I have used a “tempimages” subdirectory I created in the Netbeans project. (The Zend_Captcha_Image code deals automatically with flushing old images from the directory; of course, the image directory needs to be writeable by the Apache process that is executing the Zend script.)

The code of Zend_Captcha_Image deals with session initialization and will set a cookie that will be returned with the initial form page. It stores details of the CAPTCHA that it generated in session state. An identifier is generated along with the CAPTCHA image; this identifier is used in the composition of the form page:

```
   <html><head><title>Membership Application</title></head>
    <body><h1>Apply for Membership</h1>

      <form method='POST' action='{$script}' >
      <input type='hidden' name='id' value={$id} />
       <fieldset>
       <legend>Personal details</legend>
      …
       </fieldset>
       <fieldset>
       <legend>CAPTCHA</legend>
       <p><font size='-1'>(A CAPTCHA is a device intended to prevent membership
applications by scripted 'robots'.  Robots are used to spam web-sites with junk
advertising and malicious components.)</font></p>
       <p>This recognition test is supposed to make scripted applications less
easy.  You must enter the text string that is shown in distorted form.</p>
       <table border='1'>
       <tr><td colspan='2' align='center'>
```

```
<img src='./tempimages/{$id}.png'></td></tr>
        <tr><th>Your guess</th><td><input type='text' name='userguess' 
/></td></tr>
        </table>
        </fieldset>
        <fieldset>
        <legend>Action</legend>
        <input type='submit' value='Apply for membership' />
        </fieldset>
    </form>
```

The code checking submitted data should start with the CAPTCHA verification:

```
function handle_member_application() {
    global $captcha;
    $input = $_POST['userguess'];
    $id = $_POST['id'];
    $guess = array();
    $guess['input'] = $input;
    $guess['id'] = $id;

    if (!$captcha->isValid($guess)) {
        rejectCandidate("Incorrect CAPTCHA text",
"Are you myopic, dumb, or are you a 'bot'?  We don't want your kind as members."
        );
        exit;
    }
    …
```

The documentation on the Zend_Captcha_Image class isn't that clear. But it does have a method isValid() that will check a user's guess against the data that was saved in $_SESSION. This isValid function takes the user's guess as a two element array argument – the string representing the guess and the identifier number for the generated image. If the guess is invalid, the user's membership application should be rejected and the script can terminate.

If the CAPTCHA was successfully guessed, the other inputs should be validated. There will often be data that are most easily validated by simple PHP code:

```
function handle_member_application() {
// Check CAPTCHA
    ...
// gender - don't need Zend for this; it's either Male or Female or some hacker
// playing the fool and composing a request by hand
    $gender = $_POST['gender'];
    if (!($gender == 'Male' || $gender = 'Female')) {
        rejectCandidate("Invalid gender",
     "Trying to hack in with hand crafted submission?" .
     "We don't want your kind as members.");
        exit;
    }
    …
```

But the Zend framework libraries include a large number of specialised validation classes that can help. One of these is an email validator. At a minimum, this will check the character pattern for given for the email to verify that it is a typical user-name@hostname style email address. However, it can be configured to do far more. It can verify that the hostname exists and that it has a mail reception point! This doesn't guarantee that the email is valid (the user-name could still be false) but it does cut down out many spurious addresses invented by would be spammers:

```
function handle_member_application() {
// Check CAPTCHA and gender inputs
    ...
// email - don't simply check that it looks like an email, try to contact the
// specified server to see if it does handle email (doesn't check that actual
// email identity exists)
    $emailSupplied = $_POST['email'];

    $emailValidator = new Zend_Validate_EmailAddress(
       array(
           'allow' => Zend_Validate_Hostname::ALLOW_DNS,
           'mx' => true
            ));
    if (!$emailValidator->isValid($emailSupplied)) {
        rejectCandidate("Invalid email",
"Trying to hack in with a made up email?  We don't want your kind as members.");
        exit;
    }
```

Other validator classes are less sophisticated, and it may seem like overkill to use them. But use of these validators does contribute a certain consistency to the code – and that probably makes their use worthwhile.

The form has some more inputs to validate. There is a year of birth; it would seem reasonable to expect this to be in the range 1910...2000; we don't want little kids or centenarians joining this members only web site. The Zend_Validate_Between class allows such tests to be made. The applicant for membership also had to pick a user-name for use with the site; this was supposed to be made up of a string of 6...10 alphanumerics. It's possible to built up a composite validator, a validator chain, by combining simple validators like a string length checker etc:

```
function handle_member_application() {
// Check CAPTCHA, gender, and email inputs
      ...
// Year of birth, numeric, >=1910, <=2000
    $yearValidator = new Zend_Validate_Between(
            array('min' => 1910, 'max' => 2000));
    $birthYear = $_POST['year'];
    if (!$yearValidator->isValid($birthYear)) {
        $reason = "The data entered for year of birth were rejected because ";
        foreach ($yearValidator->getMessages() as $message) {
            $reason = $reason . $message;
        }
        rejectCandidate("Invalid year of birth",
                $reason);
        exit;
    }

// Username for use as identifier in members only areas - alphanum 6-10
characters
    $uname = $_POST['uname'];
    $validatorChain1 = new Zend_Validate();
    $validatorChain1->addValidator(
         new Zend_Validate_StringLength(array('min' => 6,'max' => 10)))
            ->addValidator(new Zend_Validate_Alnum());
    if (!$validatorChain1->isValid($uname)) {
        $reason = "Your chosen user name is invalid because ";
        foreach ($validatorChain1->getMessages() as $message) {
            $reason = $reason . $message;
        }
```

```
        rejectCandidate("Unacceptable user name", $reason);
        exit;
    }
```

Another of Zend's more exotic validators is a “Post Code” checker (OK, it doesn't quite work with UK post codes – it will get fixed one day). This can serve as another filter to keep out spammers. The form asks for a country and a post code and the Zend supplied checking code then verifies whether these are at least mutually consistent. The form has a <select> with a set of all countries known to the Zend framework, along with their country codes; there is also a string field for the postcode.

```
10     <tr><th>Your postcode (ZIP code)</th>
        <td><input type='text' size='20' maxlength='20' name='pcode' /></td>
      </tr>
 11     <tr><th>Your country</th>
 12             <td>
 13     <select name='country' size='1'>
 14 <option value='AF'>Afghanistan</option>
 15 <option value='AL'>Albania</option>
 16 <option value='DZ'>Algeria</option>
...
...
274 <option value='YE'>Yemen</option>
275 <option value='ZM'>Zambia</option>
276 <option value='ZW'>Zimbabwe</option>
277 </select>
```

The script checks consistency:

```
function handle_member_application() {
// Check CAPTCHA, gender, email, year of birth and user name inputs
...
// Now check Postcode by country
    $country = $_POST['country'];
    $pcode = $_POST['pcode'];
    $locale = new Zend_Locale($country);
    $pcodeValidator = new Zend_Validate_PostCode($locale);
    if (!$pcodeValidator->isValid($pcode)) {
        $reason = "Post code appears invalid because ";
        foreach ($pcodeValidator->getMessages() as $message) {
            $reason = $reason . $message;
        }
        rejectCandidate("Dubious post code", $reason);
        exit;
    }
```

This test on post codes completes the checking for this first version of the membership application. The checks will not keep out really determined hackers and spammers – but then one probably cannot make such guarantees. You don't have to be (can't be) hacker-proof; you just need to be sufficiently harder to hack than many other sites of similar value – the hackers will pick on them instead.

2.2 ZendComponents1 – MembershipForm1b: More validation!

This section illustrates a couple more validation steps, simple data base access, and use of email.

The Zend Validator class is extendible – if there isn't a supplied validator class that does what you want, you can define your own. Applicants for this membership based site have to supply a full

name. Of course it is impossible to truly validate a name – there is too much variation. But one can filter out obvious rubbish names invented by spammers. This requires a bit more than a check for alphabetic chharacters; names with hyphens, single apostrophes etc are all valid.   Why not define a validator that will require a name more or less in the format name initials family-name?

A validator class should extend Zend_Validate_Abstract. It has to implement an isValid() method; this should build up a record of errors as they are encountered. (A validator may stop after finding an error, or may search for all errors; this example stops at the first error.) In addition to trying to check the user's name, this code creates a tidied up version – removing things like spans of multiple whitespace characters etc.

```
<?php
class MyValid_Name extends Zend_Validate_Abstract {
    // Cannot use Zend's alpha validator (with spaces) because can get
    // names like "Dennis H. Smith" (so "." allowed), "Fergus O'Brien" (so
    // single quote allowed), or "Katherine Lloyd-Davis" (so hyphen allowed)
    const MSG_LENGTH = 'msgLength';
    const MSG_ILLEGAL = 'msgIllegal';
    const MSG_CHARPAT = 'msgCharpat';
    const MIN_NAME = 4;
    const MAX_NAME = 60;

    protected $_messageTemplates = array(
        self::MSG_LENGTH => "That name does not have an acceptable length" .
             " (at least 4 and at most 60 characters)",
        self::MSG_ILLEGAL => "That name contains impermissible characters'",
        self::MSG_CHARPAT => "That name doesn't resemble typical name pattern"
    );

    public function isValid($value) {
        $value = trim($value);
        // Change any tabs into spaces
        preg_replace("/\t/", " ", $value);
        // Compress sequences of spaces into single space
        preg_replace("/  */"," ",$value);

        $this->_setValue($value);

        $len = strlen($value);
        if ($len < self::MIN_NAME || $len > self::MAX_NAME) {
            $this->_error(self::MSG_LENGTH);
            return false;
        }

        // Reject any name with dubious characters - no B1ll Gat3$ etc
        $allowed = "/^[A-Za-z \.\-']+$/";
        if (!preg_match($allowed, $value)) {
            $this->_error(self::MSG_ILLEGAL);
            return false;
        }
        // Think it reasonable to limit things to at most 1 of both hypen and
        // single quote characters; tough on Katherine-Anne Lloyd-Davis, or
        // O'Malley O'Toole; they just can't use their full names
        $countquote = substr_count($value, "'");
        $counthypen = substr_count($value, "-");
        if (($countquote > 1) || ($counthypen > 1)) {
            $this->_error(self::MSG_ILLEGAL);
            return false;
        }

        // What about applicants giving 'names' like Ooooooh, Sexxxxy, Fizzzzz -
```

```
        // don't want persons like them becoming members!
        // No names with four or more successive vowels or four or more
        // successive consonants
        $lotsofvowels = "/[aeiou]{4}/i";
        $lotsofconsonants = "/[bcdfghjklmnpqrstvwxyz]{4}/i";
        if(preg_match($lotsofvowels,$value) ||
                  preg_match($lotsofconsonants, $value)) {
            $this->_error(self::MSG_CHARPAT);
            return false;
        }

        // A few more checks - want a surname
        // so looking for something like
        //          Smi... capital followed by lower case
        //    of    O'Br... capital single quote capital lower case
        // (MacTavish etc should get through as not matching entire string;
        // it will match on the Mac bit and ignore the rest; Lloyd-Davis
        // should match the Lloyd and ignore rest)
        $pat1 = "/([A-Z][a-z]+) | ([A-Z]'[A-Z][a-z]+)/";
        if (!preg_match($pat1, $value)) {
            $this->_error(self::MSG_CHARPAT);
            return false;
        }

        // Applicant must supply first name optional initials family name
        // or initials other name(s)
        $pat2 = "/^[A-Z][a-z]+ [A-Z\. ]*[A-Z]'?[A-Za-z]+/";
        $pat3 = "/^[A-Z]\. .* [A-Z][A-Za-z\-']*$/";
        if(!(preg_match($pat2, $value) || preg_match($pat3, $value))) {
            $this->_error(self::MSG_CHARPAT);
            return false;
        }
        return true;
    }

    public function getName() {
        return $this->value;
    }

}


?>
```

The checking script can be extended to exploit our new validator class:

```
<?php

require('/usr/local/lib/php/Smarty/Smarty.class.php');
require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance();

require('MyValidName.php');
…

function handle_member_application() {
// Check CAPTCHA, gender, email, year of birth, user name, and post code inputs
…

// Full name of applicant
    $namevalidator = new MyValid_Name();
```

```
    if (!$namevalidator->isValid($_POST["username"])) {

        $modname = htmlspecialchars($namevalidator->getName());
        $reason = $modname . "was not valid because ";
        foreach ($namevalidator->getMessages() as $message) {
            $reason = $reason . $message;
        }

        rejectCandidate("Problem with 'full name' as entered", $reason);
        exit;
    }
    $modname = $namevalidator->getName();
…
```

The name that the user picks to identify himself/herself within the group should be unique. These names will be kept in a data base table along with other data. The script handling a membership application should verify that the chosen name has not already been claimed. The script could easily be extended with a fragment of code to submit an SQL query to check the chosen name against the table. But why write code yourself when it can be written automatically?

The Zend framework library has a validator class just for this purpose - Zend_Validate_Db_RecordExists. The PHP script needs to create a connection to the database (instance of another Zend class!) and then use this to submit a simple query via the specialised Zend validator. (The main Zend DB related classes are considered in more detail in the next section.)

The database used for this example is MySQL (the project contains SQL scripts that define the tables and add some test data). My PHP installation has the PDO_MySQL components installed and these provide the most convenient connection.

```
function handle_member_application() {
// Check CAPTCHA, gender, email, year of birth, user name, post code, and full name inputs
…

    $db = new Zend_Db_Adapter_Pdo_Mysql(array(
                'host' => 'localhost',
                'username' => 'homer', // Change as appropriate
                'password' => 'doh',
                'dbname' => 'homer'
            ));

    $dbvalidator = new Zend_Validate_Db_RecordExists(
                    array(
                        'table' => 'gallerymembers',
                        'field' => 'username',
                        'adapter' => $db
                    )
    );

    if ($dbvalidator->isValid($uname)) {
        rejectCandidate("Chosen user name",
           "Unfortunately, another user already has taken that username;" .
            "please pick something else.");
        exit;
    }
```

If the user's membership application passes all these tests then it will have to go to the site owner for final review. The data relating to the application will need to be saved and the site owner notified. The response to the user will acknowledge the application and promise an eventual

decision. The code needed would involve running an SQL insert statement and composing an email message to the owner. But once again, why write code yourself when most is available in packaged form.

Zend's basic data base connection classes, such as Zend_Db_Adapter_Pdo_Mysql, provide an “insert” method that will insert a new row. The data are provided in a key => value array; the Zend system creates and runs the appropriate SQL insert statement, making use of data base features such as default values for uninitialised columns and auto-increment identifier columns (it gets details of these from table meta-data). So the data insertion is simplified to the following code:

```
function handle_member_application() {
// Check all inputs as far as practical, now save the applicant's data
…
      $insertdata = array(
        'username' => $uname,
        'fullname' => $modname,
        'postcode' => $pcode,
        'countrycode' => $country,
        'gender' => $gender,
        'email' => $emailSupplied,
        'yearofbirth' => $birthYear
    );

    $db->insert("gallerymembers", $insertdata);
```

The class Zend_Mail_Transport_Smtp is just a simple wrapper around PHP's mailing functions; but it might as well be used for consistency throughout the code:

```
function handle_member_application() {
// Check the inputs, save data that look plausible, now notify owner
…

    $emailcontent = $modname . " applied for membership on " .
              date(DateTime::RSS);
    $tr = new Zend_Mail_Transport_Smtp('smtp.sapiens.com');
    Zend_Mail::setDefaultTransport($tr);
    $mail = new Zend_Mail();
    $mail->setFrom('homer@sapiens.com', 'Homer'); // change as appropriate
    $mail->addTo('homer@sapiens.com', 'Homer');
    $mail->setSubject("Zend demo application");
    $mail->setBodyText($emailcontent);
    $mail->send();
```

2.3 ZendComponents1 – MembershipForm2: Configuration

The code just shown has innumerable parameters, such as email addresses and database passwords, built in. In addition, there are text strings used for output in response pages; text strings used in a development version will likely need to be changed in a production version.

It is never satisfactory to have such data built in to the code. Changes to data values necessitate editing code with risks of introduction of errors, or failure to completely update values that may be needed in more than one place. It is better to load such data from a separate more readily edited configuration file. The Zend libraries provide excellent support for such separate configuration (configuration files are required when using the full Zend MVC framework as they hold configuration data for the framework itself – they can also hold application specific data, or such data can be held separately).

The example code shown above can be improved by exploiting Zend's Zend_Confi_Ini class.

Another improvement might be to log hacking attempts. If your logs reveal a hacker trying to access your site, and being caught on validation checks, you might want to add a suitable clause in the style "Deny from *130.141.152.163*" to your web-server configuration.

This section illustrates the use of the Config_Ini class and Zend's Zend_Log class.

The Zend Config classes support a variety of formats for configuration data files, including XML files, but the most commonly used are probably ".ini" files. Ini files are text files where you can define data elements and their values. The data elements can be hierarchic – data-object, sub-object, field. The overall .ini file can have multiple sections – again hierarchic in nature. The file can start with a section defining the default values for data elements, and then have subsections that contain overriding values and additional data-elements that are relevant to particular configurations of an application.

In this example, we need an .ini file with data such as the following:

```
Application data -
;   location of log files, details of mysql database,
;     and information for email messaging of owner
[standard]
logfilelocation              = ./logs/hackers.txt
messages.badguess            = "Are you myopic, ... members."
...
messages.uname1              = "Your chosen user name is invalid because "
messages.pcode               = "Post code appears invalid because "
database.params.host         = localhost
database.params.username     = homer
database.params.password     = doh
database.params.dbname       = homer
database.params.table        = gallerymembers
database.params.field        = username
mail.smtpserver              = smtp.sapiens.com
mail.from.id                 = homer@sapiens.com
mail.from.name               = Homer
mail.to.id                   = homer@sapiens.com
mail.to.name                 = Homer
mail.subject                 = Zend demo application
```

Such a file can be read in using a Zend_Config_Ini object; the data become properties of that objects – so it will have a logfilelocation property with a string value, a messages property which is an object with several properties each having a string value etc. Where values are needed in the PHP script they can be taken from these properties.

The Zend_Log and Zend_Log_Writer_Stream classes work together to make it easy to log issues encountered by the PHP script – issues such as possible hacker attacks.

The following code fragments illustrate some of the changes to the code given earlier, the new code utilises the configuration file and keeps logs of issues:

```
<?php

require('/usr/local/lib/php/Smarty/Smarty.class.php');
require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance();


require('MyValidName.php');

// Global variables
$config = new Zend_Config_Ini('./config/appdata.ini',
                              'standard');
```

```
$writer = new Zend_Log_Writer_Stream($config->logfilelocation);
$logger = new Zend_Log($writer);
$smarty = new Smarty();
$captcha = 0;

function logAHacker($msg) {
    global $logger;
    $ip = $_SERVER["REMOTE_ADDR"];

    $logentry = $ip . "\t" . $msg;
    $logger->info($logentry);
}
…
function handle_member_application() {
    global $config;
    …
      if (!$captcha->isValid($guess)) {
        logAHacker("Guessing captcha");
        rejectCandidate("Incorrect CAPTCHA text",
                $config->messages->badguess);
        exit;
    }
    …
    $db = new Zend_Db_Adapter_Pdo_Mysql(array(
                'host' => $config->database->params->host,
                'username' => $config->database->params->username,
                'password' => $config->database->params->password,
                'dbname' =>$config->database->params->dbname
            ));
    …
```

2.4 Exploiting Zend Components

There are other component libraries that you could use with your PHP scripts. These Zend components are as good as any others. Most PHP scripts could be improved through judicious use of things like separate configuration files and Zend_Config_Ini, logging, standardised code for data validation etc. Such usage does not make that great an improvement in your applications but it represents an easy step toward adopting more sophisticated parts of the framework.

3 Zend_DB

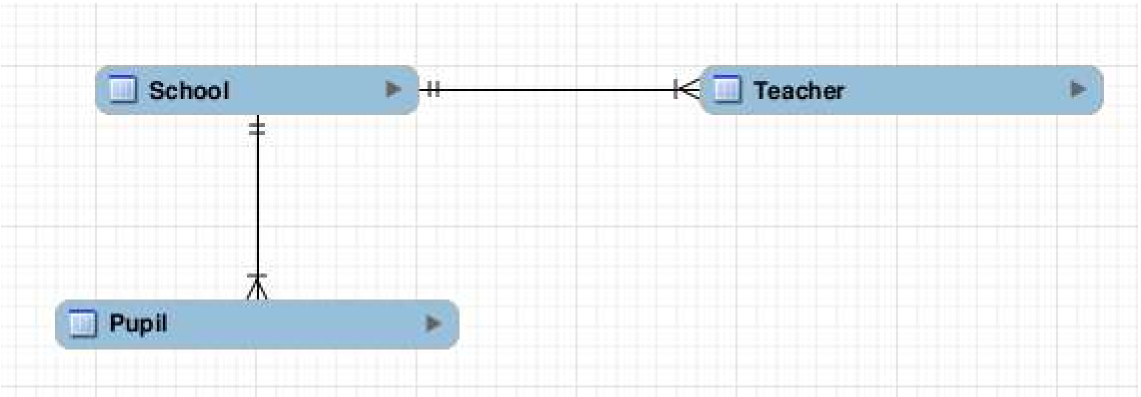
There are really only two web applications:

1 Request data
  Read and validate input data, reject request if data invalid.
  Bind data values to parameters of some SQL select statement
  Run SQL request
  Retrieve desired data from relational tables
  Format and return response page

2 Contribute data
  Read and validate input data, reject request if data invalid
  Bind data to parameters in a SQL insert or update statement
  Run SQL request
  Format response acknowledging data contribution

Overwhelmingly, web-applications are concerned with getting data out of, or putting data into relational tables. So inevitably, there is an awful lot of tiresome code involving SQL as strings, statement objects, database handles etc. Anything that helps with such data access makes life easier for developers and probably improves the general standard of code.

Zend_DB is a group of classes that facilitate working with databases. Some of the Zend_DB classes are illustrated in this section. These examples require a slightly more elaborate database with related tables. The tables used are the same as those in my earlier tutorial on Java Persistence Architecture; they are School, Teacher, and Pupil.



As I already had populated tables in an Oracle database, I used these. My PHP installation didn't have the PDO_Oracle module, so I use a direct OCI connection with the Zend classes. (Internally, Oracle capitalises all column names; it seems that the OCI connection requires similar capitalisation of column names in requests.)

The imaginary application is intended for use by a school board that must administer a number of primary schools. The application allows administrators to get listings of teachers employed at particular schools, view pupil enrolments, add teachers, transfer teachers etc.

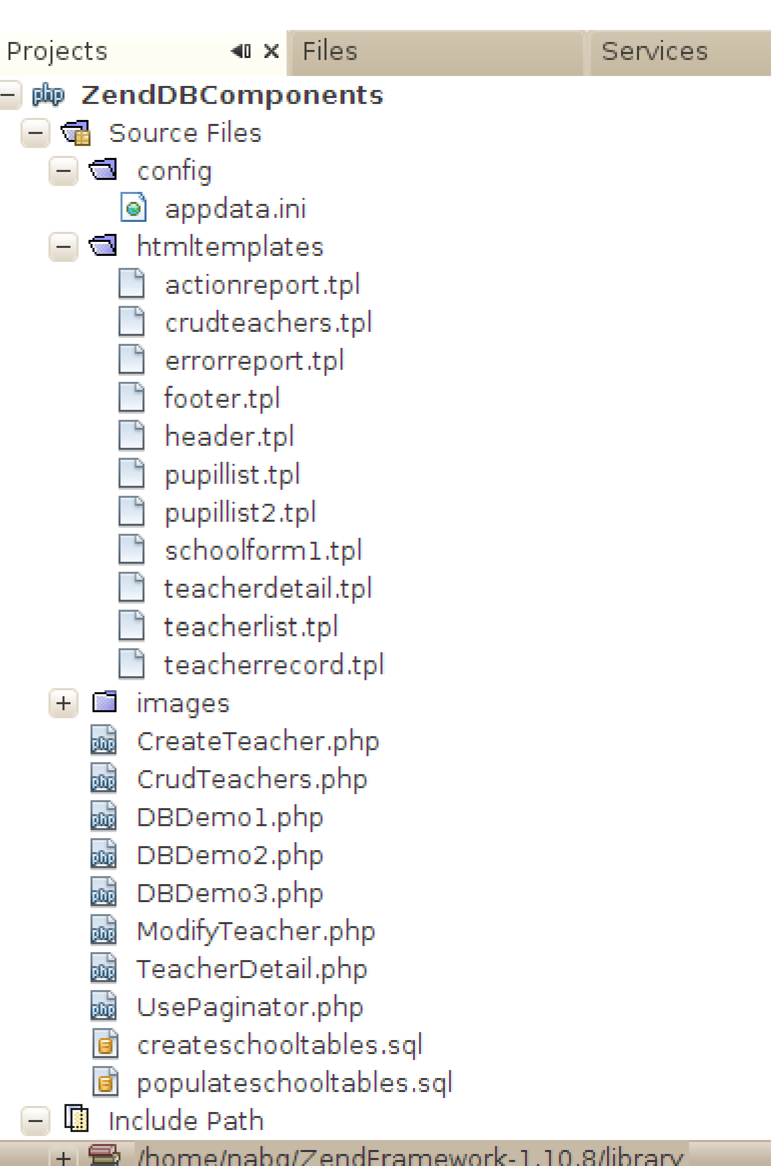
The first group of examples illustrate a number of classes including Zend_Db_Adapter, Zend_DB_Statement, Zend_DB_Select, and Zend_Paginator. Although taking advantage of the Zend classes to simplify database access, the PHP scripts are still bound up with composing a request, getting a result set as an array of rows etc. The second group of examples use Zend_Table and related classes. These provide higher level access – essentially a rather simple object-relational mapping system that allows the PHP script to work with entity objects.

3.1 ZendDBComponents: Zend_DB_Adapter and related classes

The scripts in this project allow an administrator to view lists of teachers and pupils, view details of a selected teacher, and create and modify data on teachers.

The appdata.ini file contains the configuration data for the connection to an Oracle database; these data are loaded using a Zend_Config_Ini object.

```
;Application data -
;
[standard]
logfilelocation              = ./logs/hackers.txt
database.adapter             = Oracle
database.params.username     = nabg
database.params.password     = notreallymypassword
database.params.dbname       = wraith:1521/csci
```

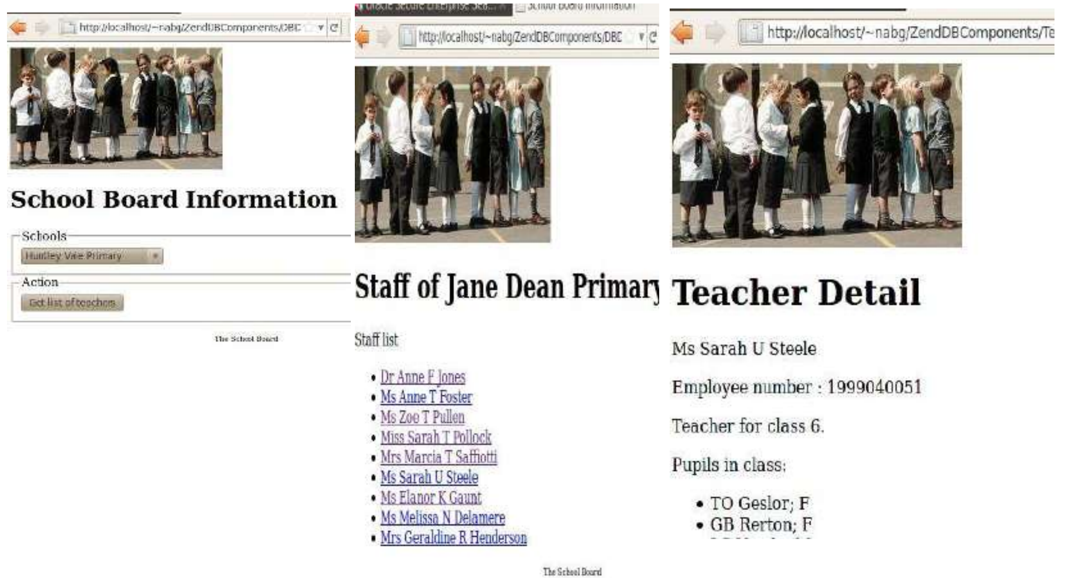


Smarty templates continue to be used for page rendering. Smarty support “include” files that make it easier to build a consistent set of web pages that have common elements like header and footer sections:

```
<html><head><title>School Board Information</title></head>
    <body>
{include file="htmltemplates/header.tpl"}
<h1>School Board Information</h1>
      <form method={$method} action='{$script}' >
       <fieldset>
       <legend>Schools</legend>
        <select name='schoolname' size='1'>
          {foreach item=school from=$schools}
                <option>
                    {$school['SCHOOLNAME']}
                </option>
            {/foreach}
       </select>
       </fieldset>
       <fieldset>
       <legend>Action</legend>
       <input type='submit' value='Get list of {$option}' />
       </fieldset>
       </form>
{include file="htmltemplates/footer.tpl"}
      </body></html>
```

3.1.1 DBDemo1.php and TeacherDetail.php : Simple selection using Zend_DB_Adapter

The scripts DBDemo1.php and TeacherDetail.php illustrate the most naïve use of Zend_DB_Adapter. This use is much like conventional programming with JDBC for Java, or database handles and SQL strings in Perl or PHP.



The code of DBDemo1.php, shown below, shows the easiest way of getting a database adapter (at least the easiest for scripts not built using the entire MVC framework). The Zend_DB class has a method, factory, that takes an object (loaded from the .ini file using Zend_Config_Ini) that has properties defining the adapter (PDO_MySQL, PDO_Oracle, Oracle, etc), and data such as the database name, user-name, and password. The code here gets an adapter set up in its dbadaptersetup() function.

```
<?php

require('/usr/local/lib/php/Smarty/Smarty.class.php');
require_once 'Zend/Loader/Autoloader.php';
```

```php
Zend_Loader_Autoloader::getInstance();

// Global variables
$config = new Zend_Config_Ini('./config/appdata.ini',
                'standard');
$smarty = new Smarty();
$db = NULL;

function smartysetup() { … }

function dbadaptersetup() {
    global $config;
    global $db;
    $db = Zend_Db::factory($config->database);
}

function display_request_form() {
    global $db;
    global $smarty;
    $sql = 'select schoolname from school';
    $result = $db->fetchAll($sql);

    $smarty->assign('script', $_SERVER['PHP_SELF']);
    $smarty->assign('schools', $result);
    $smarty->assign('option', 'teachers');
    $smarty->assign('method', 'POST');
    $smarty->display('./htmltemplates/schoolform1.tpl');
}

function handle_information_request() {
    global $db;
    global $smarty;
    $chosen = $_POST['schoolname'];
    $sql = 'select * from Teacher where schoolname=:name';
    $result = $db->fetchAll($sql, array('name' => $chosen));

    $smarty->assign('school', $chosen);
    $smarty->assign('teachers', $result);
    $smarty->display('./htmltemplates/teacherlist.tpl');
}


smartysetup();
dbadaptersetup();
$method = $_SERVER["REQUEST_METHOD"];
if ($method == "POST") {
    handle_information_request();
} else {
    display_request_form();
}
?>
```

SQL select requests can be composed as strings - 'select * from Teacher where schoolname=:name' – and then run using the fetchAll() (or fetchRow()) methods. Details depend a bit on the actual adapter. With the adapter for MySQL, one would use "?" place-holders for parameters and a simple array of values that bind to parameters. Here, with an Oracle adapter, named parameters must be employed.

As shown in the code above, the DBDemo1 script handles a GET request by retrieving an array of names of schools that are used to populate a <select></select> in the data entry form. When the

user chooses a school and POSTs back the form, another select query is run; this returns an array or row arrays with data for the teachers that can then be displayed as list items using a Smarty template. Each entry in the generated HTML list acts as a link to the TeacherDetails.php script:

```
<p>Staff list</p>
<ul>
{foreach item=teacher from=$teachers}
   <li>
      <a href="./TeacherDetail.php?id={$teacher['EMPLOYEENUMBER']}">
         {$teacher['TITLE']} {$teacher['FIRSTNAME']} 
         {$teacher['INITIALS']} {$teacher['SURNAME']}
      </a>
   </li>
{/foreach}
</ul>
```

The code in the TeacherDetail.php script runs a SQL select to retrieve data from the Teacher table and, if the teacher's role is a class teacher, runs another request to retrieve details from the Pupil table identifying pupils in the teacher's class. Once again, these select requests are conventional – create an SQL string, bind in parameter values, run a query, work through the result set:

```
function display_teacher_detail() {
    global $db;
    global $smarty;

   $teacherid =  $_GET['id'];

    $teacher = $db->fetchRow('SELECT * FROM TEACHER WHERE EMPLOYEENUMBER=:id',
        array('id' => $teacherid),Zend_Db::FETCH_OBJ);

    // Select pupils if a class teacher
    $pupils = array();
    $schoolclass = $teacher->ROLE;
    if(in_array($schoolclass, array('K', '1', '2', '3', '4', '5', '6'))){
        $pupils = $db->fetchAll(
           'SELECT * FROM PUPIL WHERE SCHOOLNAME=:school AND CLASSLVL=:cl',
           array('school' => $teacher->SCHOOLNAME,
           'cl' => $teacher->ROLE),
        Zend_Db::FETCH_OBJ);

    }
    $smarty->assign('teacher', $teacher);
    $smarty->assign('pupils', $pupils);
    $smarty->display('./htmltemplates/teacherdetail.tpl');
}
```

In this case, the mode for the adapter was changed so that instead of returning a row array (indexed by field names) for each row in the result set it would return "objects" (instances of PHP's std_class) with property values. It doesn't make much difference here, just a minor change in how the Smarty code will handle the data, but often such objects with properties are more convenient if the script has more complex processing to do. (The property names end up being capitalised because they are taken from the Oracle meta-data and Oracle capitalises column names.)

The Smarty code easily adapts to using properties:

```
<p>{$teacher->TITLE} {$teacher->FIRSTNAME} 
{$teacher->INITIALS} {$teacher->SURNAME}</p>
```

3.1.2 DBDemo2.php: Using Zend_DB_Select

The DBDemo2.php script illustrates use of the Zend_DB_Select class. This class is intended to simplify the creation of valid SQL select statements. One gets a DB_Select object and incrementally adds elements – elements like “where” conditions, “order-by” conditions etc.

The DBDemo2 script retrieves details of all pupils in a school:

Pupils at Huntley Vale Pr

Pupil list

- VN Bennet; F
- BN Bleamish; F
- SJ Carol; M
- CR Denby; M
- MS Destelle; M
- EA Destelle; F
- GH Eagham; M
- JK Eggers; M
- IP Egham; M
- JU Errost; M
- HJ Fales; F
- JH Fazof; M
- HG Fincross; M
- YPS Fitter; F
- C Fok; M
- EP Foz; F
- TO Fulk; F
- SA Gawley; M
- HG Gentri; F
- TR Gubby; M
- HE Habad; F
- D Hafna; M
- FD Harod; F
- JK Hartlee; M
- JH Heggle; M
- HG Henders; M
- P Henderson; F
- IK Holly; M
- WS Homer; F
- SD Host; F

```
function handle_information_request() {
    global $db;
    global $smarty;
    $chosen = $_POST['schoolname'];

    $select = $db->select();
    $select->from('PUPIL', array('INITIALS', 'SURNAME', 'GENDER'))->
            where('SCHOOLNAME=:school')->
                bind(array('school' => $chosen))->
                    order(array('SURNAME'));
    $stmt = $select->query();
    $result = $stmt->fetchAll();

    $smarty->assign('school', $chosen);
    $smarty->assign('pupils', $result);
    $smarty->display('./htmltemplates/pupillist.tpl');

}
```

3.1.3 DBDemo3.php and UsePaginator.php : Using Zend_Paginator

Report pages that have very long lists or tables and which require scrolling are not an attractive way of presenting data. Users generally prefer a system that paginates the data and allows them to move from page to page. It's hard work implementing a reporting system that has pagination – unless you have the option of using the Zend_Paginator when everything is easy.

Pupils at Huntley Vale Primai

Name	Initials	Gender
Habad	HE	F
Hafna	D	M
Harod	FD	F
Hartlee	JK	M
Heggle	JH	M
Henders	HG	M
Henderson	P	F
Holly	IK	M
Homer	WS	F
Host	SD	F

Previous 1 2 3 4 5 6 7 Next

The School Board

The Zend_Paginator has features that are designed to make it work well with Zend views and decorators, but it can be used independently of the main MVC framework creating data that are displayed using a Smarty template.

One obvious parameter for the paginator is the number of items per page. The paginator runs a “select count(*) ...” query to determine the total number of items, and from these data it determines the number of pages.

The paginator works with requests for a specific page number and a Zend_DB_Select object that you have created to select the data that you want. It augments your implied SQL with additional constraint clauses to limit the number of items retrieved and specify the starting offset for the first retrieved item. (It does not load all the table data into memory and work with an array – that would be too costly!)

The example code, in UsePaginator.php, using the paginator is:

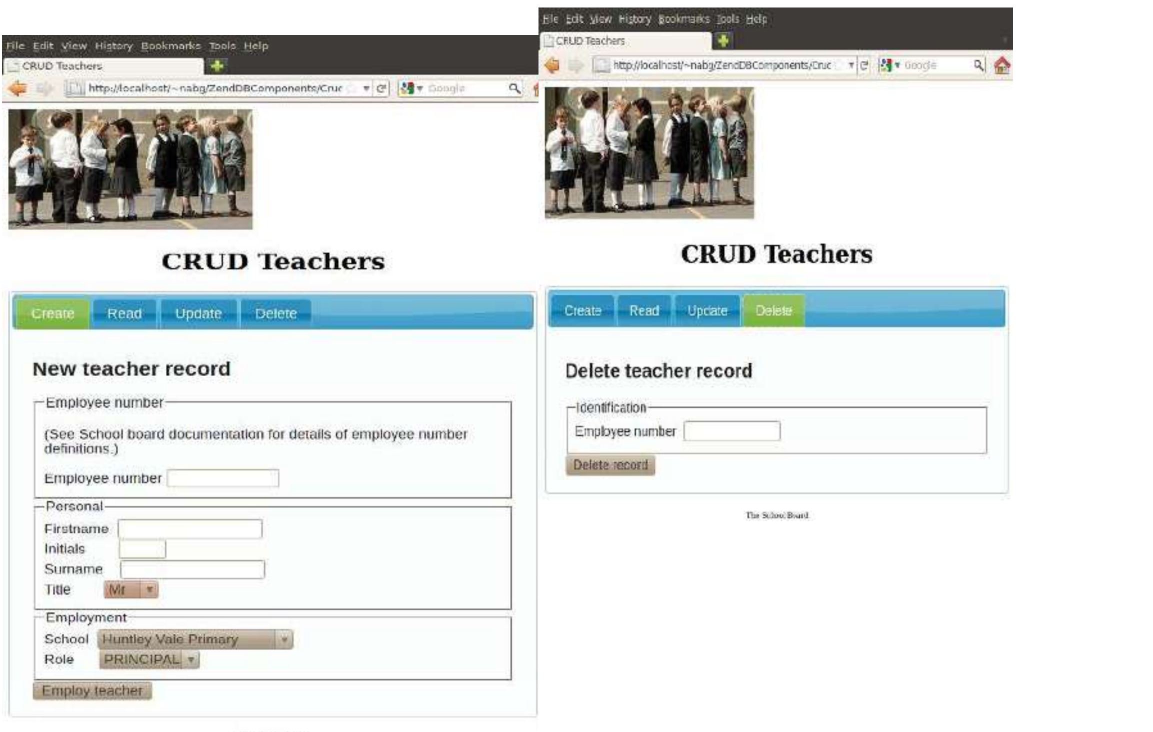
```
function handle_information_request() {
    global $db;
    global $smarty;
    $chosen = $_GET['schoolname'];
    $page = 1;
    if (isset($_GET['page']))
        $page = $_GET['page'];
    $db->setFetchMode(Zend_Db::FETCH_OBJ);
    $select = $db->select();
    $select->from('PUPIL', array('INITIALS', 'SURNAME', 'GENDER'))->
            where('SCHOOLNAME=:school')->
            bind(array('school' => $chosen))->
            order(array('SURNAME'));
    $paginator = Zend_Paginator::factory($select);


    $paginator->setCurrentPageNumber($page);
    $paginator->setItemCountPerPage(10);

    $thepages = $paginator->getPages();
    $pagerange = $thepages->pagesInRange;

    $items = $paginator->getCurrentItems();

    $pupilArray = array();
    foreach ($items as $pupil) {
        $pupilArray[] = $pupil;
    }


    $smarty->assign('script', $_SERVER['PHP_SELF']);
    $smarty->assign('first', 1);
    $smarty->assign('last', 1+ (int)($paginator->getTotalItemCount()/10));
    $smarty->assign('school', htmlspecialchars($chosen));
    $smarty->assign('pupils', $pupilArray);
    $smarty->assign('current', $page);
    $smarty->assign('pagelist', $pagerange);
    $smarty->display('./htmltemplates/pupillist2.tpl');
}
```

The array of 10 pupil objects for the current page is displayed via the following Smarty template:

```
<html><head><title>School Board Information</title></head>
<body>
{include file="htmltemplates/header.tpl"}
<h1>Pupils at {$school}</h1>

<table align='center' border='1'>
<tr><th>Name</th><th>Initials</th><th>Gender</th></tr>
{foreach item=pupil from=$pupils}
<tr>
    <td>{$pupil->SURNAME}</td><td>{$pupil->INITIALS}</td>
    <td>{$pupil->GENDER}</td>
</tr>
{/foreach}
</table>
<br>

<p align='center'>
{if $current gt $first}
<a href="{$script}?schoolname={$school}&amp;page={$current-1}">Previous</a>
{/if}
{foreach item=page from=$pagelist}
      
    <a href="{$script}?schoolname={$school}&amp;page={$page}">{$page}</a>
{/foreach}
{if $current lt $last}
<a href="{$script}?schoolname={$school}&amp;page={$current+1}">Next</a>
{/if}
</p>

{include file="htmltemplates/footer.tpl"}
</body></html>
```

3.1.4 CRUDTeachers.php, CreateTeacher.php, and ModifyTeacher.php : Create, Read, Update, Delete

The previous examples focussed on selection – what about insert, update, and delete?



The Zend_DB_Adapter class has methods like insert() and update() that simplify the coding of such operations.

The CRUDTeachers.php script retrieves the school names from the database and creates the initial multi-option form. (The form uses JQuery, and the tab-pane interface from Jquery-UI; all the Javascript being downloaded from Google.)

The insert operation is coded in the CreateTeacher.php script:

```
function handle_create() {
    global $db;
    global $smarty;
    $fname = $_POST['firstname'];
    $initials = $_POST['initials'];
    $surname = $_POST['surname'];
    $title = $_POST['title'];
    $schoolname = $_POST['schoolname'];
    $role = $_POST['rolename'];
    $empnum = $_POST['empnum'];
    // If employee number were allocated from an Oracle sequence, e.g.
    // a sequence name teacher_seq, one could get the next employee number by
    // $empnum = $db->nextSequenceId("teacher_seq");
    $data = array(
        'EMPLOYEENUMBER' => $empnum,
        'SURNAME' => $surname,
        'INITIALS' => $initials,
        'FIRSTNAME' => $fname,
        'TITLE' => $title,
        'SCHOOLNAME' => $schoolname,
        'ROLE' => $role
    );

    $rowcount = $db->insert("TEACHER", $data);

    if ($rowcount == 1)
        $msg = "New teacher record created";
    else
        $msg = "Failed to create record";
    $smarty->assign('msg', $msg);
    $smarty->assign('header','Create Teacher Record');

    $smarty->display('./htmltemplates/actionreport.tpl');
}
```

Record creation is easy! The data base adapter's insert method is invoked with arguments giving the table name and a name => value array with known values to be inserted into the specified columns of the new row. For the Teacher data table, the primary key is an employee number which is assigned as part of the recruitment process; it is not an auto-generated identifier. Different data base systems use varying means for automatically assigning identifiers; the Zend documentation explains the minor changes needed for specific data bases. Columns that do not have values assigned in the data table passed as an argument to the insert() function will have to take default values as defined for that table.

The “read, update, delete” options all take an employee number as input and make a GET request to the ModifyTeacher.php script. This loads the requested teacher record using the data base adapters fetchRow() method and displays the record via a Smarty template. For “read” and “delete” the displayed record is read-only; for “update”, appropriate fields may be edited. The update and delete displays have submit actions that POST requests for the actual operation to be performed.

Teacher Record

Teacher Record

Teacher Record

The operations are performed in the code in ModifyTeacher that handles the POST request.

Both are simple. The update operation is achieved using the update() method of the data base adapter; the arguments used in the call are the table name, the array of column-name => value elements for changed data, and a where clause to identify the record. Deletions are performed by invoking the delete() method with a suitable “where” clause.

```
function handle_command() {
    global $db;
    global $smarty;
    $command = $_POST['command'];
    $empnum = $_POST['empnum'];
    $empnum = $db->quote($empnum, 'INTEGER'); // beware of hackers

    //echo "$command on $empnum";
    if($command=="Update") {
        $data = array(
            "SURNAME" => $_POST["surname"],
            "FIRSTNAME" => $_POST["firstname"],
            "INITIALS" =>$_POST["initials"],
            "TITLE" => $_POST["title"],
            "ROLE" => $_POST["role"],
            "SCHOOLNAME" => $_POST["schoolname"]

        );
        $db->update("TEACHER", $data, "EMPLOYEENUMBER = $empnum");
        $head = "Update Teacher Record";
        $msg = "Teacher record updated";
    }
    else {
        $db->delete("TEACHER", "EMPLOYEENUMBER = $empnum");
        $head = "Delete Teacher Record";
        $msg = "Teacher record deleted";
    }

     $smarty->assign('msg', $msg);
     $smarty->assign('header',$head);

    $smarty->display('./htmltemplates/actionreport.tpl');

}
```

3.2 ZendDBTableExamples: A more “object-oriented” style

All the example scripts in the previous section suffer from the problem of SQL and database concepts intruding into the logic of the program. Rather than have something in the style “compose SQL query, run query, collect results and extract data” you might well prefer to have a program that

worked with School, Teacher, and Pupil entity objects and some helper intermediary class(es) that could be asked to "*load a Pupil entity*", "*get me a collection of all Teacher entities where their role is Principal*", "*save this new School entity*". Zend_DB_Table and related classes provide a means of creating code with this more object-oriented style.

3.2.1 DBTable1.php and NewSchool.php : extend Zend_Db_Table_Abstract

We can start simply with the creation of a class that will allow the PHP script to work with school entities corresponding to rows in the School table. A suitable class definition is:

```php
<?php

class SchoolTable1 extends Zend_Db_Table_Abstract {
    // Explicitly identify table, primary key etc
    // (In most cases, framework can resolve such things automatically through
    // a combination of naming conventions and its ability to examine
    // table meta-data.)
    protected $_name = 'SCHOOL';
    protected $_primary = 'SCHOOLNAME';
}
?>
```

That is all there is to it!

Class SchoolTable1 extends the library supplied Zend_Db_Table_Abstract class, and through this it is tied into all the automated persistence mechanisms. Code in the Zend library uses the $_name member to identify the table, and analyses the database meta data for that table. It was not really necessary to identify the primary key for the School table – Zend code would have sorted that out; but if you aren't following conventions on naming tables, fields etc it is best to explicitly declare such things.

There are no new methods and no new data members defined for SchoolTable1; it doesn't really need anything that isn't defined already in Zend_Db_Table_Abstract. Its base class has methods like fetchAll() that retrieve collections of school entity objects. Its role is to act as a "gateway" between the PHP script and the persistent table (see Martin Fowler's "Data Table Gateway" design pattern).

What will these school entity objects be? Really, they are instances of PHP's std_class with properties defined that correspond to the columns in the School data table. All of this is handled by the Zend library code.

Our new SchoolTable1 class is put to use in this script (DBTable1.php) that produces a page with a list of the names of all the schools:

```php
<!DOCTYPE HTML >
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Zend DB Table Exercises - exercise 1</title>
    </head>
    <body>
        <h1 align="center">The schools</h1>
<?php

require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance();

// Get the Zend auto-loader in first, as SchoolTable1 class is derived
// from a Zend_DB class
require('SchoolTable1.php');
```

```
// Global variables
$config = new Zend_Config_Ini('./config/appdata.ini', 'standard');

function dbadaptersetup() {
    global $config;
    $db = Zend_Db::factory($config->database);
    Zend_Db_Table::setDefaultAdapter($db);
}

dbadaptersetup();
$schoolinfo = new SchoolTable1();
$schools = $schoolinfo->fetchAll();
foreach($schools as $aschool) {
    print "<p>";
        print $aschool->SCHOOLNAME;
        print "<br>    ";
        print $aschool->ADDRESS;
    print "</p>";
}
?>
    </body>
</html>
```

The script code first creates a database adapter, using the configuration data in the application's .ini file. This is then set as the “default adapter”. When one creates an instance of a table gateway class one must provide it with access to the database. If only one database is in use it is simplest to define an adapter to this database as the “default adapter”, then it isn't necessary maintain and pass around adapter references.

The code creates its SchoolTable1 object; it can now use the table. A simple fetchAll() request to the table returns a collection of simple school entity objects. The response page is generated with data taken from the properties of these objects.

The NewSchool.php script displays a form that allows a user to define the name and address of an additional school to be run by the school board. Data posted from this form are used to create a new record in the persistent table:

```
function handle_newschool_request() {
    global $smarty;
    $name = $_POST['schoolname'];
    $address = $_POST['address'];
    // Obviously should validate the input - but laziness prevails
    $schoolTable = new SchoolTable1();
    $data = array('SCHOOLNAME' => $name, 'ADDRESS' => $address);
    $schoolTable->insert($data);
    $smarty->assign('header', 'School records');
    $smarty->assign('msg', 'New school record created');
    $smarty->display('./htmltemplates/actionreport.tpl');
}
```

3.2.2 DBTable2.php : Zend_DB_Table_Select

You don't often want all the data; so you need something better than just a plain fetchAll()! Actually, the fetchAll() function (and the related fetchRow()) function can take an instance of the Zend_DB_ Table_Select class. This class is a specialisation of the Zend_DB_Select class illustrated earlier. It again allows you to build up a selection request by adding qualifiers for “where” clauses, “order-by” clauses etc.

This usage is illustrated in the DBTable2.php example script. This handles a “GET” request by displaying a form that allows a user to select a school, and the corresponding “POST” request that results in the generation of a list of teachers employed at that school.

There code uses a TeacherTable class to access the Teacher table in the database; it is every bit as complex as the SchoolTable class. The TeacherTable class's fetchAll() method will return a collection of teacher entity objects that are once again just extensions of std_class with properties to match the columns in the persistent table.

```
class TeacherTable extends Zend_Db_Table_Abstract {

    protected $_name = 'TEACHER';
    protected $_primary = 'EMPLOYEENUMBER';

}
```

```
<?php

require('/usr/local/lib/php/Smarty/Smarty.class.php');
require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance();

require('SchoolTable1.php');
require('TeacherTable.php');

// Global variables
$config = new Zend_Config_Ini('./config/appdata.ini',
                'standard');
$smarty = new Smarty();

function display_request_form() {
    global $smarty;
    $schoolinfo = new SchoolTable1();
    $schools = $schoolinfo->fetchAll();
    $result = $schools->toArray();
    $smarty->assign('script', $_SERVER['PHP_SELF']);
    $smarty->assign('schools', $result);
    $smarty->assign('option', 'teachers');
    $smarty->assign('method', 'POST');
    $smarty->display('./htmltemplates/schoolform1b.tpl');
}

function handle_information_request() {
    global $smarty;
    $chosen = $_POST['schoolname'];
    $teacherTable = new TeacherTable();
    $selector = $teacherTable->select();
    $selector->where('SCHOOLNAME=:scl')->
            bind(array('scl' => $chosen))
                  ->order('SURNAME');

    $teachers = $teacherTable->fetchall($selector);
    $smarty->assign('school', $chosen);
    $smarty->assign('teachers', $teachers);
    $smarty->display('./htmltemplates/teacherlistb.tpl');
}

function smartysetup() { … }

function dbadaptersetup() { … }

smartysetup();
```

```
dbadaptersetup();
$method = $_SERVER["REQUEST_METHOD"];
if ($method == "POST") {
    handle_information_request();
} else {
    display_request_form();
}
?>
```

3.2.3 ChangeSchool.php, TeacherTable2.php, and TeacherClass.php : extending Zend_DB_Table_Row_Abstract

If all the script simply displays data elements, then the entity classes based just on std_class will suffice. If you want to manipulate the entity objects, and extend their functionality, then it something a little more elaborate is required.

The Zend_DB classes allow for the definition of an entity class that represents a row from the table. It is necessary to define the relationship between the table class and the row class; so this example uses a new more elaborate table class:

```
class TeacherTable2 extends Zend_Db_Table_Abstract {

    protected $_name = 'TEACHER';
    protected $_primary = 'EMPLOYEENUMBER';
    protected $_rowClass = 'TeacherClass';

}
```

and the related “row class”:

```
class TeacherClass extends Zend_Db_Table_Row_Abstract {
    //put your code here
    ...
}
```

The row class will have methods like save() and delete() defined. New functions can be added. For example, one might want a function that returned the full name of the teacher (rather than extract separately the elements for first name, initials, and surname). The function would be added to the row class:

```
class TeacherClass extends Zend_Db_Table_Row_Abstract {
   function FULLNAME() {
        $fullname = $this->FIRSTNAME . " " .
                $this->INITIALS . " " .
                $this->SURNAME;
        return $fullname;
    }

    ...

}
```

This definition would result in a possibly irritating discontinuity in the code using TeacherClass objects:

```
// Have a teacher class object $aTeacher
…
// Need surname
    $sname = $aTeacher->SURNAME;   // access as property
```

```
…
// Need full name
    $fullname = $aTeacher->FULLNAME();    // access via function
```

One programming idiom that you will often see used with these row classes is the re-definition of __get() (defined in std_class) to allow permit a consistent use of the property style interface.

```
class TeacherClass extends Zend_Db_Table_Row_Abstract {
   function FULLNAME() {
        $fullname = $this->FIRSTNAME . " " .
                $this->INITIALS . " " .
                $this->SURNAME;
        return $fullname;
    }

    function __get($key) {
        if (method_exists($this, $key)) {
            return $this->$key();
        }
        return parent::__get($key);
    }
}
```

The use of the row class (and modified table class) is illustrated in the ChangeSchool.php script. This script displays a form that allows a user to enter an employee number and select a school, and processes these data by re-allocating the employee to a new school. The entity object, instance of TeacherClass is loaded using the TeacherTable2 class; it is modified as an object; and saved in its modified form:

```
function handle_information_request() {
    global $smarty;
    $teacherid = $_POST['empnum'];
    $teacherTable = new TeacherTable2();
    $selector = $teacherTable->select();
    $selector->where('EMPLOYEENUMBER=:empnum')->
           bind(array('empnum' => $teacherid));

    $teacherObject = $teacherTable->fetchRow($selector);

    if(!$teacherObject) {
     $smarty->assign('msg',
        'Teacher record not found - invalid employee number');
     $smarty->display('./htmltemplates/errorreport.tpl');
     exit;
    }
    $sname = $_POST['schoolname'];
    $teacherObject->SCHOOLNAME= $sname;
    $teacherObject->save();
    $smarty->assign('header', 'Teacher reassigned');
    // Function call style
    // $fullname = $teacherObject->FULLNAME();
    // As quasi property
    $fullname = $teacherObject->FULLNAME;
    $smarty->assign('msg', $teacherid . ', ' .
          $fullname . ' now working at ' . $sname);
    $smarty->display('./htmltemplates/actionreport.tpl');
}
```

The FULLNAME for the teacher can be retrieved either by function call or, with __get() overriden, by property request.

The rationale for overriding __get() is to achieve this greater consistency in the code. But it's incomplete. One can assign to properties - $teacherObject->SCHOOLNAME = $name. It would be necessary to re-define __set(), and invent some interesting implentation function (splitting a full name into component parts), if one wanted the ability to assign values to the supposed SCHOOLNAME property. Usually it seems that only __get() is overridden, which really just shifts any inconsistencies in the code.

3.2.4 UseRelatedTables.php, SchoolTable2.php, and TeacherTable3.php: using related tables

The conceptual model for the data in this example has a "School" owning a collection of "Teacher" objects (and another collection of "Pupil" objects). But this conceptual model isn't apparent in any of the code shown so far.

However, the Zend_DB libraries provide all that is needed to rework the code so that one can have School objects that can be asked for the Teacher (or Pupil) collections when these are needed. Of course what happens is a bit of code behind the scenes runs a SQL query like "*select * from Teacher where schoolname=this->schoolname*" and assembles data from the result set into the required collection. Code that exists behind the scenes is code that you don't have to write and which doesn't clutter up your own application script!

The cost? It is all achieved via minor extensions to the definitions of the "Table" classes. The SchoolTable class declares dependent classes (only the Teacher dependency is declared in this example, but the declaration does expect an array of class names so one could have declared the Pupil dependency as well):

```
class SchoolTable2 extends Zend_Db_Table_Abstract {
    protected $_name = 'SCHOOL';
    protected $_primary = 'SCHOOLNAME';
    protected $_dependentTables = array('TeacherTable3');
}
```

The re-defined TeacherTable class contains the data that characterise the relationship (as represented by the foreign keys in the tables):

```
class TeacherTable3 extends Zend_Db_Table_Abstract {
    protected $_name = 'TEACHER';
    protected $_primary = 'EMPLOYEENUMBER';
    protected $_rowClass = 'TeacherClass';
    protected $_referenceMap    = array(
        'MapToSchool' => array(
            'columns'           => array('SCHOOLNAME'),
            'refTableClass'     => 'SchoolTable2',
            'refColumns'        => array('SCHOOLNAME')
        )
    );
}
```

The "schoolname" column in the Teacher table is a foreign key referencing the "schoolname" primary key column in the School table. The column specifications take arrays; this allows for cases where one of the tables has a composite primary key. If there are relationships with more than one table, there will be more than one "mapping rule". The rules are named for subsequent reference; here there is just the one rule "MapToSchool".

With these new table definitions, one can have code like the example UseRelatedTables.php script

which loads a School entity from the School table and then accesses its collection of teachers:

```
<!DOCTYPE HTML >
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Relational tables example</title>
    </head>
    <body>
        <h1>Relational tables</h1>
        <p>Loads a row from the school table, i.e. a school record, and then
            the 'dependent rowset' that has the teachers associated with that
            school.</p>
<?php

require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance();

require('SchoolTable2.php');
require('TeacherTable3.php');

// Global variables
$config = new Zend_Config_Ini('./config/appdata.ini',
                'standard');


function dbadaptersetup() { … }

dbadaptersetup();
$chosen = "Mount Druitt Public School";

$theSchoolTable = new SchoolTable2();
$chosenSchool = $theSchoolTable->fetchRow("SCHOOLNAME='$chosen'");
$theTeachers = $chosenSchool->findDependentRowset(
      "TeacherTable3", "MapToSchool");
print "<br><h2>$chosen</h2><br><h3>Teachers</h3><ul>";
foreach($theTeachers as $aTeacher) {
    print "<li>$aTeacher->FULLNAME</li>";
}
print "</ul>";
?>
    </body>
</html>
```

The findDependentRowset() method takes as arguments the name of the dependent table class and the rule identifier - “MapToSchool”.

3.3 Mapper classes

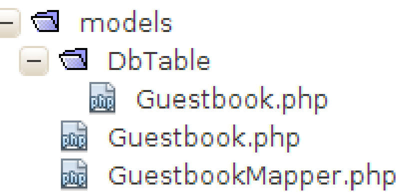
The two main Internet tutorials on the Zend framework (the QuickStart guide that is part of the framework documentation, and Oracle's “Human Relations Management” guide) have both chosen a more elaborate data model. The models used in these tutorials are supposed to exemplify Martin Fowler's “Data Mapper” design pattern.

Years ago, some great computer scientist pointed out that all problems in computing science can be solved by adding an extra level of indirection. A Data Mapper acts in a way as an extra level of indirection between the classes that you would like to work with in your business model and the persistent entities that you have represented in your relational tables. You end up with two kinds of entity object; one with minimal functionality tied closely to the relational schema, and one with maybe extensive domain functionality but no reference to any persistence mechanism. At a

minimum, the “data mapper” class organises the transfer to data back and forth between instances of these classes as needed.

One advantage that is often emphasised is that this separation should simplify testing. You can create the classes that represent you domain model, and develop these using a test-driven approach with constant unit testing – without ever having to bother about the final data persistence. Another argument for the separation is that since persistence is separate from domain model, you potentially have more choices for implementing persistence – maybe XML text files suit would you better than a relational database.

The example in the QuickStart tutorial involves maintaining a “guestbook”. The schema for the table defines an auto-id field, a varchar string field for the guest's email address, a text field for a visitor comment, and a datestamp. The program's model for these data involves three classes.



The class “Application_Model_DbTable_Guestbook” (the Zend framework has naming conventions that relate to directory and file path names – it's like Java, only worse) is just a simple class mapped directly to the underlying table – similar to the table classes illustrated earlier. Script code could use fetchAll() and fetchRow() methods of this class to retrieve entity objects that would be instances of std_class augmented with properties taken from the table meta-data.

```
class Application_Model_DbTable_Guestbook extends Zend_Db_Table_Abstract {
    protected $_name = 'guestbook';
}
```

The class “Application_Model_Guestbook” is the domain view of something similar. It is a completely independent PHP class with data fields for the email, comment, date, and id data. It has a collection of accessor (getX) and mutator (setX) methods for manipulating these data. (It also has overridden __get and __set methods that support property-style rather than function style access.)

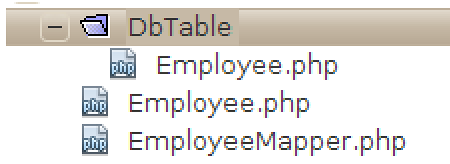
```
class Application_Model_Guestbook {

    protected $_comment;
    protected $_created;
    protected $_email;
    protected $_id;

    public function __construct(array $options = null) { … }

    public function __set($name, $value) { …  }

    public function __get($name) { …  }

    public function setOptions(array $options) { … }

    public function setComment($text) {
        $this->_comment = (string) $text;
        return $this;
    }

    public function getComment() { … }

    …
```

```
    public function getId() {
        return $this->_id;
    }

}
```

The “Application_Model_GuestbookMapper” class handles the relationship between the domain Guestbook and persistence DbTable_Guestbook classes.

```
class Application_Model_GuestbookMapper {
// Link to the Application_Model_DbTable_Guestbook class
// that handles persistence
    protected $_dbTable;
    public function setDbTable($dbTable) { … }
    public function getDbTable() { … }

    public function find($id, Application_Model_Guestbook $guestbook) {
      // Given primary key, use Application_Model_DbTable_Guestbook (table)
      // class to load in a row (as an instance of std_class with added
      // column properties)
        $result = $this->getDbTable()->find($id);
        if (0 == count($result)) {
            return;
        }
        $row = $result->current();
      // Copy the data out of that object into the instance of the domain
      // defined Application_Model_Guestbook class supplied as argument
        $guestbook->setId($row->id)
                ->setEmail($row->email)
                ->setComment($row->comment)
                ->setCreated($row->created);
    }

    public function save(Application_Model_Guestbook $guestbook) {
      // Given an instance of domain guestbook class
      //     if entry with that id exists in the database, invoke
      //     update() method of the table class passing array with data
      //
      //     otherwise invoke insert() method of table class to create
      //     new row
        …
    }

    public function fetchAll() {
      // Use the fetchAll method of the underlying DBTable_Guestbook class
      // Copy the data from the collection of std_class objects that it
      // returns into domain defined Application_Model_Guestbook objects that
      // are added to an array
      // return this array
      …
    }

}
```

The classes defined in Oracle's example are essentially identical.



Instead of a “guestbook” entry there is an Employee object with members for employee-id, first and last name, phone, salary, hire date etc. The Application_Model_DBTable_Employee class is just another extension of Zend_DB_Abstract_Table with no additions. The domain defined Application_Model_Employee class is an independent PHP class that defines an employee object with the various data members and provides standard accessor and mutator methods. The Mapper class is almost identical to the QuickStart Mapper; it adds a delete operation.

In my view, the use of these mapper classes is an unnecessary complication for these simple applications. The domain classes are simple entities – they have no additional complex domain functionality. (Entity classes that only have standard accessor and mutator methods really shouldn't require extensive independent unit testing – there is nothing to test!) The data mapper model may become useful when you start having complex domain functionality, or when there are additional data that may have meaning at run time but do not require persistent representations.

Alternatively, you might adopt the Mapper abstraction if you decide to give the mapper class responsibilities beyond just mimicking the find/findAll/save functionality of a table class. There will be situations where you have domain specific functionality that is required in several different use-cases. Rather than duplicate it among various control classes that handle the separate use cases, you might wish to place such code in a common element – possibly a mapper class can handle these functions as well as overseeing a mapping between domain defined and schema defined entity classes.

Study the mapper examples; the concepts will be useful sometime in your future. Stick with simpler solutions in your first attempts with the Zend framework.

3.4 Using Zend_DB, simple Zend components, and Smarty

Hopefully, the examples will have provided ideas of ways of improving your routine PHP applications.

Separating out the display (view) through the use of Smarty templates is often a big improvement. The PHP script can focus on issues of finding data, unencumbered with HTML and text strings. The Smarty template is close to standard HTML and can be prettied up using some HTML editors.

The Zend libraries offer well constructed, thoroughly tested components with particular strengths in areas like data validation. The Zend_DB components should simplify much of the code for data persistence.

All the same, you are probably disappointed. The world hasn't moved for you. It's still the same old PHP script talking to a database. The Zend libraries are good – but there are others. ADODB is another persistence library that is commonly used along with Smarty. So what is special?

For something special, you have to look at the full MVC framework.

4 Model-view-control and related issues

4.1 MVC

"Model-View-Control" - it must be one of the original "design patterns" for computer programs having been developed as part of the Smalltalk project at Xerox PARC back in the 1970s.

Xerox had conceived the possibility of automating all tedious office tasks. The kinds of applications imagined for Smalltalk were specialised editors – equivalent in modern terms to word-processor programs, spreadsheets, project-planners, diagramming tools, personalised data bases etc. Each office application involved some unique data. But much of the structure of these editor programs was perceived to be common. Xerox computer scientists conceptualised the commonalities and abstracted them out into groups of Smalltalk classes and a unifying paradigm for everything Smalltalk.

Smalltalk's unifying paradigm was "pick an object, give it a command". This paradigm works in the language itself – that is how Smalltalk code is written. It was also the basis of the user-interfaces that they pioneered. The data elements being manipulated in an office application would be presented to the user via iconic and other graphical representations. The user would be able to select a particular element, such as a paragraph of text in a document being manipulated in a word-processor editor. The user could then select a command, such as "change font", that was to apply to the selected text.

The Smalltalk framework classes conceived for such editors included "Application", "Document", and "Views". The Views were the classes for displaying data elements and for input of new content. The Smalltalk group created all the radio-button clusters, check-boxes, selection lists, scrolling text fields that make up the modern user interface; these were all specialised View classes. In any particular editor program, there would be links between specific View instances, such as a scrolling text field, and the data that were to be displayed in those views.

A Document would own a collection of data elements and take responsibility for transferring data between memory and files. Each different editor program would use a specialised subclass of the Xerox Document class; this class owned collections of instances of application defined data classes and shared with those classes the code for changing the underlying data. The more general term Model is now used rather than Document.

A desktop application, whether a modern Windows application or an early Smalltalk progenitor, receives inputs as interrupts – mouse click, key stroke. Code in classes associated with the Application class has to sort out such low level events. These events represent the user attempting either to select a data-element or to give a command to a selected element (examples – mouse click in a text entry box to activate it to receive subsequent key strokes, key-stroke to add character to currently active field, click and drag on text to select for subsequent command, click on a 'menu' display to change font). Control code has to resolve which elements are to be selected or which commands are to be applied to selected elements.

Although originating with desktop applications, the MVC model applies more widely. The web applications in section 3 above have clear view and model components. The Smarty templates, used to generate response pages, provide views of data subsequent to updates. The Zend_DB related components are the model.

Control is less clear.

The typical simple web application is composed of a set of separate scripts – each having a GET form for data entry and some POST processing for dealing with the data. These scripts correspond to individual use cases, or individual data manipulation commands.

The user's browser is actually doing a bit of the work. The web-browser handles all the low-level interactions (mouse-clicks, key-strokes etc) with the user and presents the server-side script with a complete package of input data. The script must then determine the object to that is selected and the modification action that is required. Often, both object and action are implicitly defined because the script exists simply to handle a specific action on a particular object in the model.

Each script has a defined control flow – typically something along the lines "*if the input data satisfy these tests do this data processing, else do error reporting*". The data processing part will select an element – e.g. row from a persistent table – and give it a command ("*update yourself*", "*delete yourself*").

There are other implicit flow-control structures. For example, options that are only available to registered users will appear as extra links in generated response pages.

The control is there. It's just rather inchoate, fragmented. With simple web applications that involve only a handful of scripts this doesn't matter. The close correspondence of scripts and use cases should make things sufficiently clear. However, with more ambitious projects, you might want a control element that is a little more formal, a little more structured.

4.2 Front Controller

The Front Controller is a much more recently formulated design pattern.

Essentially, the Front Controller recreates "the Application" - a central component that receives input data and resolves these input data to determine what part of the data model is to be changed and what action is to be performed on those model data. It then invokes the required actions (by calling methods of "action controller" classes). Finally, it gets View elements to present the user with a response that shows the updated state of the model.

Control is unified.

Rather than have separate control/action scripts, one has a central control element and "action controllers" - PHP classes with methods that perform just the specific updates on selected parts of the data model.

Often, reworking a large web application to fit the Front Controller pattern will result in code savings as well as increased clarity. Many of the separate scripts that previously handled individual data manipulation requests would have needed similar code checking things such as logged in status, role, access right. Some of the input validation code would probably have been duplicated in different scripts. Many such duplicated elements can be reworked into the controllers. Further, some such elements are common across all applications; these elements can be migrated into code that is provided as part of a framework.

The framework supplied code of a front controller based system, such as Zend Framework, will cover most of the following aspects:

Receipt of request (GET or POST);

Routing – identifying the action(s) that are to be performed;

Dispatching – invoking the actions, i.e. instantiating a specific "action controller" and calling the required action method in that controller;

Sending response.

The "action controller" classes will have the code for changing the data in the model. These classes will have to be defined as subclasses of some framework class so that they can be manipulated by standard framework code.

Your browser web client will still be making requests for many different resources identified by

distinct URLs. If these resources are things like CSS style sheets, Javascript files, or images then your web server should simply return the requested files. But requests for actions are also going to take the form of URL requests – but these requests have to be routed via the "FrontController" element.

Using the "school board" example, one might decide to consolidate the various operations possible into a number of main groups ("action controllers"); each group having one or more actions. For example, one might conceive the following:

- School Administration (SchoolController)
  - List schools
  - Add a school
- Teacher Administration (TeacherController)
  - List teachers at school
  - Create new teacher
  - Delete teacher
  - Update teacher record
  - List pupils in teacher's class
- Pupil Administration (PupilController)
  - List pupils at school
  - Create pupil
  - Delete pupil
  - Update pupil record

The requests coming to the web server will be for URLs like www.theSchoolBoard.org/TeacherAdmin/Update?employeeid=2001010001. Your Apache web server isn't going to find any resource with such a URL.

If you adopt the Front Controller model, you are going to have to instruct your web server on how to perform some elaborate gymnastics as it tries to return files or launch scripts. Those requests for simple things like CSS files should be handled normally; the other requests – well it's a matter of finding the appropriate Front Controller and getting it to sort out what must be done.

4.3 Technicalities - "*the devil is in the detail*"

4.3.1 Reconfigure your Apache web server now!

All requests for processing in a web application are now to go to that application's Front Controller – which will typically be implemented as the script "index.php" within the htdocs sub-directory corresponding to the specific web application.

The Apache web server has a module – the ReWrite module – which was originally introduced to remap file paths; it changes the paths that appear in requests into different paths that correspond to the true location of resources. Probably, when first introduced, this module simply facilitated reorganisation of the contents of an htdocs folder – resources could be moved even though their original URLs had been published. Subsequently, other uses have been found – such as dealing with embedded session keys for those paranoid web users who disable cookies. The rewrite module can also be given the rules needed to map appropriate requests to a front controller.

Of course, you don't want every request to your Apache server being diverted to the "front controller" part of a particular Zend Framework based application. Consequently, you do not edit your Apache web server's main httpd.conf file. Instead, you place the required re-write rules in a .htaccess file in the "*public directory of the Zend application*". (If you don't know what a .htaccess

file is, you don't have sufficient background knowledge to tackle anything like the Zend Framework! The Apache documentation has a limited explanation, but you probably need something more lengthy.) That main "*public directory of a Zend application*" – the Zend Framework has fairly strict requirements relating to the organisation of an application's files and directories. The basics will be illustrated in section 5.

The data for the Apache Rewrite module have to identify those files for which no redirection is required (CSS files etc) and specify the file to which other requests are re-directed (the "front controller" code). There is a a simple language for writing redirection rules. There are different ways in which the rules can be written – and each different tutorial that you find on the web probably illustrates another way.

One style is something like:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

A rough translation from Apache into English is "*check if the file-name in the request is that of an existing file, or link, or directory – in such a case leave it alone, and don't try any more rules; for all other cases change the file-name to index.php*". Other versions of the rule file may explicitly identify .css, .js, etc files as being the ones whose names are not rewritten.

There is one more detail. The location of that "index.php" file, with the front controller code, must be defined explicitly. (If you don't bother to explicate the full file path of the index.php file then you are likely to run into mysterious bugs – some parts of your Zend Framework application may actually work, but most will not.) There are different ways to specify the location of the front controller file.

Some on-line tutorials suggest that you create a "virtual host" on which to deploy the application. Basically, this means inventing a host URL, e.g. www.theSchoolBoard.org, allowing you to point a web client at www.theSchoolBoard.org/schooladmin etc. Your Apache's httpd.conf file has to be edited to include a section defining the virtual host; one element of the definition of a virtual host is its effective htdocs directory (each virtual host on your computer uses a different htdocs directory). Specifying the "public directory" of a Zend Framework application as being its htdocs directory supplies the information needed to unambiguously identify the magical front controller file.

This business of defining a virtual host is going to give you a lot of work for little gain. You will first have to edit your Apache httpd.conf file to include the virtual host directive. Editing the Apache config file to add a virtual host is probably illustrated in the tutorial; but this editing is the least of your problems.

There is no host www.theSchoolBoard.org (at least, I hope there isn't, I just made up the URL). So, your browser will fail when it tries to use a DNS name service to lookup an IP address for www.theSchoolBoard.org. So, now you have to edit the "hosts" file that exists on the machine where you will be running your browser to create a fake entry pointing at the IP address of the machine where you are running your Apache. (Browser and web-server can be the same machine; just use address 127.0.0.1.) It still doesn't work? Well possibly your web browser is configured to use a proxy server for accessing the web. If that is the case you will need to edit your browser's own configuration properties to avoid the use of the proxy for this URL. Good luck.

I prefer an alternative solution which I think is more appropriate for learning exercises. You provide the information, about the location of the front controller in with the other rules in your .htaccess file. The RewriteBase directive takes a (partial) file path defining the location. In the set of rules below, I am specifying that the index.php file will be in the sub-sub-folder SchoolBoard/public of my own public_html directory:

```
RewriteEngine On
RewriteBase /~nabg/SchoolBoard/public
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

4.3.2 Another thing – well several things – Zend_Layout, Zend_Form, Zend_View, and "Zend_Decorator"

Response pages from sophisticated applications are likely to have more elaborate structure than the simple examples illustrated in section 3. You are likely to want standard headers, navigation aids (both context specific and more general site-wide navigation), as well as outputs reporting the results for a specific action. These pages will be like those produced by content management systems such as Drupal or Joomla.

Those familiar with such content management systems will know that they produce pages using elaborate overall templates that have <div> sections that can be configured to contain output from the different plugins and scriptlets that make up the site. The "View" side of the Zend Framework can produce something as elaborate.

You will typically need a Zend_Layout component – this will define an overall template for your pages. You will define Zend_Views that will contain just the HTML output to show data produced by individual action requests. You define these things in .phtml files. Their role is a bit like the Smarty template files – lots of static content text and HTML markup and a little bit of scripting for tasks like "*display this data element here in this HTML div*", "*loop through this data collection outputting successive rows of a list or table*". Unlike Smarty which used its own scripting language, the scriptlet code in Zend .pthml pages is standard embedded PHP. The file that you define for a Layout element will be close to a complete HTML page; the files for view elements will be individual sections of HTML that can be slotted into the Layout's page. (You are not required to use a Layout component. The default Zend Framework structure created by the Zend scripts doesn't include a Layout. Instead of using a Layout, your individual views can be complete HTML pages.)

The Zend_Form class provides a programmatic way of building up a data entry form that is to be displayed for a GET request. You start with an empty form object and add input fields of various types. When the form is complete it can be told to render itself into HTML markup that will get slotted into a Zend_Layout page. (You can of course follow a more naïve approach and just have text with HTML markup. It is your choice.) The advantage of Zend_Form is that you can specify the validation requirements along with the form elements. The validation requirements will identify Zend validator classes and any parameter values that need to be supplied (e.g. maximum length for a string). Then on GET you would render the form; on POST you would populate the form with the posted data and ask it to validate itself.

"Decorators" allow the form designer to associate labels with input elements and, possibly, add more elaborate things. You can define "error" decorators; if data entered in a form fail validation tests, a version of the form can be re-displayed with explanatory comment next to fields where data were erroneous.

4.3.3 More "issues"

Your action controllers will need to access the data base. Obviously, they will use their own DB_Table ("model") classes, but what about the actual data base connection – the adapter. Will every action controller have to have code to create an adapter?

Your controllers will need to work with appropriate view objects that render their data, and work

also with a layout object. Where should such things be created? How are references to existing objects passed around?

The code that deals with things like creation of data base adapters can be centralised. The framework code provides a class with stub methods that can be edited to include the initialisation code for a particular application. Data in ".ini" configuration files can characterise the elements that are to be created.

The Zend Framework has conventions for passing references around, naming classes, and organising the files and directories. Learning the Zend Framework in part means learning to follow these conventions.

Classes for particular purposes have to be placed in specific sub-directories. Thus, DB derived classes should end up in the "models" sub-directory; .phtml files with view templates go in a "views" sub-directory; layout .phtml files find themselves in a "layouts" directory; action-controllers – well they are in the "controllers" sub-directory. More detailed naming conventions cover issues like how to relate the names of .phtml view files to the action controllers that require them.

The Zend framework comes with a number of scripts (.sh or .bat files). (Actually, it is just one script that takes lots of command line arguments.) These scripts "know" the naming conventions. They take command line parameters and handle requests like

"*setup a new Zend Framework project*"
(creates all the required directories and inserts a few stub files)

"*provide a data base adapter that will talk to this MySQL server with this user-password combination*"
(the application.ini file is edited)

"*add an action to this controller*"
(inserts an extra method in the controller class, and creates a view template file that will handle output for that action)

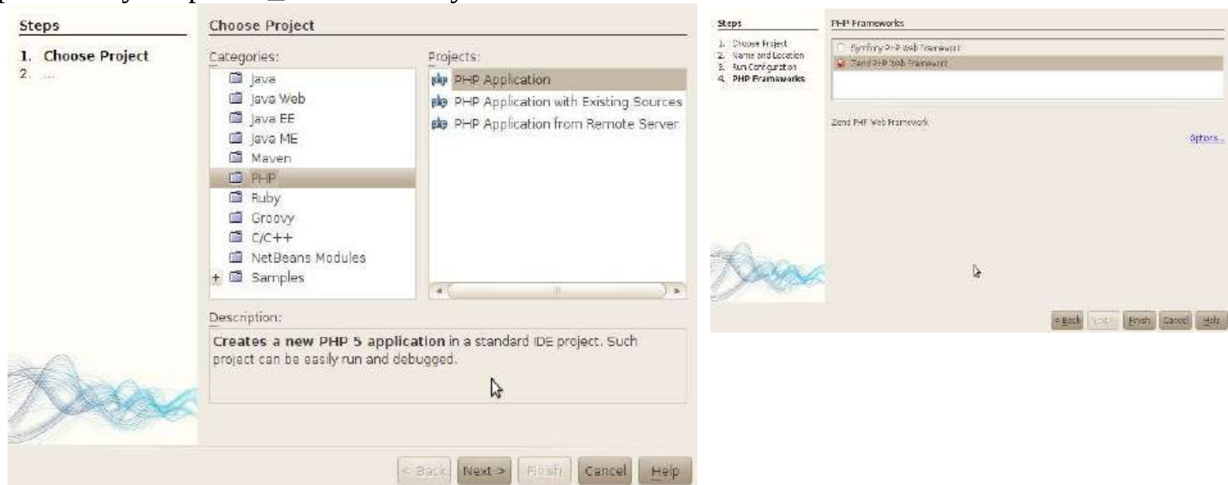
"*add a DB table class*"
(if necessary, creates a new sub-directory within the "models" part of the application and defines a new subclass of DB_Table_Abstract).

5 Zend's MVC framework

There is a streaming video tutorial at netbeans.org that illustrates the steps needed to configure the Netbeans IDE to work with the Zend framework. Once you have configured your Netbeans, you can create PHP projects that will be based on the Zend framework. With these projects, Netbeans runs the appropriate Zend script that creates the required directory structure with all necessary sub-directories and provides the various stub files and configuration files that every Zend application requires. With Zend-PHP projects, you will also have an additional "right-click" option for the project - "run Zend Command". Use of this option will allow you to select a Zend script and supply parameters through a dialog.

5.1 Starting out – what do you get? An application that advertises the Zend framework!

Create a new PHP project, ZF01, that specifies the use of the Zend framework; this should be placed in your public_html directory.

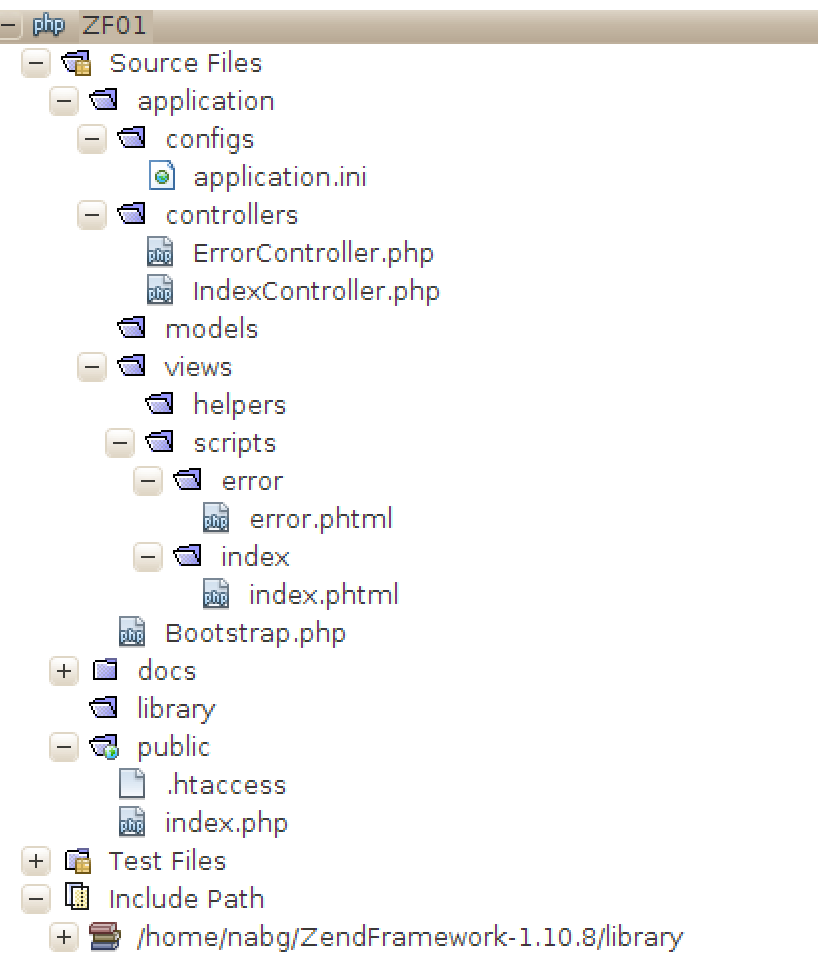


If Netbeans has been properly configured to work with the Zend framework, it will run the required Zend supplied .sh or .bat scripts to create the basic configuration with directories for "application" (essentially all the source code and configuration files), "docs" and "library" (place-holders for the documentation that you will of course remember to write, and for copies of libraries that you want deployed with this application), "tests" (stubs to help you set up some PHP unit testing of your own classes), and "public" (there will also be a "nbproject" directory for Netbeans own house-keeping files). The "public" directory is the application as deployed – it has the essential .htaccess file with the rewrite rules for the "front-controller" setup, and the "index.php" file that is the "front-controller" application.

```
nabg@info-bxsrr1s:~/public_html/ZF01$ ls
application  docs  library  nbproject  public  tests
nabg@info-bxsrr1s:~/public_html/ZF01$ ls -la public
total 16
drwxr-xr-x 2 nabg nabg 4096 2011-05-02 11:40 .
drwxr-xr-x 8 nabg nabg 4096 2011-05-02 11:40 ..
-rw-r--r-- 1 nabg nabg  193 2011-05-02 11:40 .htaccess
-rw-r--r-- 1 nabg nabg  753 2011-05-02 11:40 index.php
```

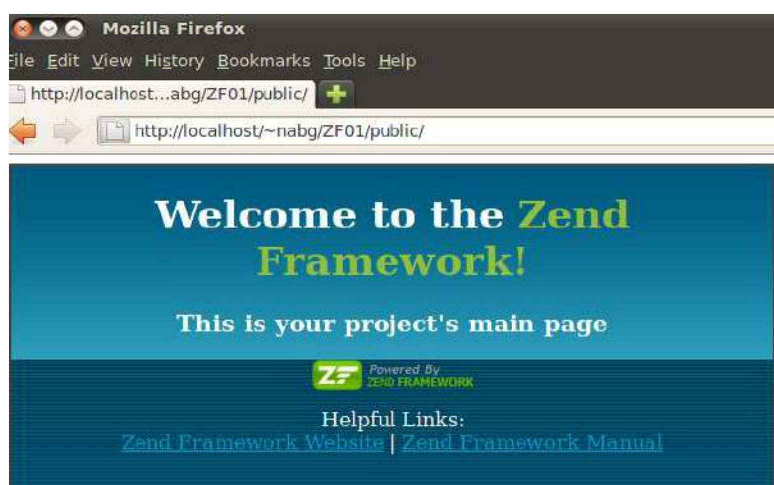
You will rarely change the index.php file – it has a little bit of standard Zend supplied code that basically says "*create an instance of Zend_Application using the models and controllers in these sub-directories, and make it run*".

The “application” directory has sub-directories for “models”, “controllers”, “views”, and configuration data. It also holds a Bootstrap.php file – this you may edit to add extra application specific initialisation code; simple applications shouldn't need to change the bootstrap.



The Zend project set up script has in fact created a complete application – one that advertises the Zend framework. This application involves instances of the auto-generated IndexController class and its associated view template file, index.phtml, in the views/index sub-directory. (The “models” directory is empty; there are no data yet to model.)

If your local Apache server has been correctly configured to allow user-directories and directory indexing using scripts, you can invoke this auto-generated application by aiming a browser at http://localhost/~*youruserid*/ZF01/public.



Generation of the advertisement page involved a series of default operations:

1 The request ~nabg/ZF01/public was for a directory rather than a file, so the Apache web server ran the public/index.php script (as specified by the DirectoryIndex parameter in its configuration file).

2 The index.php script provided by Zend created an instance of Zend_Application and let it run.

3 Because nothing more specific was requested, the Zend_Application created an instance of the IndexController class as defined in the local models directory, and invoked its indexAction method which in principle generated some additional data that could be slotted into the index/index.phtml template.

```
<?php

class IndexController extends Zend_Controller_Action
{

    public function init()
    {
        /* Initialize action controller here */
    }

    public function indexAction()
    {
        // action body
    }


}
```

4 When the IndexController object completed its index method, the Zend_Application object invoked a view renderer component that then ran the code for the actual index/index.phtml file. (This defines a fragment of HTML, not a complete page; if there were a "Layout" component defined, this would provide a page, and a place for this fragment).

```
<style>
    a:link,
    a:visited
    {...}

    span#zf-name
    {...}

    div#welcome
    {...}

    div#more-information
    {...}
</style>
<div id="welcome">
    <h1>Welcome to the <span id="zf-name">Zend Framework!</span></h1>

    <h3>This is your project's main page</h3>

    <div id="more-information">
        <p><img src="http://framework.zend.com/images/PoweredBy_ZF_4LightBG.png" /></p>
        <p>
            Helpful Links: <br />
            <a href="http://framework.zend.com/">Zend Framework Website</a> |
            <a href="http://framework.zend.com/manual/en/">Zend Framework Manual</a>
        </p>
    </div>
</div>
```

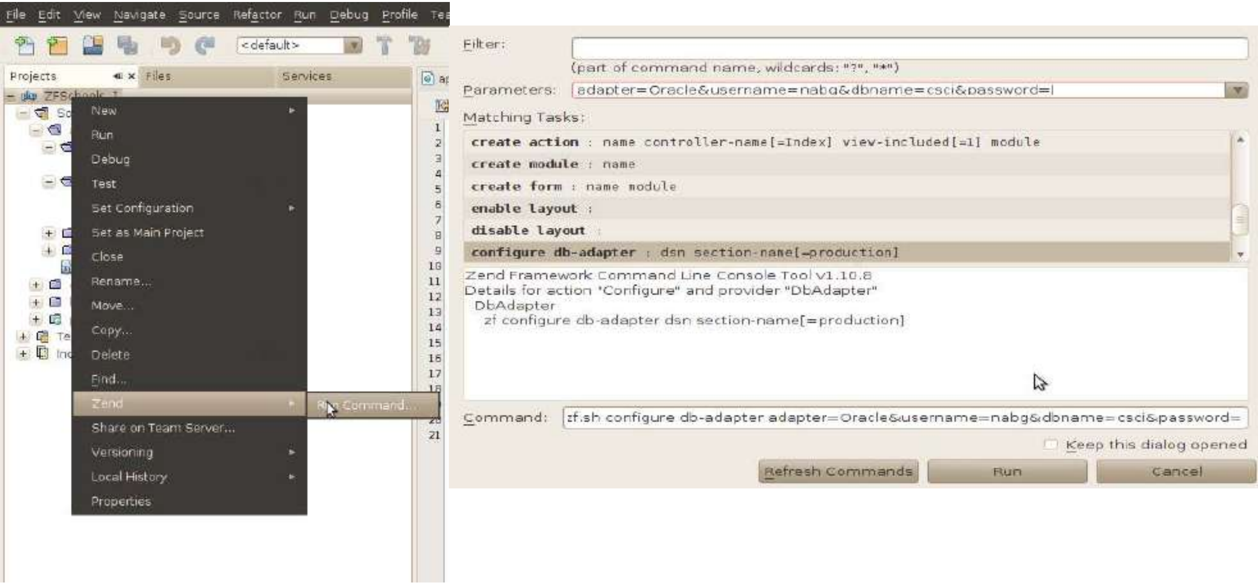
Note – we didn't set the “RewriteBase” value in the generated .htaccess file – but this demo application still “worked” (it didn't actually do anything – but it did its nothingness quite well). Sigh. That is why things can get so confusing. For any application where you actually want something done, you will need to change the .htaccess file. In this case, the extra line “RewriteBase /~nabg/ZF01/public” should have been inserted!

5.2 ZFSchools_1: a minimal Zend Framework application with a model.

Create a new PHP project, ZFSchools_1. This Zend application is going to permit viewing lists of schools and teachers in the schools data base created earlier.

5.2.1 Configuration

This application will need to access the database – so a first step might be to supply the parameter data such as user-name, password, database identifier etc. These data should be added to a configuration file for the application. The Zend Framework supplies a script that will get the required data elements and add them to the appropriate file:



This runs the configure script using parameter to define the adapter (in this case Oracle, could be PDO_MySQL etc), username, password, and database name.

Running this script (via the Netbeans dialog or manually at the command line) results in a change to the auto-generated application.ini file. The extra lines relating to the data base are added.

```
[production]
phpSettings.display_startup_errors = 0
phpSettings.display_errors = 0
includePaths.library = APPLICATION_PATH "/../library"
bootstrap.path = APPLICATION_PATH "/Bootstrap.php"
bootstrap.class = "Bootstrap"
appnamespace = "Application"
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"
resources.frontController.params.displayExceptions = 0
resources.db.adapter = "Oracle"
resources.db.params.username = "nabg"
resources.db.params.dbname = "wraith:1521/csci"
resources.db.params.password = "notmypassword"

[staging : production]
```

```
[testing : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1

[development : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
resources.frontController.params.displayExceptions = 1
```

The rest of this application.ini file is just Zend's standard. It defines the locations of controllers, libraries etc. (It is possible to change these locations if you had some really good reason to configure your files and directories in some way different from Zend's standard; you would just change the paths specified here.)

Like other .ini files, this file has "sections". "Staging", "testing", and "development" are all specialisations of "production". Here, "development" overrides some values that define details of how the framework is to display errors. If you don't specify which version you want, the Zend framework will default to using the "production" settings. (There are a number of ways of specifying a different choice, one being to add a line like "SetEnv APPLICATION_ENV development" in your Apache's httpd.conf file.)

While dealing with configuration, it would be wise to set the RewriteBase value in the .htaccess directory:

```
RewriteEngine On
RewriteBase /~nabg/ZFSchools_1/public
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

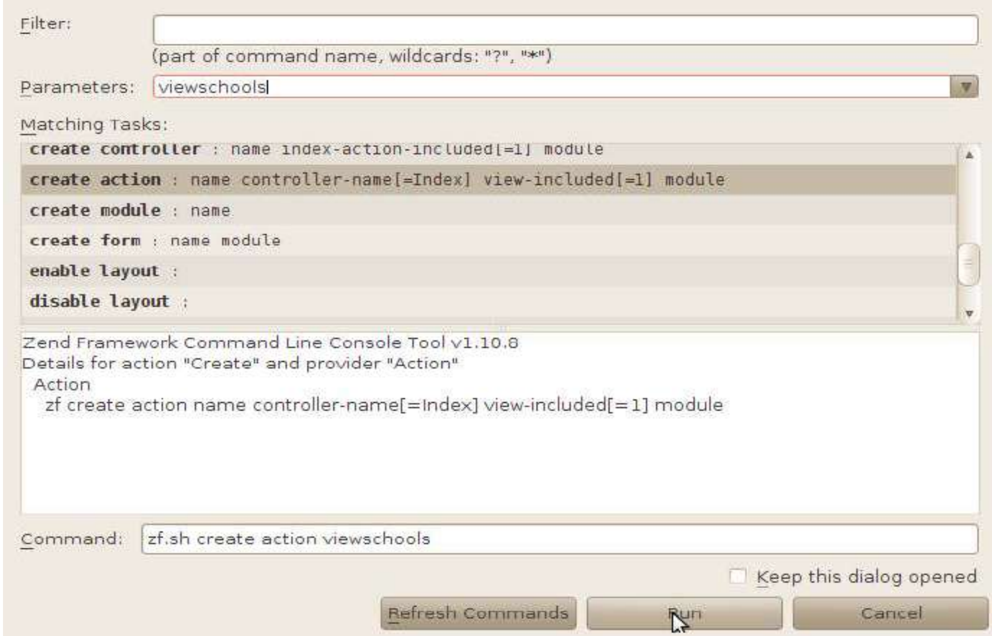
5.2.2 Adding some actions to the IndexController

In this very simple application, we want simply to add a couple of "view actions" - view all schools and view all teachers. It will be simplest if these are made actions that can be performed by the IndexController that was auto-generated. It will support three actions:

- "index" - show a page with "links" to the "show teachers" action and to the "show schools" action.
- "view schools" - list all the school names and their addresses.
- "view teachers" - list all the employed teachers.

This will involve modification of the IndexController class and its defaults view and creation of additional view classes – there will be a view template to show the result of "view schools" and another to show the result of "view teachers". It is easiest if you let the Zend framework handle the basics of these changes.

Invoke the "create action" script. This allows the definition of a new action associated with a specified controller (defaulting to the IndexController):



When run, this script modifies the code of the IndexController class and creates a new view template:

```
<?php

class IndexController extends Zend_Controller_Action
{

    public function init()
    {
        /* Initialize action controller here */
    }

    public function indexAction()
    {
        // action body
    }

    public function viewschoolsAction()
    {
        // action body
    }


}
```

```
ZFSchools_1
  Source Files
    application
      configs
        application.ini
      controllers
        ErrorController.php
        IndexController.php
      models
      views
        helpers
        scripts
          error
          index
            index.phtml
            viewschools.phtml
      Bootstrap.php
    docs
    library
```

The views/scripts/index/viewschools.phtml file simply has a fragment of HTML that can act as a place-holder for now:

```
<br /><br /><center>View script for controller <b>Index</b> and script/action
name <b>viewschools</b></center>
```

The scripts/index/index.phtml template file generated by the create project command contains the markup for the Zend Framework advert. This must be changed so that we get the beginnings of the school's web site with a link that will invoke the viewschoolsAction.

```
<div id="welcome">
    <h1>Welcome to the new School Board site</h1>

    <h2>Options</h2>
    <ol>
        <li><a href=
```

```
            "<?php echo $this->url(
                  array('controller'=>'index','action'=>'viewschools'));?>" >
            View schools</a></li>
    </ol>
</div>
```

The link has to be composed using functions in the Zend framework library; the url() function will create an appropriate link referencing the application, controller, action and, if appropriate query string arguments. The HTML source actually generated is:

```
<div id="welcome">
    <h1>Welcome to the new School Board site</h1>

    <h2>Options</h2>
    <ol>
        <li><a href=
               "/~nabg/ZFSchools_1/public/index/viewschools" >
            View schools</a></li>
    </ol>

</div>
```

(Application = /~nabg/ZFSchools_1/public/, controller=index, action=viewschools.)

Aiming a browser at localhost/~nabg/ZFSchools_1/public results in a page with a working link to viewschools that displays the auto-generated place-holder markup:

Welcome to the new School Boa

Options

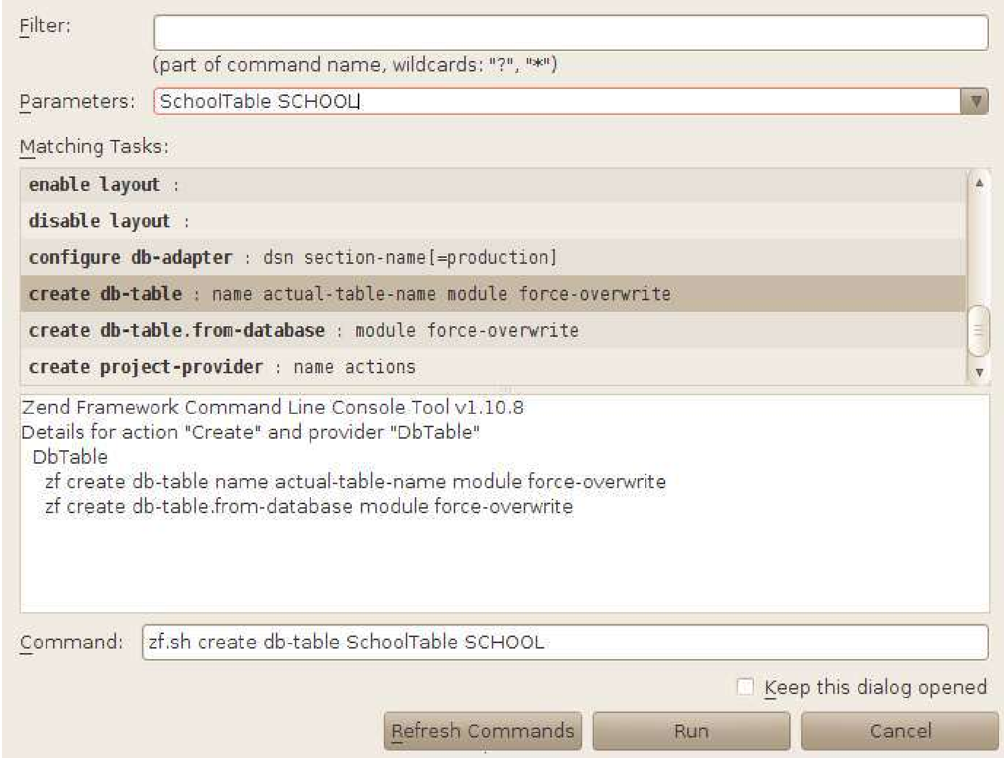
1. View schools

View script for controller **Index** and script/action name **view**

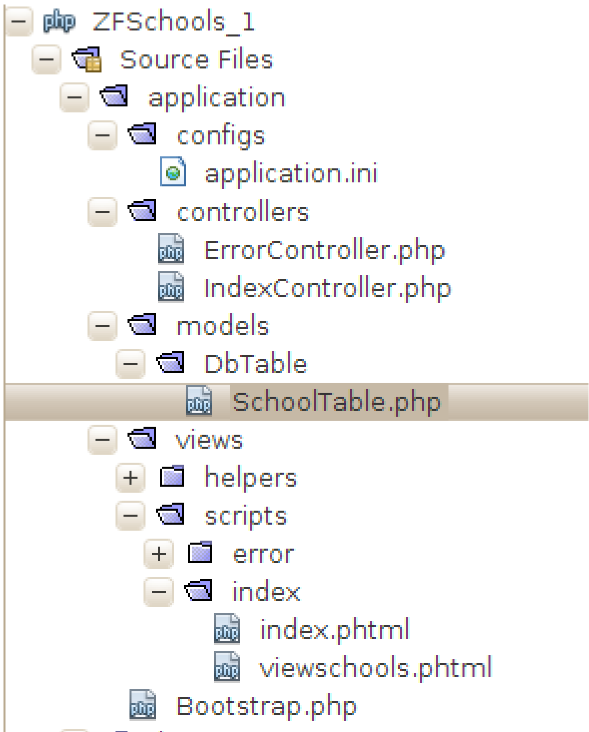
5.2.3 Defining a model

If we actually want a list of the schools, we will need some effective code in the viewschoolsAction() method that will retrieve data from the database, and some markup code that in the viewschools.phtml file that will show the retrieved data. If we want data from the database, we must start by defining the “Model” - some classes that will provide access to the persistent data.

The zf.sh (or zf.bat) script can be run with options to create a suitable class derived from Zend_DB_Table; of course, in the NetBeans environment this is done via dialogs:



When this script is run, it generates the new DB class and adds it in a subdirectory of the models directory:



The generated code is as simple as all the other Zend_DB classes that were illustrated in section 3 above:

```
<?php

class Application_Model_DbTable_SchoolTable extends Zend_Db_Table_Abstract
{

    protected $_name = 'SCHOOL'; // upper case needed for Oracle connection
```

```
}
```

Application_Model_DbTable_SchoolTable – what a complex class name!

As I mentioned above, the naming scheme for classes is analogous to that used in Java where one has classes like java.awt.Button, javax.servlet.HttpServlet etc. In Java, the “packages” (java, awt, javax etc) correspond to directories on a presumed path to the file with the class definition. Here in the Zend Framework, the name segments Application, Model, and DbTable also correspond to directories on the path to the class. Code within the Zend framework that links in classes dynamically uses the naming scheme to find the files required.

There is another oddity – the absence of a ?> at the end of the file. Apparently, having the close script bracket, and possibly more whitespace characters following, can result [in spurious blank lines appearing in the final HTML that gets generated] (or something like that).

The viewschoolsAction() code will need to use an instance of this class to access the schools table and retrieve all records:

```
    public function viewschoolsAction()
    {
        $schoolTableAccess = new Application_Model_DbTable_SchoolTable();
        $schools = $schoolTableAccess->fetchAll();
        $this->view->schooldata = $schools;
    }
```

(Note that no data base adapter is specified when the DB access class instance is created. Somewhere in the Zend library code there are a few lines that use the information in the application.ini file to create an adapter and define it as the default adapter.) The retrieved data are passed to the class generated from the associated viewschools.phtml file by assigning to a newly invented data member.

The actual view template needs a little PHP scripting to loop through the collection.

```
<?php
$this->title = "Schools";
$this->headTitle($this->title);
?>
<table border="1">
<tr>
<th>School name</th>
<th>Address</th>
</tr>
<?php foreach($this->schooldata as $school) : ?>
<tr>
<td><?php echo $this->escape($school->SCHOOLNAME);?></td>
<td><?php echo $this->escape($school->ADDRESS);?></td>
<?php endforeach; ?>
</table>
```

With these extensions, we can get a listing of schools:

School name	Address
Huntley Vale Primary	Landsdown Road, Huntley
Lady of Mercy Primary	Collette Road, Huntley
Meridian Public School	Headlands Drive, Perry
Jane Dean Primary	Bligh Road, Dulles
Mount Druitt Public School	100 Overlook Road, Druitville

5.2.4 What about the teachers?

Implementing the option to list all teachers involves:

1 Creating another “model class” - Application_Model_DbTable_TeacherTable;

2 Creating the viewteachers action in the IndexController and the corresponding viewteachers.phtml view template.

3 Implementing the Action function and defining a layout.

There are rather more teachers than there are schools, so it would be appropriate to utilise Zend's paginator component that was illustrated for the pupils listings in section 3.1.3.

There is of course nothing to see in the Application_Model_DbTable_TeacherTable class – just the one property that defines the table name. The code in the action function is quite similar to that illustrated previously (about the only difference is that it is using a DB_Table_Select rather than a DB_Adapter_Select – and that difference isn't obvious in the code):

```
class IndexController extends Zend_Controller_Action {

    public function init() {
        /* Initialize action controller here */
    }

    public function indexAction() {
        // action body
    }

    public function viewschoolsAction() { … }

    public function viewteachersAction() {
        $teacherTableAccess = new Application_Model_DbTable_TeacherTable();
        $page = $this->_getParam('page', 1);
        $select = $teacherTableAccess->select();
        $select->from('TEACHER',
                  array('TITLE', 'FIRSTNAME', 'INITIALS', 'SURNAME'))->
                        order(array('SURNAME'));
        $paginator = Zend_Paginator::factory($select);

        $paginator->setCurrentPageNumber($page);
        $paginator->setItemCountPerPage(10);

        $thepages = $paginator->getPages();
        $pagerange = $thepages->pagesInRange;
        $last = 1+ (int)($paginator->getTotalItemCount()/10);
        $this->view->paginator = $paginator;
        if($page>1) $this->view->previous = $page-1;
        if($page<$last) $this->view->next = $page + 1;
        $this->view->pagesInRange = $pagerange;
        if($last>1) $this->view->pageCount = $last;

    }

}
```

The form of the viewteachers.phtml template comes almost directly from the paginator example in the Zend Framework documentation:

```
<h1>Teachers</h1>
<?php if (count($this->paginator)): ?>
<ul>
<?php foreach ($this->paginator as $item): ?>
```

```
  <li><?php echo $item->TITLE . ' ' . $item->FIRSTNAME . ' ' .
          $item->INITIALS . ' ' . $item->SURNAME ; ?></li>
<?php endforeach; ?>
</ul>
<?php endif; ?>

// From the Zend documentation!
<?php if ($this->pageCount): ?>
<div class="paginationControl">
<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?php echo $this->url(array('page' => $this->previous)); ?>">
    &lt; Previous
  </a> |
<?php else: ?>
  <span class="disabled">&lt; Previous</span> |
<?php endif; ?>

<!-- Numbered page links -->
<?php foreach ($this->pagesInRange as $page): ?>
  <?php if ($page != $this->current): ?>
    <a href="<?php echo $this->url(array('page' => $page)); ?>">
        <?php echo $page; ?>
    </a> |
  <?php else: ?>
    <?php echo $page; ?> |
  <?php endif; ?>
<?php endforeach; ?>

<!-- Next page link -->
<?php if (isset($this->next)): ?>
  <a href="<?php echo $this->url(array('page' => $this->next)); ?>">
    Next &gt;
  </a>
<?php else: ?>
  <span class="disabled">Next &gt;</span>
<?php endif; ?>
</div>
<?php endif; ?>
```

With these additions, the application can display the teachers ten at a time:

Teachers

- Miss Agnes R Letetsu
- Miss Susan J Mortlake
- Miss Jackie S O'Connor
- Ms Mary P O'Farrel
- Miss Jane P Ogilvy
- Miss Karol M Parkinson
- Ms Joanne R Peach
- Ms Therese Z Percival
- Mrs Karla T Petrus
- Mrs Ivy B Piper

< Previous | 1 | 2 | 3 | 4 | 5 | Next >

5.2.5 Make it pretty! Add a "Layout"

The application can be prettied up by adding a layout. To start, you just run the "enable layout" option with the zend.sh script; this creates a new layouts directory within the application directory, this has a "scripts" sub-directory with a default layout.phtml file. I suppose one should also add a stylesheet; for this, I created a css sub-directory within the public directory of the application and added a very simple .css file created using Netbeans' CSS editor. I also copied the "images" directory from the section 3 examples to the application's public directory.

```
body {

    background-color: #f7f3e8;
    font-family: Georgia,'Times New Roman',times,serif;
    color: #2b3e42;
}

h1 {
    font-size: 1.5em;
    color : #77bed2;
}

h2 {
    font-size: 1.2em;
}
```

The layout.phtml file provides the overall page definition with a <div> for content. The Zend framework's view renderer will arrange to drop the HTML from an action's view into this <div>.

```
<?php
$this->headMeta()->appendHttpEquiv('Content-Type', 'text/html;charset=utf-8');
$this->headTitle()->setSeparator(' - ');
$this->headTitle('The School Board');
echo $this->doctype();
?>
<html >
    <head>
        <?php
        echo $this->headMeta(),
        $this->headTitle(),
        $this->headLink()->appendStylesheet(
                $this->baseUrl() . '/css/mystyle.css');
        ?>
    </head>
    <body>
        <div id='headdiv'>
            <img src="<?php echo $this->baseUrl() . '/images/kids.jpeg'; ?>"
/>
        </div>
        <h1><?php echo $this->headTitle(); ?></h1>
        <div id="content">
            <?php echo $this->layout()->content; ?>
        </div>
    </body>
</html>
```

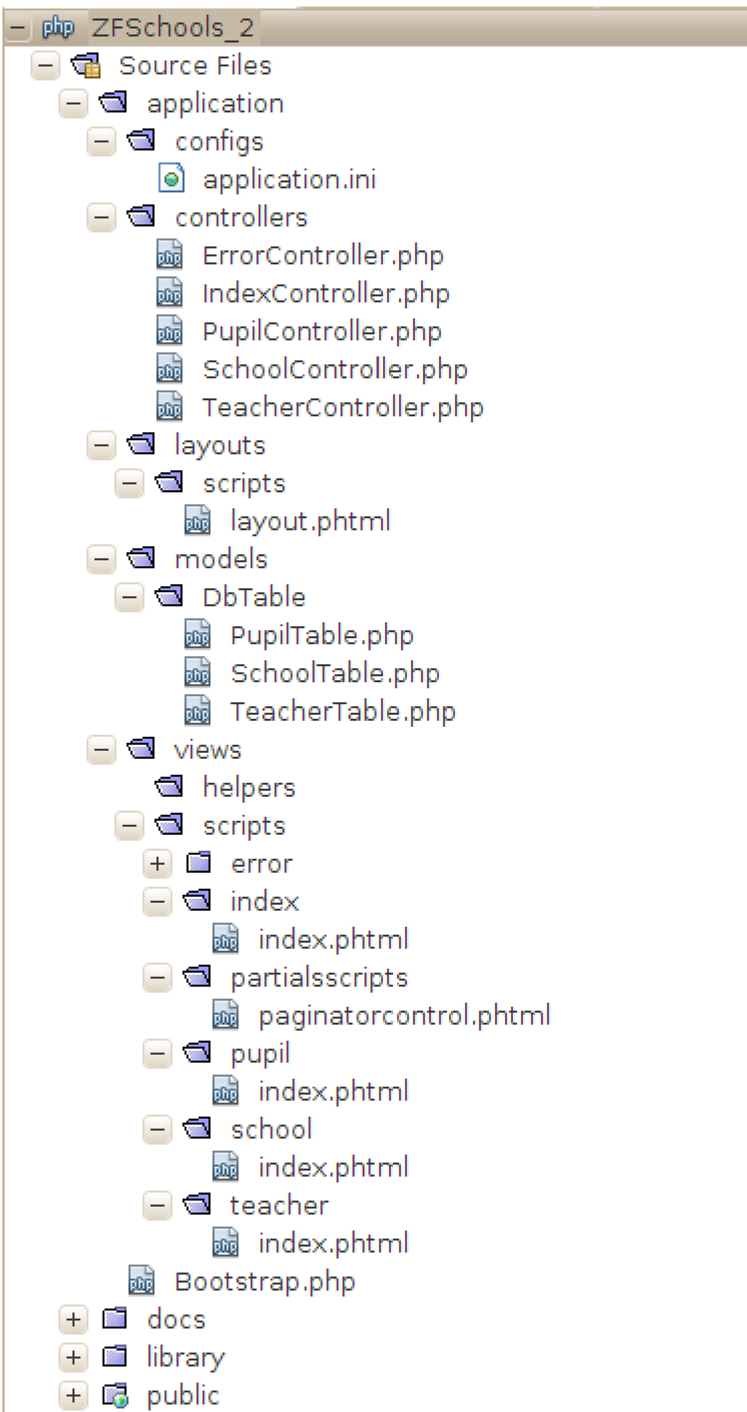
With these additions, the pages are (somwhat) "prettier".

5.3 CRUD

Web developers seem to spend so much of their lives working on CRUD applications. We had better extend the Zend Schools application with some 'create, read, update, delete' operations.

5.3.1 ZFSchools_2: More Controllers, more …

The Schools application will be extended, little by little, to add operations like “add school”, “update teacher/pupil” etc. Rather than have all the actions in one big controller, distinct controllers will be defined for “School”, “Teacher” and “Pupil”.



A new Netbeans ZF project, ZFSchools_2, was created. The public/CSS and public/image directories were copied form the earlier project. The new application.ini file had the database connection parameters cut-and-pasted (it's quicker than using the zf.sh script to add a data base adapter!), and the RewriteBase value was added to the .htaccess file. Then, zf.sh commands were

run:

1 'create db-table' for each of the SCHOOLS, TEACHERS, and PUPILS tables – resulting in the three classes in models/DbTable.

2 'create controller' for School, Teacher, and Pupil – resulting in the classes SchoolController etc added to the controllers/ directory, and the corresponding "index.phtml" scripts in school, teacher, and pupil sub-directories of views/scripts.

3. 'enable layout' – adds the layouts directory and the layout.phtml file.

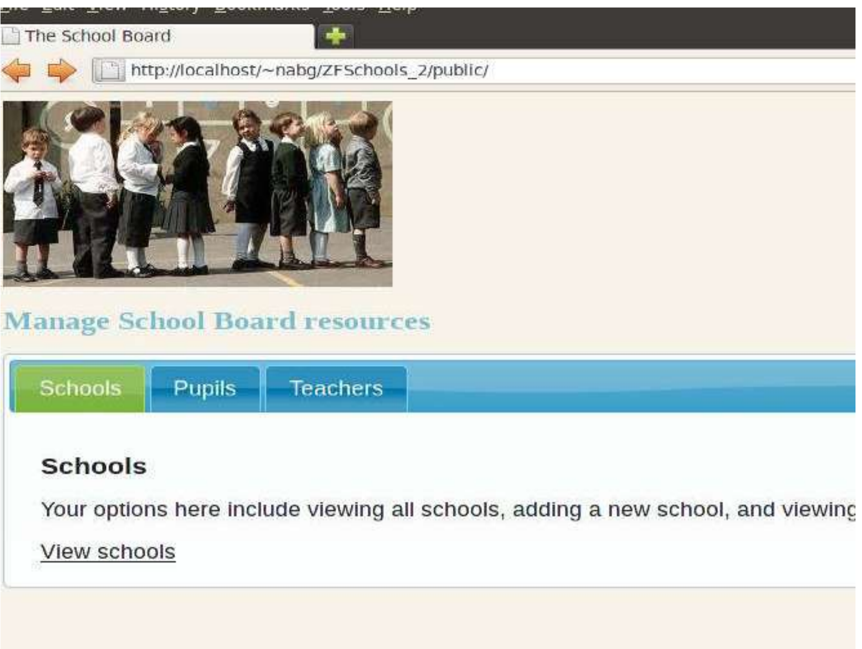
The new layout.phtml file is an extension of that in the previous section – this one includes links to the jQuery library and UI library downloaded from Google. (The Zend Framework has some helper classes that can be used to add jQuery or Dojo Javascript components to pages. Use of these helpers is beyond the scope of this introductory tutorial.)

```
<html >
    <head>
        <?php
        echo $this->headMeta(),
        $this->headTitle(),
        $this->headLink()->appendStylesheet(
                $this->baseUrl() . '/css/mystyle.css');
        ?>
        <!-- Use Google's copy of the JQuery stuff -->
        <link href="http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.11/themes/start/jquery-ui.css"
              type="text/css" rel="Stylesheet" />

        <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.5.1/jquery.min.js"><
        <script type="text/javascript"  src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.11/jquery-ui.mi

        <script type="text/javascript">
            $(document).ready(function() {
                $('#tabs').tabs();
            });
        </script>
    </head>
    <body>
        <div id='headdiv'>
            <img src="<?php echo $this->baseUrl() . '/images/kids.jpeg'; ?>" />
        </div>
        <h1><?php echo $this->headTitle(); ?></h1>
        <div id="content">
            <?php echo $this->layout()->content; ?>
        </div>
    </body>
</html>
```

The main index() action in the IndexController is now simply to provide links to the more specialised controllers. The index() function itself is empty; all the work is in the associated views/scripts/index/index.phtml file -

```
h1>Manage School Board resources</h1>
<div id="tabs">
    <ul>
        <li><a href="#tab-1">Schools</a></li>
        <li><a href="#tab-2">Pupils</a></li>
        <li><a href="#tab-3">Teachers</a></li>
    </ul>

    <div id="tab-1">
        <h2>Schools</h2>
        <p>Your options here include viewing all schools, adding a new school,
            and viewing lists of pupils or teachers at a selected school.</p>
        <a href=
               "<?php echo $this->url(
                array('controller' => 'school', 'action' => 'index')); ?>" >
                View schools</a>
```

```
    </div>
    <div id="tab-2">
        <h2>Pupils</h2>
        <p>Your options here include viewing/updating details of an individual
        pupil, adding or deleting a pupil record.</p>
        <a href=
               "<?php echo $this->url(
                  array('controller' => 'pupil', 'action' => 'index')); ?>" >
                View pupils</a>
    </div>
    <div id="tab-3">
        <h2>Teachers</h2>
        <p>Your options here include viewing/updating details of an individual
        teacher, adding or deleting a teacher record.</p>
        <a href=
               "<?php echo $this->url(
                  array('controller' => 'teacher', 'action' => 'index')); ?>" >
                View teachers</a>
    </div>

</div>
```



Each of the links invokes index action of the appropriate controller.

In first step of implementation, we can have these three controllers' index() actions each simply list all schools, all pupils, or all teachers. The code needed will just involve a reworking of the viewschools() and viewteachers() actions illustrated in the previous section.

There are relatively large numbers of teachers and pupils, so in both cases one would want to use pagination. One wouldn't want the code that places the pagination controls duplicated in both the .phtml scripts (the one for listing teachers, the other for pupils); it would be much better if such code could be shared.

The Zend Framework makes special provision for shared elements used to generate parts of responses. There are “view helper classes” and “partial views”. There are numerous standard view helper classes (a couple will be illustrated later); new helper classes can be defined and placed in the views/helpers folder. Partial views can be defined and called from code in a views/scripts.phtml file. These features are a little bit too complex for this tutorial; but I'm following their example by

having a “partial script”.

I have added a sub-directory, partialsscripts, to the views/scritps directory and added code that will display the pagination controls and which will be invoked from the view script files for both students and teachers. The necessary data (page range, links for previous and next etc) will be passed to this common code via a hash array.

The implementation of the three controllers is very similar at this stage. Each just has an index() action that retrieves all data. The TeacherController can serve as an example:

```php
<?php

class TeacherController extends Zend_Controller_Action
{

    public function init()
    {
        /* Initialize action controller here */
    }

    public function indexAction()
    {
        // Get table access object.
        $teacherTableAccess = new Application_Model_DbTable_TeacherTable();
        // Gwhen first invoked via link on IndexController's index page
        // there will be no argument, when subsequently invoked via
        // pagination controls there will be a page=... argument
        $page = $this->_getParam('page', 1);
        // Build basic select statement, modified by paginator code
        $select = $teacherTableAccess->select();
        $select->from('TEACHER', array
            ('TITLE', 'FIRSTNAME', 'INITIALS', 'SURNAME'))->
                order(array('SURNAME'));
        $paginator = Zend_Paginator::factory($select);

        $paginator->setCurrentPageNumber($page);
        $paginator->setItemCountPerPage(10);

        // Paginator adds constraints to limit to 10 results
        // starting at appropriate record in sorted list
        // and retrieves data
        $thepages = $paginator->getPages();
        $pagerange = $thepages->pagesInRange;
        $last = 1+ (int)($paginator->getTotalItemCount()/10);
        // Forward paginator to view script along with an array
        // containing pagination control data
        $this->view->paginator = $paginator;
        $paginatordata = array();

        if($page>1) $paginatordata['previous'] = $page-1;
        if($page<$last) $paginatordata['next'] = $page + 1;
        $paginatordata['pagesInRange'] = $pagerange;
        $paginatordata['pageCount'] = $last;

        $this->view->paginatordata = $paginatordata;

    }


}
```

The view script, views/scripts/teacher/index.phtml, has the code to create a HTML list with teacher data and the statement that invokes the partial script that outputs the HTML for the controls:

```
<h1>Teachers</h1>
<?php if (count($this->paginator)): ?>
<ul>
<?php foreach ($this->paginator as $item): ?>
  <li><?php echo $item->TITLE . ' ' . $item->FIRSTNAME . ' ' .
          $item->INITIALS . ' ' . $item->SURNAME ; ?></li>
<?php endforeach; ?>
</ul>
<?php endif; ?>
<?php echo $this->partial(
      '/partialsscripts/paginatorcontrol.phtml',
      $this->paginatordata);
?>
```

The Zend Framework code for the partial() method of view turns the argument data ($this->paginatordata) into members of the object that is creating the output from the partial script. The file views/scripts/partialsscripts/paginatorcontrol.phtml has just the code to output the pagination controls:

```
<?php

if ($this->pageCount >0): ?>
<div class="paginationControl">
<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?php echo $this->url(array('page' => $this->previous)); ?>">
    &lt; Previous
  </a> |
<?php else: ?>
  <span class="disabled">&lt; Previous</span> |
<?php endif; ?>

<!-- Numbered page links -->
<?php foreach ($this->pagesInRange as $page): ?>
  <?php if ($page != $this->current): ?>
    <a href="<?php echo $this->url(array('page' => $page)); ?>">
        <?php echo $page; ?>
    </a> |
  <?php else: ?>
    <?php echo $page; ?> |
  <?php endif; ?>
<?php endforeach; ?>

<!-- Next page link -->
<?php if (isset($this->next)): ?>
  <a href="<?php echo $this->url(array('page' => $this->next)); ?>">
    Next &gt;
  </a>
<?php else: ?>
  <span class="disabled">Next &gt;</span>
<?php endif; ?>
</div>
<?php endif; ?>
```

The new application should now work the index() actions of the SchoolController, PupilController and TeacherController each producing a full listing of the corresponding database records:

5.3.2 ZFSchools_2b: Some actual 'create, update' actions ...

The ZFSchools_2 project was duplicated to create a new version. (If the directory is copied within Netbeans, the file .zfproject.xml must be copied manually.) In this version, the SchoolController will have actions:

- List teachers at a chosen school;
- List pupils at a chosen school;
- Add a new school.

These actions will involve some form input, but this will be handled naively rather than with Zend's form components.

The IndexController.php script and views/scripts/index/index.phtml are the same as the previous version – simply a set of links to the index actions of the other controllers. The SchoolController now has four actions – index, add, listpupils, and listteachers.

```
<?php

class SchoolController extends Zend_Controller_Action {

    public function init() {
        /* Initialize action controller here */
    }

    public function indexAction() {
        $schoolTableAccess = new Application_Model_DbTable_SchoolTable();
        $schools = $schoolTableAccess->fetchAll();
        $this->view->schooldata = $schools;
    }

    public function addAction() {...}

    public function listpupilsAction() {...}

    public function listteachersAction() {...}

}
```

The index action gets the list of schools that is to be shown in the new version of the views/scripts/school/index.phtml based view:

The forms in this view are defined using fairly conventional HTML markup; the only unusual elements are the “action” attributes of the forms which now must reference actions of the controller. The url() function is used to generate these in the correct format.

```
<?php
$this->title = "Schools";
$this->headTitle($this->title);
?>
<table border="1">
    <tr>
        <th>School name</th>
        <th>Address</th>
    </tr>
    <?php foreach ($this->schooldata as $school) : ?>
        <tr>
            <td><?php echo $this->escape($school->SCHOOLNAME); ?></td>
            <td><?php echo $this->escape($school->ADDRESS); ?></td>
        <?php endforeach; ?>
</table>
<div id="tabs">
    <ul>
        <li><a href="#tab-1">Add School</a></li>
        <li><a href="#tab-2">List Pupils</a></li>
        <li><a href="#tab-3">List Teachers</a></li>
    </ul>
    <div id="tab-1">
        <form method="post"
              action=
              "<?php echo $this->url(
                 array('controller' => 'school', 'action' => 'add')); ?>" >
            <fieldset>
               <legend>School details</legend>
               <input type="text" name ="schoolname" size="64" maxlength="64" />
               <br/>
               <input type="text" name="address" size="128" maxlength="128" />
               <br/>
               <input type="submit" value="Add School">
            </fieldset>
        </form>
    </div>
    <div id="tab-2">
        <form method="post"
              action=
              "<?php echo $this->url(
                 array('controller' => 'school',
                 'action' => 'listpupils')); ?>" >
            <fieldset>
                <legend>School details</legend>
                <select name="schoolname">
                    <?php foreach ($this->schooldata as $school) : ?>
                        <option>
                             <?php echo
                                   $this->escape($school->SCHOOLNAME); ?>
                        </option>
                    <?php endforeach; ?>
                </select>
                <input type="submit" value="List Pupils">
            </fieldset>
        </form>
    </div>
    <div id="tab-3">
        ...
```

```
    </div>
</div>
```

In the generated form page, the links appear as in this fragment:

```
<div id="tab-1">

    <form method="post"
          action=
          "/~nabg/ZFSchools_2b/public/school/add" >
        <fieldset>
            <legend>School details</legend>
            <input type="text" name ="schoolname" size="64" maxlength="64" />
            <br/>
            <input type="text" name="address" size="128" maxlength="128" />
            <br/>
            <input type="submit" value="Add School">
        </fieldset>
    </form>
</div>
```

(These are “search engine friendly” links.)

The code for the add action in the SchoolController is:

```
    public function addAction() {
        $aschoolname = $this->getRequest()->getPost('schoolname');
        $anaddress = $this->getRequest()->getPost('address');
        $filterchain = new Zend_Filter();
        $filterchain->addFilter(new Zend_Filter_StripTags())->
                addFilter(new Zend_Filter_StringTrim());
        $aschoolname = $filterchain->filter($aschoolname);
        $anaddress = $filterchain->filter($anaddress);
        if ((strlen($aschoolname) < 6) || (strlen($anaddress) < 10)) {
            $this->view->novaliddata = true;
            $this->view->schoolname = $aschoolname;
            $this->view->address = $anaddress;
        } else {
            // Add the school
            $schoolTable = new Application_Model_DbTable_SchoolTable();
            $data = array('SCHOOLNAME' => $aschoolname,
                  'ADDRESS' => $anaddress);
            $schoolTable->insert($data);
            $this->view->schoolname = $aschoolname;
            $this->view->address = $anaddress;
        }
    }
```

The posted data, school name and address, are accessible via the “request” object associated with the controller. Obviously, some data checking is required (though presumably this application would be an in-house application not accessible via the Internet so that the level of checking is less). Here, rather than validate the data (what would a valid address look-like?), Zend “Filter” classes are used simply to clean up the data.

Zend Filter classes are similar in character to the validators illustrated in section 2. There are a number of predefined filter classes, and you can define your own. Here one would probably want to filter out any characters other than alphanumerics, white space, and a few punctuation characters that might be valid in an address (commas etc). There isn't a predefined filter class for this; so

instead, any HTML or PHP tags in the input data are stripped out using the “Strip tags” filter, and then any trailing spaces are trimmed off. A new record can be added using the table access class – Application_Model_DbTable_SchoolTable – much as previously illustrated in section 3.

The final report – either noting the creation of a new record or complaining about the inputs – is generated via the associated view/scripts/school/add.phtml template:

```
<h1>Add new school</h1>
<?php if (isset($this->novaliddata)): ?>
<p>There was something wrong with either the school name or the address that
you entered.
<?php else: ?>
  <p>A new school record has been created for
    <?php echo $this->schoolname; ?>,
    <?php echo $this->address; ?>
  </p>
<?php endif; ?>
```

The SchoolController actions for listpupils() and listteacher() are reworkings of the code illustrated in the previous section (where they were the index actions of the PupilController and TeacherController). There are a couple of complications.

The lists of pupils or teachers are to be school specific; so the “select” object is extended with a “where” element using the school name as a constraint. The initial invocation of an action like listpupils will be a “POST” from the form on the SchoolController's index.phtml page (so the school name would accessible via the request->getPost() method as illustrated above). But there are many pupils to list for a given school, so a paginator will be needed. This will use <a href=...> links to access previous, next, and other pages. These links, which are equivalent to GET requests, are going to have to supply both school name and page number. Further, the links are going to use the “search engine friendly” style – the arguments are not going to be there as a query string, instead they are going to form part of a pseudo file-path, e.g. /~nabg/ZFSchools_2b/public/school/listpupils/page/2/schoolname/Huntely+Vale+Primary

```
<div class="paginationControl">
<!-- Previous page link -->
  <a href="/~nabg/ZFSchools_2b/public/school/listpupils/page/2/schoolname/Huntley+Vale+Primary">
    &lt; Previous
  </a> |

<!-- Numbered page links -->
      <a href="/~nabg/ZFSchools_2b/public/school/listpupils/page/1/schoolname/Huntley+Vale+Primary">
        1    </a> |
        <a href="/~nabg/ZFSchools_2b/public/school/listpupils/page/2/schoolname/Huntley+Vale+Primary">
        2    </a> |
        <a href="/~nabg/ZFSchools_2b/public/school/listpupils/page/3/schoolname/Huntley+Vale+Primary">
        3    </a> |
        <a href="/~nabg/ZFSchools_2b/public/school/listpupils/page/4/schoolname/Huntley+Vale+Primary">
        4    </a> |
        <a href="/~nabg/ZFSchools_2b/public/school/listpupils/page/5/schoolname/Huntley+Vale+Primary">
        5    </a> |
        <a href="/~nabg/ZFSchools_2b/public/school/listpupils/page/6/schoolname/Huntley+Vale+Primary">
        6    </a> |
        <a href="/~nabg/ZFSchools_2b/public/school/listpupils/page/7/schoolname/Huntley+Vale+Primary">
        7    </a> |

<!-- Next page link -->
  <a href="/~nabg/ZFSchools_2b/public/school/listpupils/page/4/schoolname/Huntley+Vale+Primary">
    Next &gt;
  </a>
</div>
```

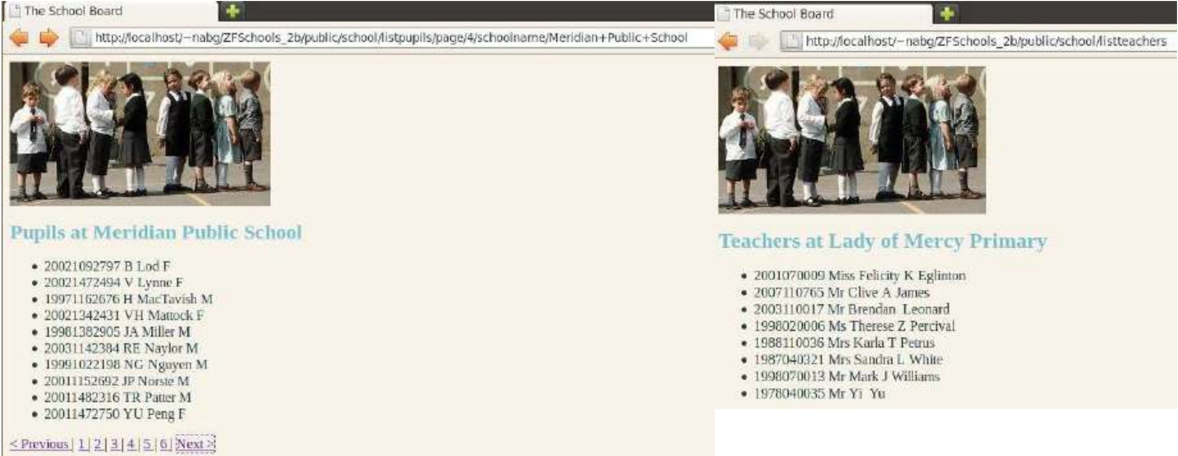
The Zend framework will sort out the data; the data (whether POSTed, GETed, or assembled into a pseudo filepath) can be accessed using the _getParam() method of the controller as in the code below. (The second argument in _getParam is a default that can be used if no parameter was supplied.) The other extra features of the code are the "where" component in the select, and the adding of the school name to the data array that will be used by the paginator when generating links:

```
  public function listpupilsAction() {
        $aschoolname = $this->_getParam('schoolname');
        $page = $this->_getParam('page', 1);
        $pupilTableAccess = new Application_Model_DbTable_PupilTable();

        $select = $pupilTableAccess->select();
        $select->from('PUPIL', array('ENROLNUMBER', INITIALS',
            'SURNAME', 'GENDER'))->
                where('SCHOOLNAME=:school')->
                bind(array('school' => $aschoolname))->
                order(array('SURNAME'));
        $paginator = Zend_Paginator::factory($select);

        $paginator->setCurrentPageNumber($page);
        $paginator->setItemCountPerPage(10);

        $thepages = $paginator->getPages();
        $pagerange = $thepages->pagesInRange;
        $last = 1 + (int) ($paginator->getTotalItemCount() / 10);
        $this->view->paginator = $paginator;
        $paginatordata = array();

        if ($page > 1)
            $paginatordata['previous'] = $page - 1;
        if ($page < $last)
            $paginatordata['next'] = $page + 1;
        $paginatordata['pagesInRange'] = $pagerange;
        $paginatordata['pageCount'] = $last;
        $paginatordata['schoolname'] = $aschoolname;

        $this->view->paginatordata = $paginatordata;
        $this->view->schoolname = $aschoolname;
    }
```

The code for the views/scripts/partialsscripts/paginatorcontrol.phtml file had to be extended to add the school name to generated links:

```
<div class="paginationControl">
<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?php echo $this->url(array('page' => $this->previous,
        'schoolname' => $this->schoolname)); ?>">
    &lt; Previous
  </a> |
<?php else: ?>
  <span class="disabled">&lt; Previous</span> |
<?php endif; ?>
```

The code for listing teachers for a school is similar in style.



The processing of Teacher records follows the same approach as in section 3.1.4. The index action for the TeacherConroller will display a page with tabbed options (jQuery UI style tabs) for record creation, view, update, and delete. The create form is similar to that shown earlier; its form's action attribute references the “create” action of the TeacherController:

```
class TeacherController extends Zend_Controller_Action {

    public function init() { }

    public function indexAction() { … }

    public function createAction() {
        $fname = $this->getRequest()->getPost('firstname');
        $initials = $this->getRequest()->getPost('initials');
        $surname = $this->getRequest()->getPost('surname');
        $title = $this->getRequest()->getPost('title');
        $schoolname = $this->getRequest()->getPost('schoolname');
        $role = $this->getRequest()->getPost('rolename');
        $empnum = $this->getRequest()->getPost('empnum');

        $errormsgs = array();
        // Obviously need to check the input data - i.e. use of
        // Zend_Validate and Zend_Filter possibly with own
        // validator classes as well.
        // Here limited to a couple of checks just to illustrate
        // how to get Zend validators into a controller.
        // The first name and surname must both be alpha only - tough
        // on persons with names like O'Toole or Lloyd-Davis,
        // The employee number must be lots of digits
        $filterchain = new Zend_Filter();
        $filterchain->addFilter(new Zend_Filter_StripTags())->
                addFilter(new Zend_Filter_StringTrim());

        $validatorChain1 = new Zend_Validate();
        $validatorChain1->addValidator(
                  new Zend_Validate_StringLength(array('min' => 1, 'max' => 16)))
                      ->addValidator(new Zend_Validate_Alpha());
        $digitChecker = new Zend_Validate_Digits();

        $fname = $filterchain->filter($fname);
        $surname = $filterchain->filter($surname);
        $initials = $filterchain->filter($initials);

        if (!$validatorChain1->isValid($fname)) {
```

```
            $errormsgs[] = "<li>The given first-name was invalid</li>";
        }
        if (!$validatorChain1->isValid($surname)) {
            $errormsgs[] = "<li>The given surname was invalid</li>";
        }
        if (!$digitChecker->isValid($empnum)) {
            $errormsgs[] = "<li>The employee number was invalid</li>";
        }
        $roles = array('PRINCIPAL', 'K', '1', '2', '3', '4', '5', '6');
        $titles = array('Dr', 'Mr', 'Mrs', 'Miss', 'Ms');

        if (!in_array($title, $titles)) {
            $errormsgs[] = "<li>Invalid title</li>";
        }
        if (!in_array($role, $roles)) {
            $errormsgs[] = "<li>Invalid role</li>";
        }

        $schoolname = $filterchain->filter($schoolname);


        $dbvalidator = new Zend_Validate_Db_RecordExists(
                        array(
                            'table' => 'SCHOOL',
                            'field' => 'SCHOOLNAME',
                ));
        if (!$dbvalidator->isValid($schoolname)) {
            $errormsgs[] = "<li>Unknown school</li>";
        }

        if (count($errormsgs) > 0) {
            $this->view->errors = $errormsgs;
        } else {

            $data = array(
                'EMPLOYEENUMBER' => $empnum,
                'SURNAME' => $surname,
                'INITIALS' => $initials,
                'FIRSTNAME' => $fname,
                'TITLE' => $title,
                'SCHOOLNAME' => $schoolname,
                'ROLE' => $role
            );
            $teacherTableAccess = new Application_Model_DbTable_TeacherTable();
            $teacherTableAccess->insert($data);

        }
    }

    public function modifyAction() { }

}
```

The data for the new record are posted from the form and so can be accessed via the getPost(param) method of the request object associated with the controller. Zend_Validate and Zend_Filter components are used to apply some data checks – this includes use of Zend_Validate_Db_RecordExists to verify that the school name given is that of a known school. (Yes, the form offered only valid school names as options – but you can never trust any input from forms.) If all data pass the validation tests, a new record is created.

The associated view (views/scripts/teacher/create.phtml) simply acknowledges successful record

creation, or lists problems found with the data:

```
<h1>Creating new teacher employment record</h1>
<?php if (isset($this->errors)) : ?>
    <p>The operation failed because:</p><ul>
    <?php foreach ($this->errors as $anerror) {
        echo $anerror;
    } ?>
    </ul>
<?php else : ?>
        <p>New teacher record successfully created.</p>
<?php endif; ?>
```

The “Read” operation for the Teacher data should, as in the earlier example, display the data for a specifically identified teacher (identified by employee number); the data should be read-only, but there should be links back to update and delete operations. The “Update” operation should display these same data with all editable apart from the employee number; there should be a link that causes the update to be performed. The “Delete” operation should display the data read-only, with a link that causes the deletion to be performed.

If the application were developed from start with the framework, it would be logical to define “read”, “update”, and “delete” actions for the TeacherController each with their own view scripts. The actions and their views would share some code – as separate private member functions in the TeacherController class and partial scripts for the views. But it is possible to simply adapt the design used in the earlier version (section 3.1.4). A single modify action and associated view will handle all three cases.

The modify view (views/scripts/teacher/modify) can display messages (“*invalid employee number*”, “*record updated*”, “*record deleted*” etc) or will show a teacher record in a form. Form fields are normal or are read-only as required (and specified by other arguments passed to the view); actions for the form are defined appropriately depending on whether this is a read, update, or delete.

The view script is:

```
<h1>Teacher Record</h1>
<?php if (isset($this->messages)) : ?>
<p><?php echo $this->messages; ?></p>
<?php else : ?>
```

```
    <table border='1' align='center'>
    <?php if ($this->command != 'Read') : ?>
       <form method='POST'
           action="<?php echo $this->url(
                 array('controller' => 'teacher', 'action' => 'modify')); ?>" >
           <input type='hidden' name='command'
                 value='<?php echo $this->command; ?>' >
           <input type='hidden' name='empnum'
                 value='<?php echo $this->empnum; ?>' >
    <?php else : ?>
        <form>
     <?php endif; ?>
         <tr>
             <th>Employee number</th>
             <td><?php echo $this->teacher->EMPLOYEENUMBER ?></td>
         </tr>
         <tr>
             <th>First name</th>
             <td>
             <input type='text' <?php echo $this->style1 ?>
                 name='firstname'
                 value='<?php echo $this->teacher->FIRSTNAME ?>'
                 size='16' maxlength='16' />
              </td>
         </tr>
         <tr>
             <th>Initials</th>
             ...
         </tr>
         <tr>
              <th>Surname</th>
              ...
         </tr>
         <tr>
             <th>Title</th>
                   <td>
                       <select size='1' <?php echo $this->style2 ?>
                       name='title' >
                       <?php
                       foreach ($this->titles as $atitle) {
                           if ($atitle == $this->teacher->TITLE)
                               echo "<option selected>";
                           else
                               echo "<option>";
                           echo $atitle . "</option>";
                       }
                       ?>
                   </select>
               </td>
           </tr>
           <tr>
               <th>Role</th>
               ...
           </tr>
           <tr>
               <th>School</th>
               ...
           </tr>
           …
  </form>
</table>
<?php endif; ?>
```

The modify action in TeacherController has sections for "GET" and "POST". A "GET" request will have an employee number and a sub-command (read, update, or delete) as parameters. A "POST" for an update will have new data values.

A "GET" request is handled by loading the specified teacher record from the database. If an invalid employee number was supplied, the load will result in an empty record – a message should be displayed via the view and the action terminated. If the record was successfully loaded, it should be forwarded to the associated view along with other display options:

```
class TeacherController extends Zend_Controller_Action {

    public function init() { }

    public function indexAction() { … }

    public function createAction() {

    public function modifyAction() {
        if ($this->getRequest()->getMethod() == "GET") {
            // Should have an employee number and a command - Read Update Delete
            // In all cases display the record if found
            $empnum = $this->getRequest()->getParam('empnum');
            $command = $this->getRequest()->getParam('command');

            $schoolTableAccess = new Application_Model_DbTable_SchoolTable();

            $schools = $schoolTableAccess->fetchAll();

            $teacherTableAccess = new Application_Model_DbTable_TeacherTable();


            $ateacher = $teacherTableAccess->find($empnum); // rowset
            $ateacher = $ateacher->current(); // a row
            // If given a bad employee number (not in table) will get back an
            // empty teacher record
            if (is_null($ateacher->EMPLOYEENUMBER)) {
                $this->view->messages =
                  "You appear to have entered an invalid employee number.";
                return;
            }

            $roles = array('PRINCIPAL', 'K', '1', '2', '3', '4', '5', '6');
            $titles = array('Dr', 'Mr', 'Mrs', 'Ms', 'Miss');
            if ($command == "Update") {
                $style1 = "";
                $style2 = "";
            } else {
                $style1 = "readonly";
                $style2 = "disabled";
            }

            $this->view->command = $command;
            $this->view->style1 = $style1;
            $this->view->style2 = $style2;
            $this->view->teacher = $ateacher;
            $this->view->empnum = $empnum;
            $this->view->schools = $schools;
            $this->view->roles = $roles;
            $this->view->titles = $titles;
        } else {
```

```
            // Should be a POST -
            // update - replace data in record
            // delete - do the actual deletion
            $command = $this->getRequest()->getPost('command');
            $empnum = $this->getRequest()->getPost('empnum');
            $teacherTableAccess = new Application_Model_DbTable_TeacherTable();
            $selector = "EMPLOYEENUMBER=$empnum"; // Damn the SQL inject

            if ($command == "Update") {
                // Obviously should validate the new data - refactor those
                // checks that are currently in "create" action
                // but laziness prevails - no-one will ever hack this site.
                $data = array(
                    "SURNAME" => $this->getRequest()->getPost("surname"),
                    "FIRSTNAME" => $this->getRequest()->getPost("firstname"),
                    "INITIALS" => $this->getRequest()->getPost("initials"),
                    "TITLE" => $this->getRequest()->getPost("title"),
                    "ROLE" => $this->getRequest()->getPost("role"),
                    "SCHOOLNAME" => $this->getRequest()->getPost("schoolname")
                );
                $teacherTableAccess->update($data,$selector);
                $message = "Employee record updated";
            }
            else {
                // should be delete
                $teacherTableAccess->delete($selector);
                $message = "Employee record deleted";
            }
            $this->view->messages = $message;
        }
    }
}
```

5.4 ApplicationForm_II : Zend Form

One can build data entry forms the naïve way with just HTML markup being echoed by the view script; and one can write the necessary validation code that checks submitted data. But such code is long-winded, repetitious, boring, and consequently tends to be rather error prone. Weren't computers supposed to save us from boring, repetitious work?

Enter on cue: Zend_Form.

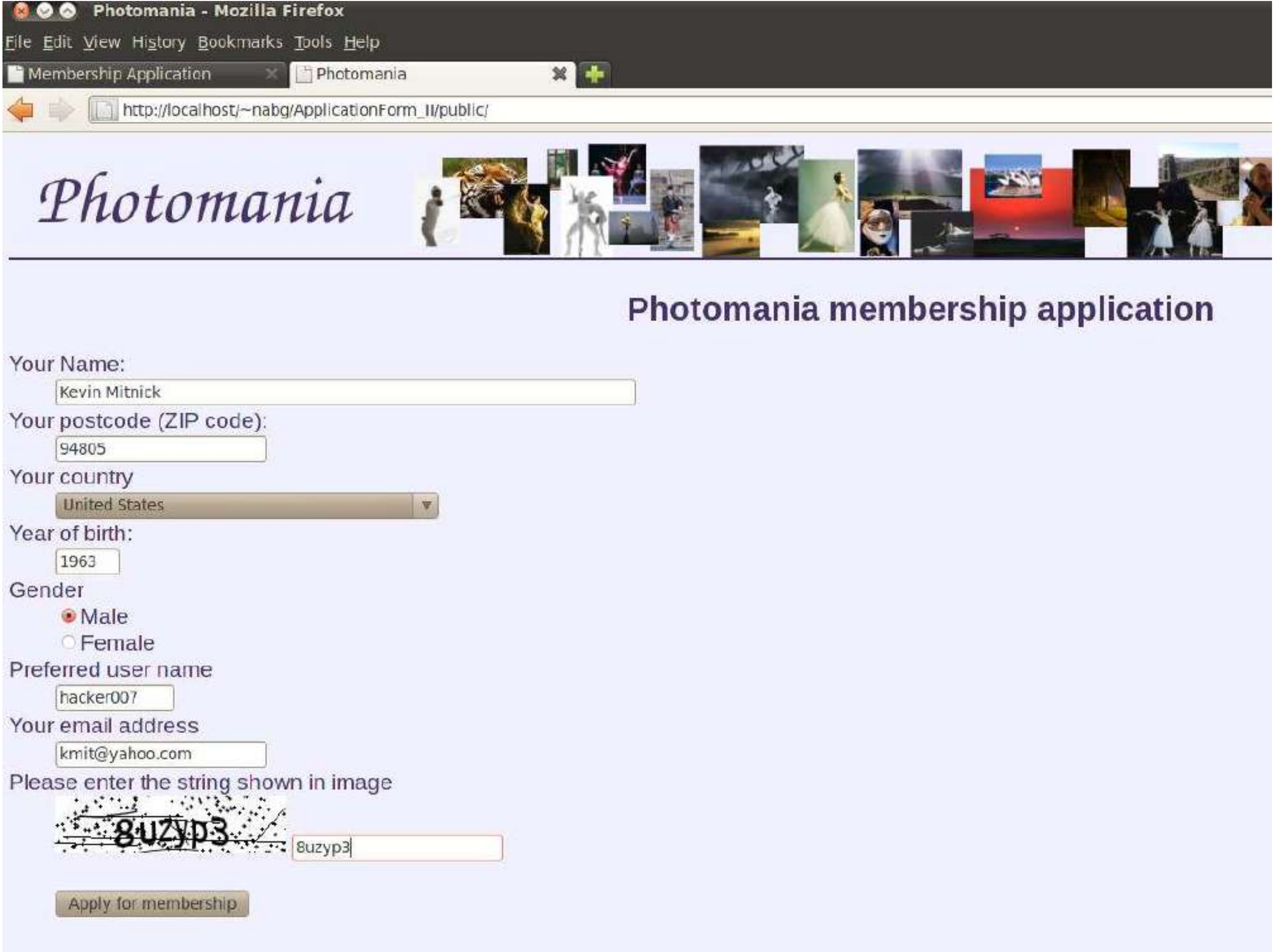
Application specific forms are defined as classes that extend the supplied Zend_Form class. The definition will specify the fields – each field being an instance of one of a set of supplied Zend_Form_Element classes. An example element is Zend_Form_Element_Text – this will get rendered as an <input type= 'text' … /> in the HTML that is eventually generated. When such a text element is added in the form definition, one can supply additional data – e.g. attributes like size and maxlength. One can also specify filters – these will be realised using instances of Zend supplied filter classes or application defined filters. Finally, one can add validators; these will identify the class of the validator and supply any additional parameters.

A controller will be used to create an instance of the form. Typically, a "GET" request to the controller results in display of the form; data are processed when "POST"ed back. Checking is simply a matter of asking the form to validate the posted data! If data fail validation tests, the form can be redisplayed with the user submitted data – any invalid elements being tagged with error comments.

Form display is handled through the view associated with the controller. The view renders the form using Zend supplied code. This defaults to using a HTML definition list to format the display.

Each form element has a label – this becomes a <dt> element in the definition list. The HTML markup for the data entry element becomes the <dd> part of the list.

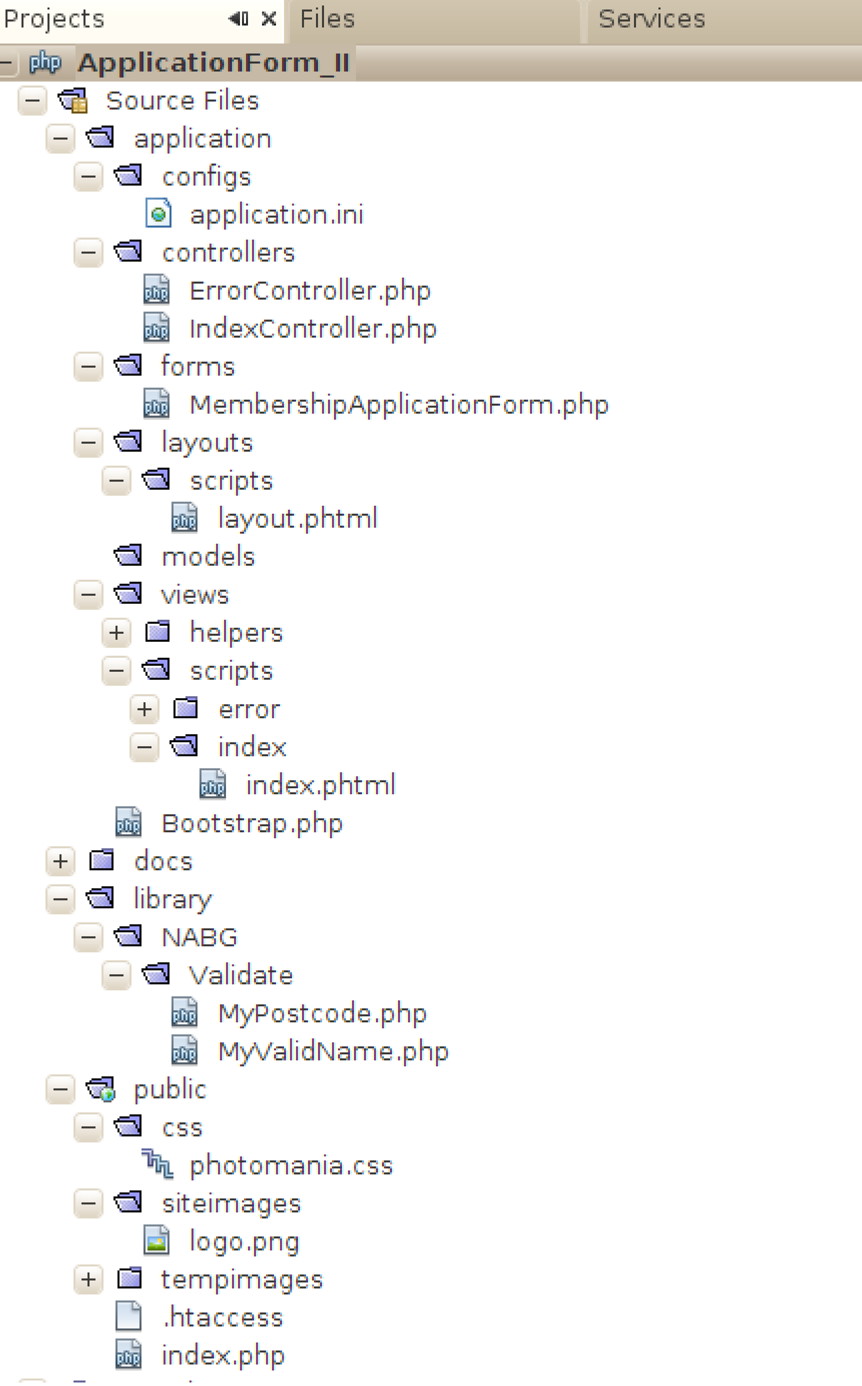
5.4.1 Membership form revisted



This is a reworking of the “membership form” example used in section 2.

This application has:

- An IndexController and associated views/scripts/index/index.phtml view file;
- A form class – MembershipApplicationForm.php – in the forms subdirectory of the application directory;
- A layout file; this references an image in the public/siteimages directory;
- A tempimages directory (that must have write permission for the process running the web server); this is where captcha images are placed;
- A css file (in public/css);
- A couple of specialised Validator classes; these have to be named and located in accord with Zend's conventions for its auto-loader; the classes are NABG_Validate_MyValidName and NABG_Validate_MyPostCode; they have to be placed in the library folder with sub-directories providing an access path that corresponds to the class names – NABG/Validate;
- a .htaccess file with the RewriteBase set correctly.



As in the example in section 2, the application will have to check that the name selected for use on the system has not already been picked. This will involve a check on the data table that contains member names. Consequently, the application needs to be configured with a database adapter. Honestly, it's easier to do this by manually editing the application.ini file rather than trying to enter the dsn correctly in the dialog for the zend.sh create db-adapter.

```
resources.frontController.params.displayExceptions = 1
resources.layout.layoutPath = APPLICATION_PATH "/layouts/script
resources.db.adapter = PDO_MYSQL
resources.db.params.host         = localhost
resources.db.params.username     = mark
resources.db.params.password     = nothispassordeither
resources.db.params.dbname       = mark
```

The form will be displayed by the IndexController for a “GET” request. For a “POST” request, the controller will validate the data submitted. If the data appear valid, an acknowledgement will be displayed (this version neither saves the data to file nor sends mail to the site administrator). If some data are invalid the form is redisplayed with data and error messages.

The controller creates the instance of the form, and attaches it to the associated view:

```
class IndexController extends Zend_Controller_Action
{

    public function init()
    {
        /* Initialize action controller here */
    }

    public function indexAction()
    {
        $myform = new Application_Form_MembershipApplicationForm();
        $this->view->form = $myform;
        if($this->getRequest()->isPost()) {
            $inputdata = $this->getRequest()->getPost();
            if($myform->isValid($inputdata)) {
                $messages = "Thank your for filling in the form";
                $this->view->messages = $messages;
            }
            else {
                $myform->populate($inputdata);
            }

        }
        // If a GET request, just show the form.
    }

 }
```

Validation using an instance of a form class is much easier than doing it manually. It is simply a matter of grabbing the posted data and asking the form to handle the validation steps! If the data failed validation, they are put back into the form before it is redisplayed – error comments are added automatically.

The view and layout are both simple:

```
<?php

echo "<div id='appform'>";
echo "<h1>Photomania membership application</h1>";
if ($this->messages) {
    echo $this->messages;
} else {

    echo $this->form;
}
echo "</div>";
```

```
<?php
$this->headMeta()->appendHttpEquiv('Content-Type', 'text/html;charset=utf-8');
$this->headTitle()->setSeparator(' - ');
$this->headTitle('Photomania');
```

```
echo $this->doctype();
?>
<html >
    <head>
        <?php
        echo $this->headMeta(),
        $this->headTitle(),
        $this->headLink()->appendStylesheet(
                $this->baseUrl() . '/css/photomania.css');
        ?>
    </head>
    <body>
        <div id='headdiv'>
        <img src="<?php echo $this->baseUrl() . '/siteimages/logo.png'; ?>" />
        </div>
        <h1><?php echo $this->headTitle(); ?></h1>
        <div id="content">
            <?php echo $this->layout()->content; ?>
        </div>
    </body>
</html>
```

The form class should be created using the appropriate zend.sh options. The structure of the form is defined in the init() function; it is simply a matter of creating form elements and setting their attributes and finally adding them all to the form.

There are many standard validators supplied in the framework. But this application required two more specialised validators. Firstly, names are to be checked – using the name checking code illustrated in section 2. Secondly, the post code entered is required to be consistent with the country selected; but this requires checking on combinations of inputs from the same form, and so is not quite standard. Two specialised validator classes were defined and placed in the library for this application. The form has to be configured so that it checks for validator classes in this application library as well as in the standard framework libraries.

```
<?php
  2
  3 class Application_Form_MembershipApplicationForm extends Zend_Form {
  4
  5     public function init() {
  6         $this->addElementPrefixPath(
                  'NABG_Validate', 'NABG/Validate/', 'validate');
  7         $countries = array(
  8             'AF' => 'Afghanistan',
  9             'AL' => 'Albania',
                  ...
270             'ZW' => 'Zimbabwe'
271         );
272
273         $genders = array('Male', 'Female');
274         $this->setMethod('post');
275
276
```

Each form element is defined and then attributes, filters, and validators are specified. The username field is for the applicant's real name. It is a text input field with given size and maximum length. A value must be supplied. Zend Filter classes are used to remove any attempts at cross-site scripting via injected Javascript tags etc. The validator class is MyValidName – i.e. the class NABG_Validate_MyValidName (the path combing from the prefix path defined above).

```
277         $username = new Zend_Form_Element_Text('username');
278         $username->setLabel('Your Name:')
279                 ->setAttrib('size', 60)
280                 ->setAttrib('maxlength', 60)
281                 ->setRequired(true)
282                 ->addFilter('StripTags')
283                 ->addFilter('StringTrim')
284                 ->addValidator('MyValidName');
```

The next input element is for the user's post code. Again this is an <input type='text' … /> element with defined size and maximum length. The validator is another application specific validation class.

```
285         $postcode = new Zend_Form_Element_Text('pcode');
286         $postcode->setLabel('Your postcode (ZIP code):')
287                 ->setAttrib('size', 20)
288                 ->setAttrib('maxlength', 20)
289                 ->setRequired(true)
290                 ->addFilter('StripTags')
291                 ->addFilter('StringTrim')
292                 ->addValidator('MyPostcode');
```

The next input element is a <select> that supports only single selection. The options are supplied as an array; this is the array of 'country code' => 'country name' data defined above. The Zend framework automatically supplies a validator that will verify that the submitted country code is one of the supplied values.

```
293         $country = new Zend_Form_Element_Select('country');
294         $country->setLabel('Your country');
295         $country->addMultiOptions($countries);
296
```

The input element for the 'year of birth' can be defined using solely stock Zend components. It is a <input type='text' .../> element with defined size; the filters eliminate non-digits; an instance of Zend_Validate_Between can be used to check whether the year is within a plausible range. (The false argument specifies that the checking should NOT stop if this test fails; other checks should also be done.)

```
297         $yearofbirth = new Zend_Form_Element_Text('year');
298         $yearofbirth->setLabel('Year of birth:')
299                 ->setAttrib('size', 4)
300                 ->setAttrib('maxlength', 4)
301                 ->setRequired(true)
302                 ->addFilter('Digits')
303                 ->addValidator('Between', false,
                            array('min' => 1910, 'max' => 2000));
304
305         $gender = new Zend_Form_Element_Radio('gender');
306
307         $gender->setLabel("Gender")->setRequired(true);
308         $gender->addMultiOption("male", "Male");
309         $gender->addMultiOption("female", "Female");
310         // TO set male as checked in radio cluster -
            // set that as value of element
311         $gender->setValue("male");
312
```

The uname field is for the name by which the member would wish to be known on site. This needs two validators – is it an alphanumeric string of suitable length, and is it new (i.e. not already in the members table).

```
313         $uname = new Zend_Form_Element_Text('uname');
314         $uname->setLabel("Preferred user name")
315                 ->setAttrib('size', 10)->setAttrib('maxlength', 10)
316                 ->addFilter('Alnum')->
317                 addValidator('StringLength', false,
318                     array('min' => 6,'max' => 10))
319                 ->addValidator('Db_NoRecordExists', false,
320                         array('table' => 'gallerymembers',
321                         'field' => 'username'));
322
```

The email element uses the Zend_Validate_EmailAddress validator.

```
323         $email = new Zend_Form_Element_Text('email');
324         $email->setLabel("Your email address")->setAttrib('size', 20)->
325                 addValidator('EmailAddress', false,
326                         array('allow' => Zend_Validate_Hostname::ALLOW_DNS,
                         'mx' => true));
327
328         $submit = $this->createElement('submit', 'submit');
329         $submit->setLabel("Apply for membership");
330
```

The captcha element requires a lot of configuration data such as the correct URL to use when accessing the image, the location of fonts, and the directory where captcha images should be stored.

```
331         $captcha = new Zend_Form_Element_Captcha('zombie',
332                 array(
333                     'label' => "Please enter the string shown in image",
334                     'captcha' => array(
335                         'captcha' => 'Image',
336                         'font' =>
                 '/usr/share/fonts/truetype/ubuntu-font-family/Ubuntu-R.ttf',
337                         'timeOut' => 200,
338                         'wordLen' => 6,
339                         'imgDir' =>
                      APPLICATION_PATH.'/../public/tempimages',
340                         'imgUrl' =>
         Zend_Controller_Front::getInstance()->getBaseUrl().'/tempimages'
341                     )
342                 ));
343
```

When all the elements have been created and configured, they can be added to the form.

```
344         $this->addElements(array(
345             $username,
346             $postcode,
347             $country,
348             $yearofbirth,
349             $gender,
350             $uname,
351             $email,
352             $captcha,
353             $submit
354         ));
355     }
```

```
356
357 }
358
```

The class NABG_Validate_MyValidName is just a renamed version of the class defined in section 2. The Zend_Validate_PostCode validator requires the “locale” for the putative post code. Sometimes this can be defined by the application – if you are serving only local customers you could have it default to your locale. But here, the locale has to correspond to the country identified by the applicant. So there has to be some code that picks up the country from the form input, creates the locale, and then performs the validation. This requires a specialised sub-class of Zend_Validate_Abstract. The isValid() method that must be implemented takes two arguments – one is the value being checked, and the other “context” is an array with other input data. So, here one can get the country from the $other array, create the locale for the country, and then use the standard PostCode validator. (This code's error message handling is kind of simplified.)

```
 * @author nabg
 */
class NABG_Validate_MyPostcode extends Zend_Validate_Abstract {
    const NOT_MATCH = 'notMatch';

    protected $_messageTemplates = array(
        self::NOT_MATCH => 'Post code not valid for your country'
    );

    public function isValid($value, $context = null) {
        $value = (string) $value;
        $this->_setValue($value);

        if (is_array($context)) {
            if (isset($context['country'])) {
                $country = $context['country'];
                $locale = new Zend_Locale($country);
                $pcodeValidator = new Zend_Validate_PostCode($locale);
                if(!$pcodeValidator->isValid($value)) {
        $this->_error(self::NOT_MATCH);
        return false;
                }
                return true;
            }
        }

        $this->_error(self::NOT_MATCH);
        return false;
    }
}
?>
```

What happens if data are invalid?

You get a chance to re-enter your data with your mistakes highlighted:

Photomania

Photomania membership applicat

Your Name:

mitnick

- **That name doesn't resemble typical name pattern**

Your postcode (ZIP code):

CA94805

- **Post code not valid for your country**

Your country

United States

Year of birth:

1903

- **'1903' is not between '1910' and '2000', inclusively**

Gender

Male

Female

Preferred user name

007

- **'007' is less than 6 characters long**

Your email address

fake

- **'fake' is no valid email address in the basic format local-part@hostname**

Please enter the string shown in image

c5va4y

- **Captcha value is wrong**

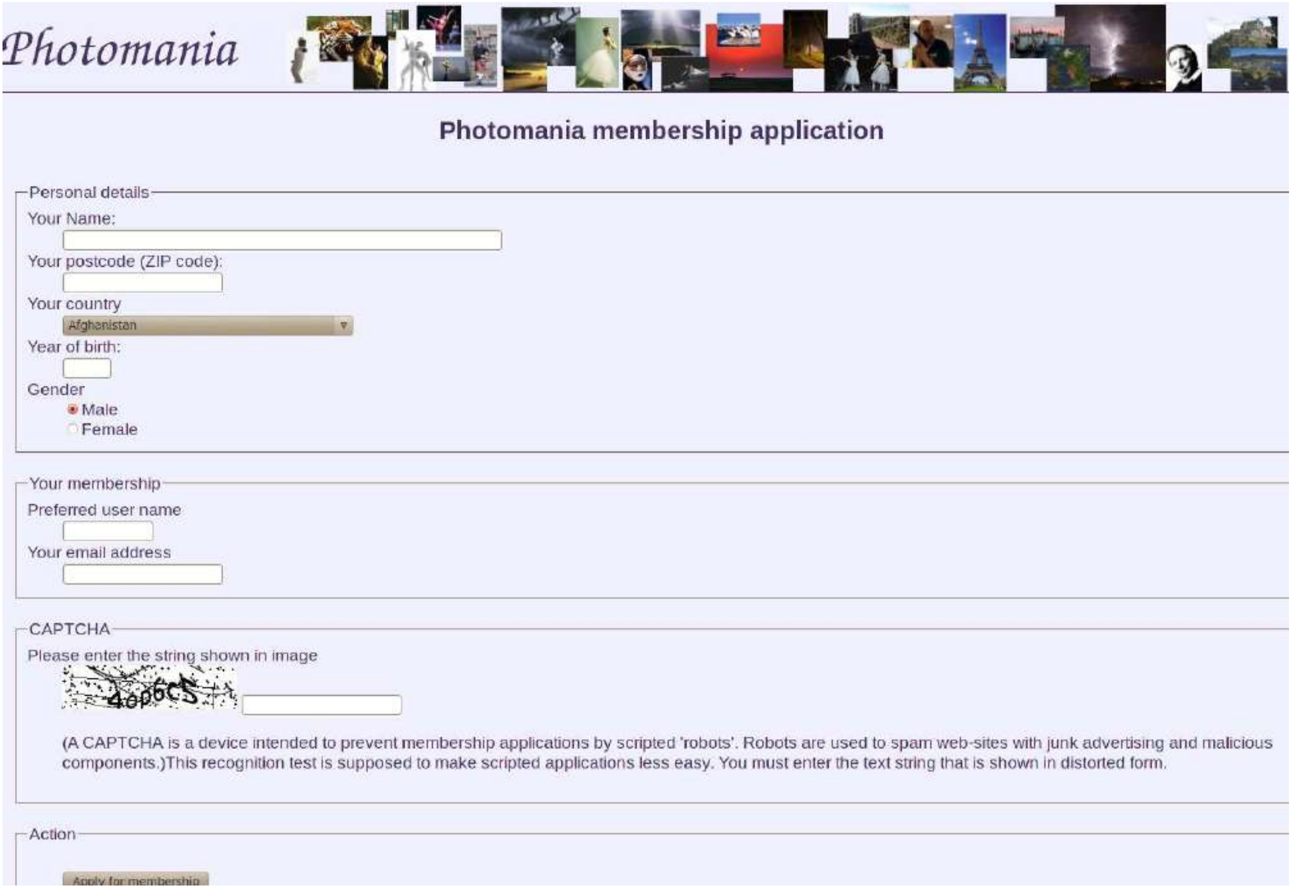
Apply for membership

One tends to expect that values from <select> inputs with defined options will be correct – but that isn't true in the presence of hackers. As mentioned above, the Zend Form framework automatically adds “in array” validators for such inputs.

If you want to test your site against malicious inputs, then you may find the “Tamper Data” plugin for Firefox to be useful (there are equivalent plugins for IE). Such tools allow a tester (or hacker) to adjust inputs prior to actual submission:

(Adjusting the country to an invalid code, e.g. QQ, causes this application to fail – the failure occurs when a locale can not be found for the imaginary country code.) The odd error message about a haystack is just the way that the “in array” validator reports that it cannot find the input in the set of allowed values. (It is possible to customise error messages to some degree.)

5.4.2 Form not pretty enough? Add decorators



It's relatively easy to add some <fieldset> sections. One just needs to define the groups of elements that belong together and provide a value for the <legend> element of the final fieldset – the standard Zend code for dealing with forms will generate the required more elaborate HTML. Similarly, you can add “descriptions” to elements – these appear as additional text.

```
// Decorations!
$captcha->setDescription(
        "(A CAPTCHA is a device intended to prevent membership applications by scrip
        " Robots are used to spam web-sites with junk advertising and malicious compo
        "This recognition test is supposed to make scripted applications less easy. '
            "You must enter the text string that is shown in distorted form.");
$this->addDisplayGroup(array(
    'username',
    'pcode',
    'country',
    'year',
    'gender'
),'personal', array('legend' => 'Personal details'));
$this->addDisplayGroup(array('uname', 'email'),
       'membership', array('legend' => 'Your membership') );
$this->addDisplayGroup(array('zombie'),'captcha', array('legend' => 'CAPTCHA'));
$this->addDisplayGroup(array('submit'), 'action', array('legend' => 'Action'));
```

Much more elaborate changes are possible. For example, it is possible to change from the default "definition list" format used for a form to a table.

However, decorations soon become quite complex and confusing. They are best left to professional decorators – there are a few out on the web including:

- http://devzone.zend.com/article/3450
- http://codeutopia.net/blog/2008/08/07/zend_form-decorator-tips/
- http://zendgeek.blogspot.com/2009/07/applying-zendform-decorators-to-all.html

5.5 Authentication and Access

What about a site that has several different classes of user, each class having different privileges with respect to site functionality?

The putative "Photomania" site can serve as an example. This site is to have:

- A single "owner"
  The owner can do anything on the site. More particularly, the owner alone can use the scripts that will change the status of other users in a "members" table.
- Contributors – hopefully quite a few.
  Contributors can upload photos to the site. Each contributor will have his/her own "gallery" (just a subdirectory) in which pictures are stored. Contributors can run the scripts that upload the photos etc.
- Members – hopefully many.
  Members can run the scripts that add comments and tags to pictures. (There is no predefined scheme for changing status from member to contributor; it's up to the owner to decide how that transition might be effected.)
- "Applicants and Suspended"
  These are two additional levels for entries in the members table. Applicants have completed the application form but their requests have still to be processed by the owner. The owner may "suspend" membership if a member misbehaves, e.g. by posting spamming messages. Persons in these categories simply have routine "guest" access.
- Guests – i.e. general Internet users
  Anyone on the Internet may view the galleries and if they wish apply for membership. They can also try logging in if they are members and have received passwords.

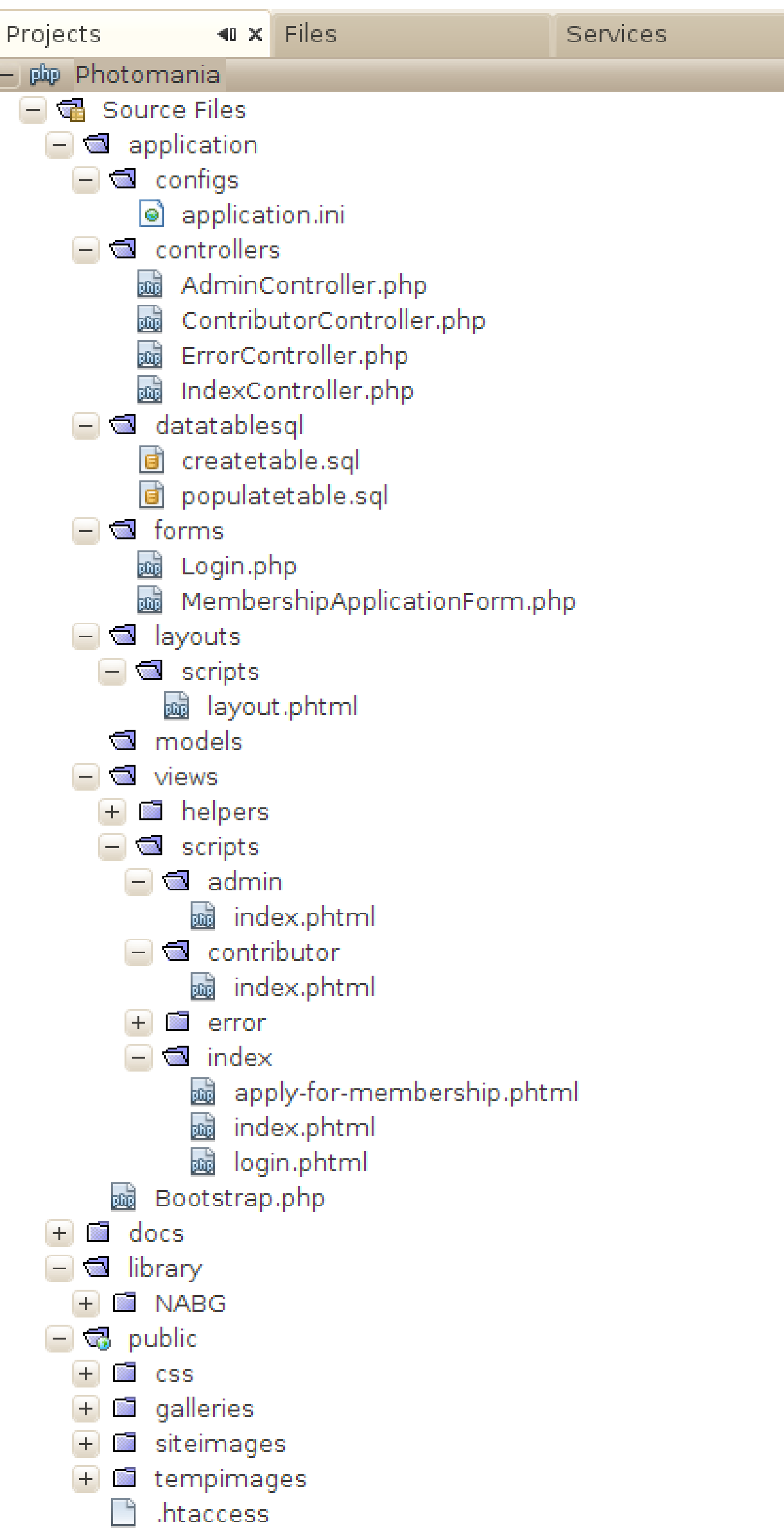
Authentication will be handled using conventional (name, password) login. (Little details, like what to do if a user forgets a password, will be ignored in this example.) Once logged in, a user will get a "role" based on their entry in the members table. This role will determine which actions they may subsequently access.

5.5.1 Log me in: Zend_Auth

The Zend_Auth class, a singleton class, works with a chosen "adapter"; this adapter scheme allows an application to select among different authentication mechanisms. There are several mechanisms implemented through other classes in the Zend_Auth library. One uses standard HTTP authorisation where user names and passwords would be managed by the web-server, another uses a data base, still other variants can work with directory services like LDAP. The data base variant is probably the one most commonly used.

With the data base variant, Zend_Auth + Zend_Auth_Adapter_DbTable, user names and passwords are stored in an application defined database table, along with such other data as the application requires. These data will almost always include a specified “role”, and can have other data (such as data for personalising displayed pages). The Zend code can be used to check data supplied in a log-in form. If the user-name and password combination is valid, a record will be created in $_SESSION; this record can be checked in subsequent scripts to confirm logged in status. The application can chose what other data from the database table are stored as part of this record.

The Photomania project has the code illustrating the use of Zend_Auth:



The application has four controllers – Index, Admin, Contributor, and the standard Error controller. The Admin and Contributor controllers are just place-holders in this version; in the version presented later, the Admin controller will handle the tasks of reviewing applicants for membership and changing status of member records, while the Contributor controller will handle picture upload.

The Index controller has the following actions defined:

- index:
  This will present links to other actions. The links that are displayed depend on whether the user is or is not logged in, and if logged in will vary with the user's role.
- login:
  This displays and handles the login form.
- logout:
  This logs out the user, and redisplays the default index page.
- apply-for-membership:
  This displays and handles the membership application. It is the code from section 5.4 with the addition of the code, originally illustrated in section 2, that creates an applicant record in the "MEMBERS" data table and sends an email notification to the designated site owner.

This is the first of these examples application where it is necessary to add some code to the Bootstrap.php class. In this application, the data defining the data base and the email address of the owner are added to the standard application.ini file:

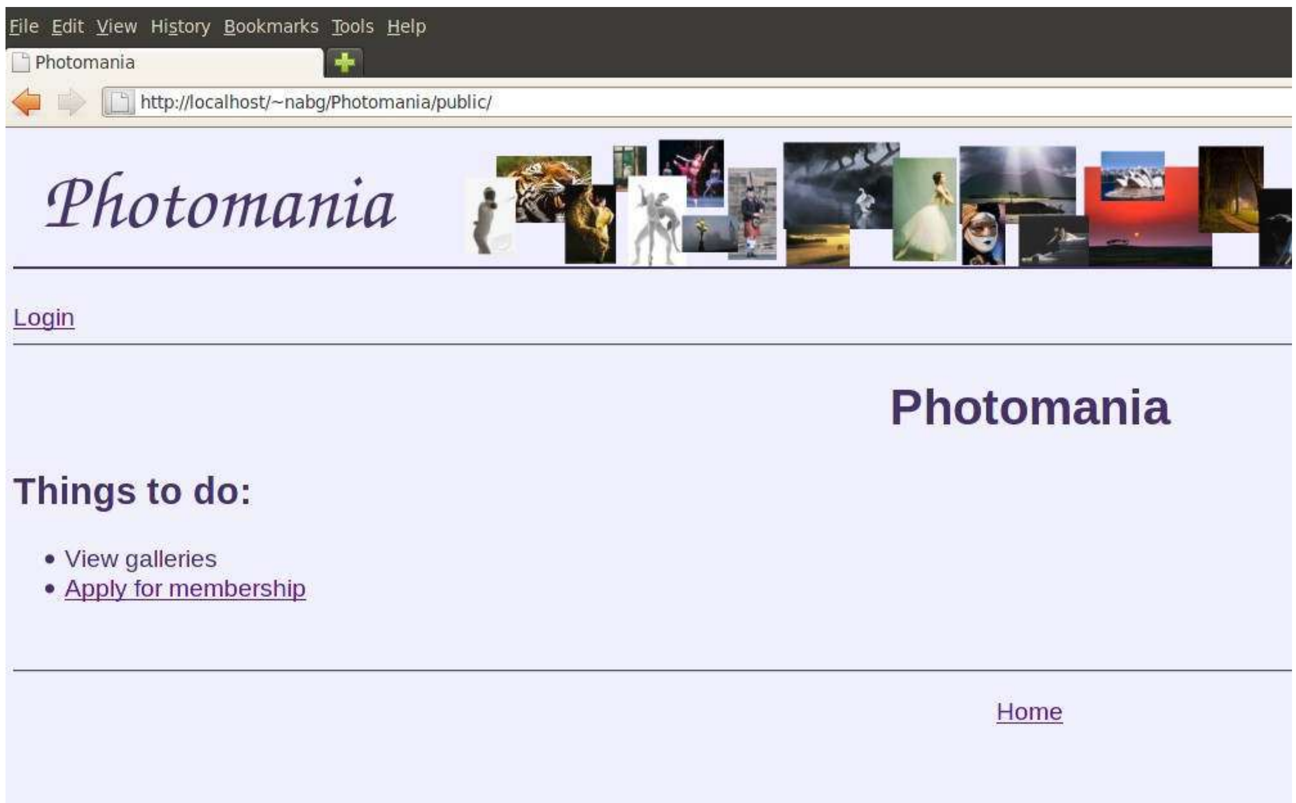
```
[production]
phpSettings.display_startup_errors = 0
phpSettings.display_errors = 0
includePaths.library = APPLICATION_PATH "/../library"
...
resources.db.params.password    = nothispassword
resources.db.params.dbname      = mark
mail.smtpserver             = smtp.someplace.org
mail.from.id                = humee@someplace.org
mail.from.name              = "Hu Mee"
mail.to.id                  = humee@someplace.org
mail.to.name                = "Hu Mee"
mail.subject                = Photomania
```

The Zend framework reads the application.ini file at start up and grabs all the data that it needs; it then discards the structures that it built while reading the file. If there are additional application specific data, like the "mail" data here, then it's best to reload them and save the resulting Zend_Config object in the Zend_Registry. (The Zend_Registry is just a global hash map that can be accessed by any code in the application.) The Bootstrap.php file is a good place for the extra code needed to save a copy of the config object.

```
<?php

class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected function _initConfig()
    {
        $config = new Zend_Config($this->getOptions());
        Zend_Registry::set('config', $config);
        return $config;
    }
```

}

The IndexController's index action has no code – it simply allows the associated views/scripts/index /index.phtml file to be displayed. A Layout.phtml file defines a standard layout for the site pages. The initial view of the site would be as shown here (the View galleries option isn't yet implemented so it isn't a link yet):



The Layout script is in part:

```
<body>
        <div id='headdiv'>
        <img src="<?php echo $this->baseUrl() . '/siteimages/logo.png'; ?>" />
        </div>
        <h1><?php echo $this->headTitle(); ?></h1>
        <?php
         $authenticationdata = Zend_Auth::getInstance();
        if ($authenticationdata->hasIdentity()) {
            echo "<h2>Logged in</h2>";
            echo "<p>Currently logged in as ";
            echo $authenticationdata->getIdentity()->uname;
            echo "; role = ";
            echo $authenticationdata->getIdentity()->role;
            echo ". <a href='";
            echo $this->url(
                   array('controller' => 'index', 'action' => 'logout'));
            echo "' >Logout</a></p>";
        } else {
            echo "<a href='";
            echo $this->url(
                   array('controller' => 'index', 'action' => 'login'));
            echo "' >Login</a></li>";
        }
```

```
        ?>
        <hr size="2" />
        <div id="content">
            <?php echo $this->layout()->content; ?>
        </div>
        <br />
        <hr size="2"/>
        <p align="center">
            <a href='<?php echo $this->baseUrl() ?>'>Home</a></p>
    </body>
</html>
```

The Zend_Auth object's hasIdentity() function checks for the appropriate record in $_SESSION[]. If there is no such record, a link to the IndexController's login action is displayed. If a user has successfully logged in, there will be a record which can be fetched using the getIdentity() method of the Zend_Auth object. The data you get depend on what you choose to store! This application saves “user name” (the login name chosen by the user when creating the account) and the role; the code creating such a record is shown later.

The actual views/scripts/index/index.phtml script, that generates the content for the <div id='content'> section, again uses login status to customise the links that will be displayed. If the user hasn't logged in, the default “view galleries” and “apply for membership” links are shown. (The view galleries scripts will eventually provide the links that allow members to make comments on the pictures.) If the user is logged in as a contributor, there will be a link to the controller used for picture upload; the “owner” has links for picture upload and site administration.

```
<h1 align="center">Photomania</h1>
<h2>Things to do:</h2>
<ul>
    <li>View galleries</li>

    <?php
        if(Zend_Auth::getInstance()->hasIdentity()) {
            $role = Zend_Auth::getInstance()->getIdentity()->role;
            if(($role=="CONTRIBUTOR") || ($role=="OWNER"))
                echo '<li><a href="'.
                 $this->url(
                   array('controller' => 'contributor', 'action' => 'index')) .
                '">Contribute pictures</a></li>';
            if($role=="OWNER") {
                echo '<li><a href="' .
                 $this->url(
                    array('controller' => 'admin', 'action' => 'index')) .
                '">Administer site</li>';
            }
        }
        else {
            echo '<li><a href="' .
              $this->url(
                  array('controller' => 'index',
                        'action' => 'apply-for-membership')) .
                '">Apply for membership</a></li>';
        }
    ?>
</ul>
```

Clicking on the “Login” link at the top of the displayed page will bring up the actual login form:

The Form class for this form is a simpler than that for the membership application:

```
<?php

class Application_Form_Login extends Zend_Form
{

    public function init()
    {
      $this->setMethod('post');
        $username = new Zend_Form_Element_Text('username');
        $username->setLabel('Username:')
                ->setAttrib('size', 10)
                ->setAttrib('maxlength', 10)
                ->setRequired(true)
                ->addFilter('StripTags')
                ->addFilter('StringTrim');
        $password = new Zend_Form_Element_Password('password');
        $password->setLabel('Password')
                ->setAttrib('size', 10)
                ->setAttrib('maxlength', 10)
                ->setRequired(true)
                ->addFilter('Alnum');
        $submit = $this->createElement('submit', 'submit');
        $submit->setLabel("Login");
        $this->addElements(array($username, $password, $submit));
    }


}
```

It is displayed using the views/scripts/index/login.phtml file

```
<?php
echo "<div id='appform'>";
echo "<h1 align='center'>Photomania login form</h1>";
    echo $this->form;
echo "</div>";
```

The code handling login is in the IndexController's loginAction() method:

```
<?php

class IndexController extends Zend_Controller_Action {
```

```
    public function init() { }

    public function indexAction() { }

    private function checkcredentials($formData) {
        $authAdapter = new Zend_Auth_Adapter_DbTable();
        $authAdapter->setTableName('gallerypasswords')
                ->setIdentityColumn('uname')
                ->setCredentialColumn('password')
                ->setCredentialTreatment('MD5(?)');

        $authAdapter->setIdentity($formData['username']);
        $authAdapter->setCredential($formData['password']);

        $authenticator = Zend_Auth::getInstance();
        $check = $authenticator->authenticate($authAdapter);
        if ($check->isValid()) {
            $matchrow = $authAdapter->getResultRowObject(null, 'password');
            $authenticator->getStorage()->write($matchrow);
            $this->_redirect('/'); // send them back to index page
        } else {
            // Let them try to login again
            $this->_redirect('/index/login');
        }
    }

    public function loginAction() {
        $loginform = new Application_Form_Login();
        $this->view->form = $loginform;
        if ($this->getRequest()->isPost()) {
            $inputdata = $this->getRequest()->getPost();
            $this->checkcredentials($inputdata);
            // Either login successful -> back to home page, or
            // login page redisplayed
        }
    }

    public function logoutAction() {
        …
    }

    public function applyForMembershipAction() {
      …
    }

}
```

For a “GET” request, the loginAction() creates the form and has it displayed. For a “POST” request, the action should get the form to verify itself (*are the strings alphanumeric and less than ten characters*?) – but I forgot to include that bit. My code, in my checkcredentials() function, just uses the Zend_Auth and Zend_Auth_Adapter_DbTable methods to check whether the input data represent a valid name and password combination.

The checkcredentials() code creates an instance of the Zend_Auth_Adapter_DbTable; this will use the default db-adapter that the framework will have created using data in the application.ini file. This $authAdapter is then passed parameters that identify the table, columns, and password encoding scheme. The data input as a candidate name and password combination are then added using the setIdentity and setCredential methods.

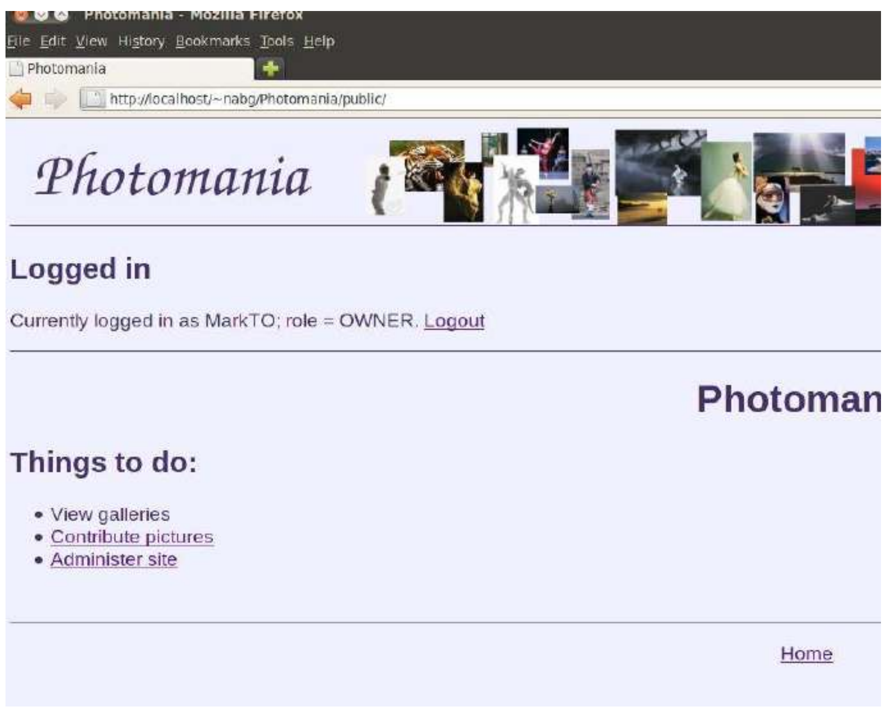
The data table, in a MySQL database has the following schema:

```
CREATE  TABLE `mark`.`gallerypasswords` (
  `uname` VARCHAR(10) NOT NULL ,
  `password` VARCHAR(32) NOT NULL ,
  `role` ENUM('MEMBER', 'CONTRIBUTOR', 'OWNER') NOT NULL DEFAULT 'MEMBER' ,
  PRIMARY KEY (`uname`) ,
  UNIQUE INDEX `uname_UNIQUE` (`uname` ASC) )
ENGINE = MyISAM
DEFAULT CHARACTER SET = latin1;

insert into `mark`.`gallerypasswords` values ('MarkTO', MD5('HardPW'), 'OWNER');
insert into `mark`.`gallerypasswords` values ('DickW1', MD5('DrtyDck'),
                                           'CONTRIBUTOR');
…
insert into `mark`.`gallerypasswords` values ('Goodeye', MD5('Critic'),
                                          'MEMBER');
```

Coding here is a little bit data base specific. MySQL provides a function for applying the MD5 hash encryption function to passwords before they are stored; other databases offer alternatives. Encrypting passwords before storage and checking using the encrypted forms makes the passwords a bit more secure though they are still vulnerable to a dictionary attack; the Zend documentation has a brief description of more elaborate schemes that offer greater security. (Scripts for creating the table and adding some initial data are included with the project.)

The authenticate() method of the Zend_Auth object works with the adapter – in essence it runs a query of the form "*select * from gallerypasswords where uname=? and password=?*". The matching row can be retrieved, $authAdapter->getResultRowObject; the arguments in this call identify the columns that are to be stored in the identity record that will be placed in $_SESSION[]. Here the arguments say save everything apart from the password field, i.e. uname and role. These data are placed into the session record using the getStorage() method to get the identify record and its write function.

If the user successfully logs in, the index page is to be redisplayed (now with new options). Otherwise, the password guesser is given the option of another try by redirection back to the login form. These site navigation steps are handled via the $this->_redirect() calls.



The logout function is simple – just ask the Zend_Auth object to remove any data that it currently has in $_SESSION[] and then redirect to an appropriate page:

```
  public function logoutAction() {
        $auth = Zend_Auth::getInstance();
        $auth->clearIdentity();
        $this->_redirect('/');
    }
```

The applyForMemberShipAction() is in part the same as the code for checking an application form as illustrated in section 5.4. If the form is validated, this version creates a record in the members table for the applicant and sends email to the owner. (The input data can be retrieved from the form after validation; all filters will then have been applied to the data.)

```
    public function applyForMembershipAction() {
        $myform = new Application_Form_MembershipApplicationForm();
        $this->view->form = $myform;
        if ($this->getRequest()->isPost()) {
            $inputdata = $this->getRequest()->getPost();
            if ($myform->isValid($inputdata)) {
                $validdata = $myform->getValues();

                $insertdata = array(
                    'username' => $validdata['uname'],
                    'fullname' => $validdata['username'],
                    'postcode' => $validdata['pcode'],
                    'countrycode' => $validdata['country'],
                    'gender' => $validdata['gender'],
                    'email' => $validata['email'],
                    'yearofbirth' => $validdata['year']
                );
                $config = Zend_Registry::get('config');
                $db = Zend_Db::factory($config->resources->db);
                $db->insert("gallerymembers", $insertdata);


                $emailcontent = $inputdata['username']
                   . " applied for membership on " . date(DateTime::RSS);
                $tr = new Zend_Mail_Transport_Smtp($config->mail->smtpserver);
                Zend_Mail::setDefaultTransport($tr);
                $mail = new Zend_Mail();
                $mail->setFrom($config->mail->from->id,
                        $config->mail->from->name);
                $mail->addTo($config->mail->to->id, $config->mail->to->name);
                $mail->setSubject($config->mail->subject);
                $mail->setBodyText($emailcontent);
                $mail->send();
                $messages = "Thank your for filling in the form." .
                   "The site administrator will review your application " .
                   "and will contact you by email if it is approved.";
                $this->view->messages = $messages;
            } else {
                $myform->populate($inputdata);
            }
        }
    }
```

This version of the project was created without any “model” classes. The code here first gets the $config object from the Zend_Registry, and then uses data from the config object first to create a

simple data base adapter that is used to insert a row, and then to parameterise a mail sender used to forward a notification to the owner.

Once he had successfully logged, in the site owner MarkTO, was able to navigate to the page where (at least in the next version) he can enter commands to administer the site:

Users other than MarkTO will never be shown links that lead to this page. Unfortunately, that doesn't mean that they cannot get to the page, and then execute the actions that will be available via links. After all, some nefarious hacker might guess that there could be an "admin" section associated with the site and simply try the URL ending /Photomania/public/admin – and the hacker would get in:

The indexAction, and later the viewAction and changeAction, methods of the AdminController will all require code that restricts use to an individual logged in with "OWNER" role. If someone navigates to any of these actions without those rights, they should be redirected to the login page.

```
class AdminController extends Zend_Controller_Action {

    public function init() {
        /* Initialize action controller here */
```

```
    }

    public function indexAction() {
        $hacker = true;
        if (Zend_Auth::getInstance()->hasIdentity()) {
            $role = Zend_Auth::getInstance()->getIdentity()->role;
            $hacker = !($role == "OWNER");
        }
        if ($hacker) {
            $this->_redirect('/index/login');
        }
    }

}
```

Control of usage can be effected using such code inserted into all actions. This approach is feasible for a small site with a few roles that are unlikely to change. When you have a large site, or many different and possibly changing roles, you will probably need to use a more elaborate and disciplined approach to access control.

5.5.2 "What are you doing here?" Zend_Acl

The Zend framework library provides for generic "access control" through its Acl component and through hook functions that allow the basic framework to be extended with mechanisms that apply defined access controls.

The Zend Acl class works with "resources", "roles", and "privileges". "Resources" can be whatever the application designer wishes – maybe parts of the data model, or maybe actions. In this example, the requirement is to control who can use the different kinds of functionality of the Photomania application. So, in this case, the real resources are either controllers or specific actions in controllers. Thus only the OWNER should be allowed access to the functions provided by the "AdminController" so restrictions should be applied to all actions of that controller. Everyone should have access to the controller for viewing pictures in the gallery, but only those with appropriate privileges (MEMBERS, CONTRIBUTORS, OWNER) should be able to access the action for adding comments.

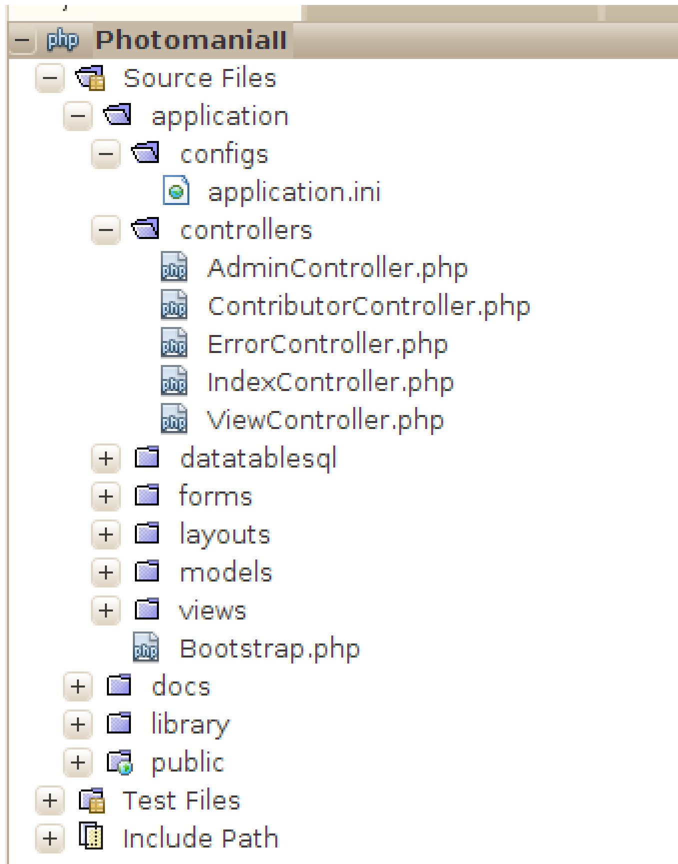
Zend_ACL provides the basics for defining such access rules. Firstly, one defines the roles; these can be specified as a hierarchy. For the Photomania site, the hierarchy would be "GUEST" (limited access), "MEMBER" who can do what a GUEST can do and more such as add tags and comments, "CONTRIBUTOR" who can do what a MEMBER can do and in addition add photos, and OWNER who can do whatever he/she wants.

"Resources" and "privileges" are actually just names. How they are interpreted is up to the application, it isn't an intrinsic part of Zend_Acl. In Zend_Acl one simply identifies named resources. Then, using "allow" and "deny" rules one can define how roles can possess different privileges with regard to these resources.

The second aspect of using Zend_Acl is the mechanism for applying automatically the "allow" / "deny" rules. Application of the rules will involve checking the "role" defined within the identify record created using Zend_Auth (with a default role of "GUEST") when invoking actions. The Zend framework provides a hook function in the mechanism that is used to invoke action methods of controller classes. The framework code that deals with identifying the controller class and action required has a "pre-dispatch" phase. The default implementation for pre-dispatch does nothing. The framework allows the developer to "plug in" instances of application defined classes that extend the appropriate framework base class and implement "predispatch()" functions. The code in these application defined classes can check access permissions, and modify or re-route requests.

The PhotomaniaII project is a complete version of a simple photo sharing site which utilises a specialised Zend_Acl class to define its access control rules which are then enforced using a "plug in" component with a pre-dispatch function.

The PhotoManiaII project is a little larger than the other projects illustrated above. PhotomaniaII has five controllers, seven form classes, a layout class, five model/DbTable classes, about fifteen .phtml template views, four classes defined in the project's library, and modified versions of the standard /public/index.php file and Bootstrap.php file. Such complexity can seem overwhelming to beginners (which may account for numerous comments on the Web by persons who claim that the Zend framework is too complex). But once you get use to the Zend framework style, everything seems to fall reasonably into place. The complete application is included in the examples accompanying these notes; only limited aspects are presented here.



The application.ini file contains resource definitions for a PDO_MySQL database adapter along with configuration data for email dispatch via an SMTP server.

There are five controllers:

- IndexController
    - Index – just some links (varying according to role) to view galleries, apply for membership, contribute photos, and administer site.
    - Apply for membership – shows and handles the application form.
    - Login – shows a simple login form and handles login attempts.
    - Logout – restricted to MEMBERS and higher roles; logs you out.
- ErrorController
    - Deny access – there had to be an action somewhere that would display a message if a hacker managed to reach a URL (controller-action combination) that was not permitted to him. This action was added to the ErrorController.

-ContributorController

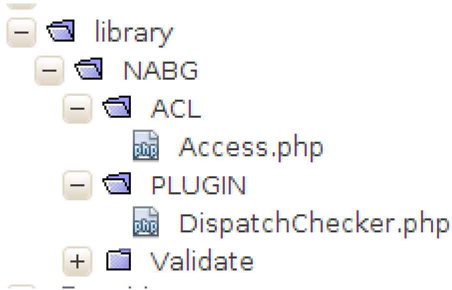
- Index – this action handles the form for upload of another photo to a contributor's gallery.

-AdminController

- Index – just links to the change-status and view-applicants actions.
- View applicants – displays list of records from the "members" table that have status "APPLICANT".
- Change-status – allows changes to status of individuals with concomitant changes in the members table and the role table.

- ViewController

- Index – just links to actions that provide different ways of viewing the galleries.
- View-picture – action that displays a selected picture together with data such as comments by contributor and subsequent comments by other members; the display page will include a link to "add comment" for users who have the privilege.
- View-by-tag – displays a form for a search tag and then a list of links to those pictures that have the requested tag.
- View-by-title – displays titles (which act as links) for all pictures in the galleries.
- View-by-contributor – displays of form for selecting a contributor and then displays a collection of thumbnail pictures (which act as links) for all pictures uploaded by that contributor.
- Add-comment – add comment and tags to a chosen picture.



The main parts of the access control code are made of two classes that are added to the library for this project. The access control class is:

```
/**
 * Description of NABG_ACL_Access
 *
 * @author nabg
 */
class NABG_ACL_Access {
   public $acl;
   public function __construct() {
       // Here define all the roles, resources, and privileges and
       // store them in a Zend_ACL object
       $this->acl = new Zend_Acl();
       // My roles - guest, member, contributor, administrator
       // A guest can do little, a member can do anything a guest can do
```

```
        // plus a little more, a contributor can do ...
        // (the roles were defined as ENUM capitalized strings in db definition)
        $this->acl->addRole('GUEST', null);
        $this->acl->addRole('MEMBER', 'GUEST');
        $this->acl->addRole('CONTRIBUTOR','MEMBER');
        $this->acl->addRole('OWNER');
        // My resources - well really they are the controllers -
        // Index, Contributor, Admin and View (controller for viewing photos),
        // maybe Error
        $this->acl->addResource("index");
        $this->acl->addResource("contributor");
        $this->acl->addResource("admin");
        $this->acl->addResource("error");
        $this->acl->addResource("view");
        // Zend_Acl is supposed to start off with an effective "Deny from All"
        // So add some allowed combinations of actions
        // Owner - do anything
        $this->acl->allow('OWNER');
        $this->acl->allow('GUEST', 'index', array('index',
            'login','apply-for-membership'));
        $this->acl->allow('GUEST', 'error');
        $this->acl->allow('GUEST', 'view', array('index','view-by-contributor',
            'view-by-tag', 'view-by-title', 'view-picture'));
        $this->acl->allow('MEMBER', 'index', array('logout'));
        $this->acl->allow('MEMBER', 'view', array('add-comment'));

    }
}
```

The code creates the Zend_Acl object, then defines the roles. The first argument in the addRole() method is a role name (my role names are capitalized as they need to match the capitalized role names that I use in an Enum datatype in my MySQl table definition). The second is the "parent" class in the role hierarchy that is being defined. Role "Owner" is special.

The resources are then defined – in my case these will be the names of controllers because I am checking for access to controllers and their actions. Here, the resource names must match the controller names as these are used internally by the Zend framework – so just the keyword in lower case.

Finally, the access rules are defined. The arguments in these calls to allow() are the role and, optionally, resource name and a list of privileges. If the list of privileges are omitted, then implicitly "all" are allowed. If no resources are specified, then the allow rule grants access to all. The first rule allows a user with role OWNER to do anything. The second rule says that a GUEST may use the index, login, and apply-for-membership actions in the index controller. The third says a GUEST can use anything in the error controller (don't want to have access control errors reported when trying to report errors!). As specified by the fourth rule, a GUEST can use several of the actions in the view controller. The fifth and sixth rules gave additional privileges to MEMBERs; a MEMBER can logout, and add comments. (A seventh rule was carelessly omitted – as will show up later; you should be able to guess what it should have been.)

The class NABG_PLUGIN_DispatchChecker (remember how class names must specify file hierarchy pathnames) has the pre-dispatch function that will apply the rules:

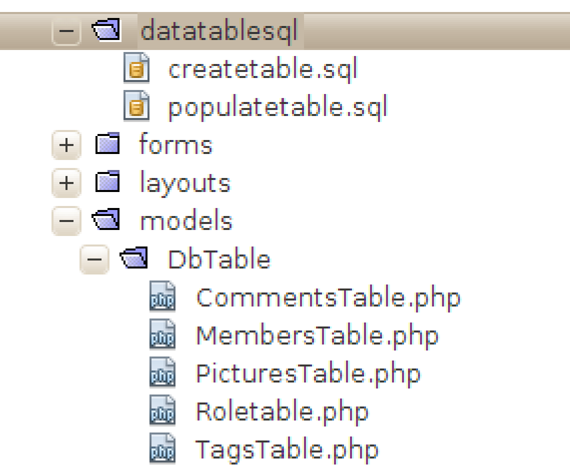
```
class NABG_PLUGIN_DispatchChecker extends Zend_Controller_Plugin_Abstract {

    public function preDispatch(Zend_Controller_Request_Abstract $request) {
        $rightschecker = Zend_Registry::get('myaccessrights');
        $role = "GUEST";
        if (Zend_Auth::getInstance()->hasIdentity()) {
            $role = Zend_Auth::getInstance()->getIdentity()->role;
```

```
        }
        $theResource = $request->getControllerName();
        $action = $request->getActionName();

        if (!$rightschecker->acl->isAllowed($role, $theResource, $action)) {
           $request->setControllerName('Error');
            $request->setActionName('denyaccess');
        }
    }
}
```

The code starts by picking up an instance of NABG_ACL_Access from the Zend_Registry (the approved place for dumping quasi-global objects). (How did an Access object get to be in the registry with the name “myaccessrights”? See the initialisation code given below.) The code then determines the role for the current user. The controller name and action are retrieved from the request object that is being dispatched (if you were using modules to organise the code in a larger project, you would have to bother about the module as well and have some system for checking modules and controllers). With all the data available, the Zend_Acl object is asked to check whether access is allowed. If access is not permitted, the request is diverted to the “deny access” action that has been added to the ErrorController (this action simply displays an error message as defined in its associated view script – views/scripts/error/denyaccess.phtml).

Of course, an instance of NABG_ACL_Access has to be created and added to the registry, and an instance of the NABG_PLUGIN_DispatchChecker class has to be created and inserted into the controller component of the Zend framework. These actions have to be completed before any attempt is made to handle a request. The necessary initialisation steps are added to the public/index.php script that sets up the Zend_Application object. The extra lines added to the standard index.php file are as highlighted here:

```
<?php
// Define path to application directory
defined('APPLICATION_PATH')
    || define('APPLICATION_PATH', realpath(dirname(__FILE__) .
'/../application'));

...

// Create application, bootstrap, and run
$application = new Zend_Application(
    APPLICATION_ENV,
    APPLICATION_PATH . '/configs/application.ini'
);

require_once "NABG/ACL/Access.php";
require_once "NABG/PLUGIN/DispatchChecker.php";

$accessRights = new NABG_ACL_Access();
Zend_Registry::set("myaccessrights",$accessRights);
$interceptor = new NABG_PLUGIN_DispatchChecker();

$frontController = Zend_Controller_Front::getInstance();
$frontController->registerPlugin($interceptor);

$application->bootstrap()
            ->run();
```



The data tables have to be created, and populated with a few entries, before the application can run. SQL scripts for MySQL are provided as part of the project (in a real deployment, such scripts should be stored in some completely different part of the file space). Five tables are defined leading to the five model/DbTable classes. There is some duplication between the "MembersTable" and the "RoleTable" (the MembersTable having been designed for an earlier example); when there is a change in status for a member both tables need to be updated.

The table definitions for the "tags" and "comments" tables are:

```
CREATE  TABLE `mark`.`gallerytags` (
  `idtag` INT NOT NULL AUTO_INCREMENT ,
  `pictureid` INT NOT NULL ,
  `tag` VARCHAR(45) NOT NULL,
   `contrib` VARCHAR(10) NOT NULL,
  PRIMARY KEY (`idtag`) )
ENGINE = MyISAM
DEFAULT CHARACTER SET = latin1;

CREATE  TABLE `mark`.`gallerycomments` (
  `idcomment` INT NOT NULL AUTO_INCREMENT ,
  `commenter` VARCHAR(10) NOT NULL ,
  `comment` VARCHAR(255) NOT NULL ,
  `pictureid` INT NOT NULL,
  PRIMARY KEY (`idcomment`) )
ENGINE = MyISAM
DEFAULT CHARACTER SET = latin1;
```

The auto-increment primary keys in these tables serve no real purpose; however, they are required by the object-relational mapping system that is used. (Every table must have a primary key, or composite primary key, defined in its meta data.) In principle, there should be foreign key constraints linking some tables; but the MyISAM engine ignores such constraints so they weren't specified in the table definitions.

The application stores photos in sub-directories of its public directory. The "siteimages" sub-directory contains images used in the web-site's pages; the "tempimages" sub-directory holds temporary "captcha" images for the "apply for membership" form; the "galleries" sub-directory holds the galleries. Each contributor's photos will be placed in his/her own sub-directory (with a thumbnail image generated along with the full size image).

The application code doesn't contain anything to create directories for new contributors. These directories must be created manually by the site administrator (and must have write permissions given to www-data or whatever that runs the Apache web-server).

Once the galleries sub-directories have been set up, and the data base tables created and populated, the application should run. The OWNER, one MarkTO with passward HardPW, should be able to login:

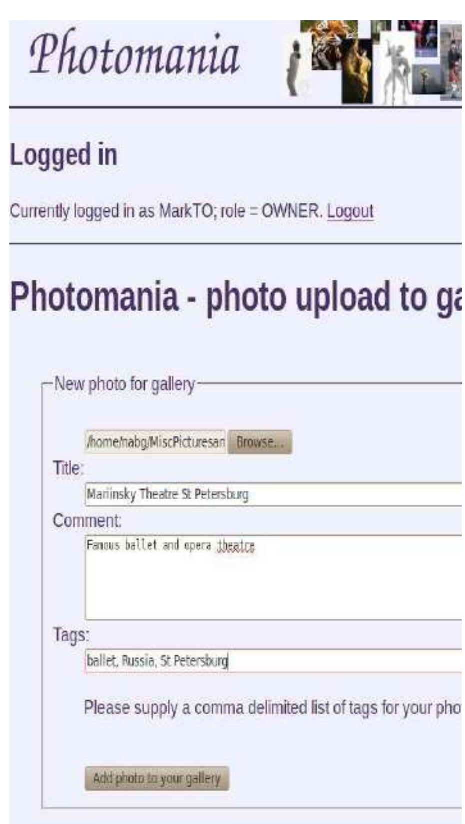
The owner will then be returned to the main index page which will now show different options as appropriate for a logged in OWNER.

The "Contribute pictures", photo upload, option illustrates use of the Zend_FOrm_Element_File component with processing of file input illustrated in the ContributorController::indexAction().

```
class Application_Form_PictureUploadForm extends Zend_Form {

    public function init() {
        $this->setMethod('post');

        $filewithpicture = new Zend_Form_Element_File('photo');
        $filewithpicture->addValidator('MimeType', false, array('image/jpeg'))
                 ->addValidator('Count', false, 1)
                 ->addValidator('Size', false, '1MB');
        ...
```

```
    }
```

The code in the index action loads the data and creates a GD image; this is then scaled down to a thumbnail copy. Both original image and thumbnail copy are saved to file. (There is just one numbering system used for all photos – just the auto-increment primary key generated when a record describing the photo is added to the “pictures” table. The photo's number is used to form the file name – of course this naming scheme does make it easier for hackers to view photos by non-standard access mechanisms.



As shown below, data posted back from the form are validated. If valid, a record is inserted into the pictures table; the insert operation returns the record primary key from the auto-increment sequence. The data from the uploaded file must be “received” -this is just a feature of the Zend framework's file upload mechanism. The data are then loaded, copied to a gallery file, and used to create a thumbnail for the gallery as well. The rest of the code of the indexAction() function updates the tags table with tag data.

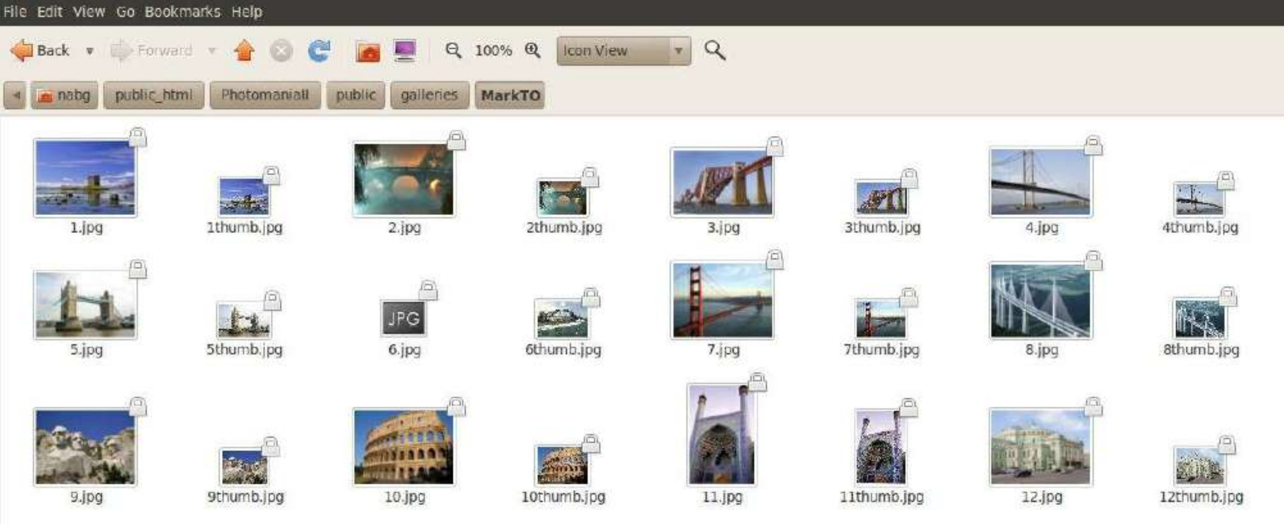
```
class ContributorController extends Zend_Controller_Action {

    public function indexAction() {
        $myform = new Application_Form_PictureUploadForm();
        $this->view->form = $myform;
        if ($this->getRequest()->isPost()) {
            $inputdata = $this->getRequest()->getPost();
            if ($myform->isValid($inputdata)) {
                $title = $myform->getValue('title');
                $comment = $myform->getValue('comment');
                $contrib = Zend_Auth::getInstance()->getIdentity()->uname;
                $data = array(
                    'contributor' => $contrib,
                    'title' => $title,
                    'comment' => $comment
                );
```

```
                $dbtable = new Application_Model_DbTable_PicturesTable();
                $picid = $dbtable->insert($data);


                $outfilename = "./galleries/$contrib/$picid.jpg";
                // Get the bytes uploaded now!
                $myform->photo->receive();
                $filename = $myform->photo->getFileName();

                $numbytes = filesize($filename);

// Read data from file - note that on Windows mode would have to be "rb"
// as the data are binary
                $handle = fopen($filename, "r");

                $imagedata = fread($handle, $numbytes);
                fclose($handle);

                $outhandle = fopen($outfilename, 'w');
                fwrite($outhandle, $imagedata, $numbytes);
                fclose($outhandle);
                $image = imagecreatefromstring($imagedata);
                // save a resized copy
                $imagewidth = imagesx($image);
                $imageheight = imagesy($image);
                $scale = (int) (1 + ($imagewidth / 50));
                $newwidth = (int) (imagesx($image) / $scale);
                $newheight = (int) (imagesy($image) / $scale);
                $thumbimage = imagecreatetruecolor($newwidth, $newheight);
                imagecopyresized($thumbimage, $image, 0, 0, 0, 0,
                        $newwidth, $newheight, $imagewidth, $imageheight);
                $outfilename =  "./galleries/$contrib/${picid}thumb.jpg";
                imagejpeg($thumbimage, $outfilename);
                imagedestroy($image);
                imagedestroy($thumbimage);
            …
      }
}
```

Photomania - photo upload to gallery

Picture #12 uploaded to your gallery.

The picture is tagged with these tags :ballet, Russia, St Petersburg



Another of the CONTRIBUTORs defined in the populatetable.sql file attempted to upload a picture but encountered the following response:

There was an "allow" rule missing in that NABG_ACL_Access class:

```
$this->acl->allow('CONTRIBUTOR', 'contributor');
```

If you add that line, your other registered CONTRIBUTORs should be able to upload pictures.

Once the galleries have been populated, the view options can be tested.

The view by title option generates a page with all the picture titles acting as links to the view-picture action that displays the actual photo.

```
<h1 align="center">Viewing the Photomania Galleries</h1>
<h2>Titles</h2>

<ul>
    <?php
    foreach($this->titles as $item) {
        echo "<li><a href='";
        echo $this->url(array(
            'controller' => 'view',
            'action' => 'view-picture',
            'picid' => $item->idpictures));
        echo "'>";
        echo $item->title;
        echo "</a></li>";
    }
    ?>
</ul>
```

Titles

- Alfredo and Violetta in Brindisi scene
- Bluebell
- Castle stalker
- Clare bridge
- Colliseum Rome
- Dandelion
- Forth rail bridge

The generated HTML being:

```
<ul>
    <li><a href='/~nabg/PhotomaniaII/public/view/view-picture/picid/18'>Alfredo
and Violetta in Brindisi scene</a></li>
```

The arguments for an action just become part of the complex URL as illustrated here with picid=18 being passed.

The view by tag option has a form to enter a tag. When this is posted back, data must be retrieved from the tag table and picture table. This is most readily done by just getting a Zend_DB_Adapter and using this to run the requisite SQL select statement. The resulting list of titles is displayed in the same way as shown above.

```
public function viewByTagAction() {
// method of ViewController
        $myform = new Application_Form_ViewByTagForm();
        $this->view->form = $myform;
        if ($this->getRequest()->isPost()) {
            $inputdata = $this->getRequest()->getPost();
            if ($myform->isValid($inputdata)) {
                $tagwanted = $myform->getValue('tag');

                $bootstrap = $this->getInvokeArg('bootstrap');
                $resource = $bootstrap->getPluginResource('db');
                $db = $resource->getDbAdapter();
                $sqlrequest = "select idpictures,title from pictures where " .
     "idpictures in (select pictureid from gallerytags where tag='$tagwanted')";
                $rows = $db->fetchAll($sqlrequest, Zend_Db::FETCH_OBJ);

                $this->view->tagwanted = $tagwanted;
                $this->view->data = $rows;
            } else {
                $myform->populate($inputdata);
            }
        }
```

The view by contributor option requires a HTML <select> with options that are the identifiers of all contributors who have uploaded pictures. The form definition creates an empty Zend_Form_

Element_Select:

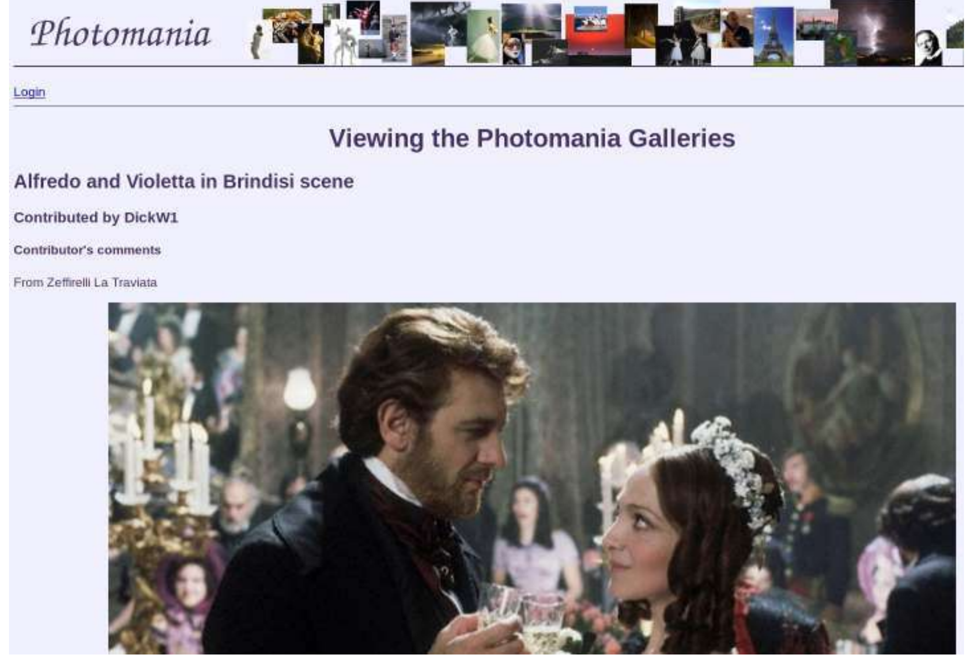
```
class Application_Form_ViewByContributorForm extends Zend_Form {

    public function init() {
        $this->setMethod('post');
        $contribs = new Zend_Form_Element_Select('contrib');
        $contribs->setLabel('Contributor');

        $submit = $this->createElement('submit', 'submit');
        $submit->setLabel("View by contributor");
        $this->addElements(array($contribs, $submit));
    }

}
```

The options are added in the controller action before the form is used:

```
    public function viewByContributorAction() {
// method of ViewController
        $myform = new Application_Form_ViewByContributorForm();

        $bootstrap = $this->getInvokeArg('bootstrap');
        $resource = $bootstrap->getPluginResource('db');
        $db = $resource->getDbAdapter();
        $sqlrequest = "select distinct contributor from pictures";
        $rows = $db->fetchAll($sqlrequest);
        $persons = array();
        foreach ($rows as $row)
            $persons[] = $row['contributor'];
        $myform->getElement('contrib')->addMultiOptions($persons);
        $this->view->form = $myform;
      …
      …
      }
```

The display for view by contributor is as an array of thumbnail images that act as links to the view-picture action.

The picture display page shows the full sized photo along with the contributor's comments and the tags assigned. If there are comments by other members these are listed below the picture. When appropriate, the displayed page will have a link to the action for adding comments.



5.6 But wait, there's more!

Of course there is more. I haven't mentioned things like “view helpers” and “action helpers”, and there are many other more sophisticated things that you can do with plugins etc.

If you work through all the examples in this gentle introduction then you should be able to handle the more severe text books and web tutorials.

Notes

Note 1: The code

There should be an accompanying gzip file containing all the code as NetBeans projects.

You will have to modify the code before it will run! There are things like directory pathnames in the .htaccess files – these need to be changed to your file arrangements; there are database URLs, user names and passwords in application.ini files; and there are configuration data for email that reference SMTP servers. All these need to be changed.

Note 2: The set up

You will of course have to have your Apache, PHP, MySQL system set up correctly (and if you need Oracle, you will need the instant client software and will have to edit the php.ini file to add the Oracle configuration data). Maybe one day I'll add some more notes on how to configure the system from scratch.

The following fragments from my phpinfo() report might help:

apache2handler

Apache Version	Apache/2.2.14 (Ubuntu)

Apache API Version	20051115
Server Administrator	webmaster@localhost
Hostname:Port	127.0.1.1:80
User/Group	www-data(33)/33
Max Requests	Per Child: 0 - Keep Alive: on - Max Per Connection: 100
Timeouts	Connection: 300 - Keep-Alive: 15
Virtual Server	Yes
Server Root	/etc/apache2
Loaded Modules	core mod_log_config mod_logio prefork http_core mod_so mod_alias mod_auth_basic mod_authn_file mod_authz_default mod_authz_groupfile mod_authz_host mod_authz_user mod_autoindex mod_cgi mod_deflate mod_dir mod_env mod_mime mod_negotiation mod_php5 mod_reqtimeout mod_rewrite mod_setenvif mod_status mod_userdir

gd

GD Support	enabled
GD Version	2.0

mysql

MySQL Support	enabled
Active Persistent Links	0
Active Links	0
Client API version	5.1.41
MYSQL_MODULE_TYPE	external
MYSQL_SOCKET	/var/run/mysqld/mysqld.sock
MYSQL_INCLUDE	-I/usr/include/mysql
MYSQL_LIBS	-L/usr/lib -lmysqlclient_r

mysqli

MysqlI Support	enabled

Client API library version	5.1.41
Active Persistent Links	0
Inactive Persistent Links	0
Active Links	0
Client API header version	5.1.41
MYSQLI_SOCKET	/var/run/mysqld/mysqld.sock

oci8

OCI8 Support	enabled
Version	1.4.4
Revision	$Revision: 305257 $
Active Persistent Connections	0
Active Connections	0
Oracle Instant Client Version	10.2
Temporary Lob support	enabled
Collections support	enabled

PDO

PDO support	enabled
PDO drivers	mysql, sqlite, sqlite2

pdo_mysql

PDO Driver for MySQL	enabled
Client API version	5.1.41

pdo_sqlite

PDO Driver for SQLite 3.x	enabled
SQLite Library	3.6.22

Note 3: The projects

1 SmartyPicLibrary (example of section 1.1)

Illustrates the separation of View from model and control. The Smarty view templates provide an easy way to layout pages; standard PHP coding styles handle the model and control aspects.

You will need to modify the file paths specifying where Smarty library is and where it stores its temporary files. You will need to change the database user-name and password details.

2 ZendComponents1 (examples from section 2)

Examples illustrating use of some simple Zend components:

- Zend_Captch_Image
- Zend_Validate_Email_Address
- Zend_Validate_StringLength
- Zend_Validate_Alnum
- Zend_Validate_Between
- validator chain
- Zend_Locale
- Zend_Validate_PostCode
- Application specific validator class
- Zend_Db_Adapter
- Zend_Validate_Db_RecordExists
- Zend_Log_Writer
- Zend_Config
- Zend_Mail

As well as changing the Smarty pathnames and database details, remember to alter the email configuration, create a directory (with appropriate permissions) for the captcha images, and set a suitable file-path for the font files.

3 ZendDBComponents (examples from section 3.1)

Examples illustrating use Zend's classes for working with databases:

- Zend_Db::factory()
- Zend_Db_Adapter
- Zend_Db_Select
- Zend_Paginator

The usual re-configuration steps will bee needed.

4 ZendDBTableExamples (examples from section 3.2)

- Zend_Db_Table
- Own classes extending Zend_Db_Table_Abstract

- Zend_Db_Table_Select
- Own classes extending Zend_Db_Table_Row_Abstract

Why the different versions of SchoolTable and TeacherTable? The different examples show varied approaches and require changed class definitions.

5 ZF01 (example in section 5.1)

- zend.sh and (some) of its options
- IndexController

You don't really need this – it's the default empty project created by running the zend.sh set up script (NetBeans runs the script when you specify a PHP project using Zend).

6 ZFSchools_1 (examples in section 5.2)

- class Application_Model_DbTable_SchoolTable - naming conventions when working with framework
- Controllers and actions
- .phtml template view files
- Zend_Paginator
- Layout

Remember to change the RewriteBase value in this and all remaining projects (it's in the .htaccess file in the /public directory).

7 ZFSchools_2 and ZFSchools_2b (examples in section 5.3)

- Lots of controllers and actions
- Zend_Filter classes and filter chains
- Zend_Validate again

Just the usual re-configuration as always needed.

8 ApplicationForm_II (examples in section 5.4)

- Zend_Form and Zend_Form_Elements
- Automated form validation
- A library of application defined classes

9 PhotoMania (example for section 5.5.1)

- Zenda_Auth
- Modifying the Bootstrap.php

10 PhotoManiaII (example for section 5.5.2)

- Zend_ACL

- Plugins for dispatcher