Laravel-Metable Documentation

Release 1.0

Sean Fraser

May 15, 2018

Contents

1 Introduction **3**
- 1.1 Features . . . 3
- 1.2 Installation . . . 3
- 1.3 Example Usage . . . 4

2 Handling Meta **5**
- 2.1 Attaching Meta . . . 5
- 2.2 Retrieving Meta . . . 6
- 2.3 Retrieving All Meta . . . 6
- 2.4 Checking For Presence of Meta . . . 6
- 2.5 Deleting Meta . . . 7
- 2.6 Eager Loading Meta . . . 7

3 Querying Meta **9**
- 3.1 Checking for Presence of a key . . . 9
- 3.2 Comparing value . . . 9
- 3.3 Ordering results . . . 10
- 3.4 A Note on Optimization . . . 11

4 Data Types **13**
- 4.1 Scalar Values . . . 13
- 4.2 Objects . . . 14
- 4.3 Adding Custom Data Types . . . 16

Laravel-Metable is a package for easily attaching arbitrary data to Eloquent models for Laravel 5.

CHAPTER 1

Introduction

Laravel-Metable is a package for easily attaching arbitrary data to Eloquent models for Laravel 5.

1.1 Features

- One-to-many polymorphic relationship allows attaching data to Eloquent models without needing to adjust the database schema.
- Type conversion system allows data of numerous different scalar and object types to be stored, queried and retrieved. See the list of supported *Data Types*.

1.2 Installation

1. Add the package to your Laravel app using composer

```
composer require plank/laravel-metable
```

2. Register the package's service provider in `config/app.php`. In Laravel versions 5.5 and beyond, this step can be skipped if package auto-discovery is enabled.

```
<?php
'providers' => [
    ...
    Plank\Metable\MetableServiceProvider::class,
    ...
];
```

3. Publish the config file (`config/metable.php`) and migration file (`database/migrations/####_##_##_######_create_metable_table.php`) of the package using artisan.

```
php artisan vendor:publish --provider="Plank\Metable\MetableServiceProvider"
```

4. Run the migrations to add the required table to your database.

```
php artisan migrate
```

5. Add the Plank\Metable\Metable trait to any eloquent model class that you want to be able to attach metadata to.

1.3 Example Usage

Attach some metadata to an eloquent model

```
<?php
$post = Post::create($this->request->input());
$post->setMeta('color', 'blue');
```

Query the model by its metadata

```
<?php
$post = Post::whereMeta('color', 'blue');
```

Retrieve the metadata from a model

```
<?php
$value = $post->getMeta('color');
```

CHAPTER 2

Handling Meta

before you can attach meta to an Eloquent model, you must first add the `Metable` trait to your Eloquent model.

```php
<?php

namespace App;

use Plank\Metable\Metable;
use Illuminate\Database\Eloquent\Model;

class Page extends Model
{
        use Metable;

        // ...
}
```

2.1 Attaching Meta

Attach meta to a model with the `setMeta()` method. The method accepts two arguments: a string to use as a key and a value. The value argument will accept a number of different inputs, which will be converted to a string for storage in the database. See the list of a supported *Data Types*.

```php
<?php
$model->setMeta('key', 'value');
```

To set multiple meta key and value pairs at once, you can pass an associative array or collection to `syncMeta()`.

```php
<?php
$model->syncMeta([
        'name' => 'John Doe',
        'age' => 18,
]);
```

2.2 Retrieving Meta

You can retrieve the value of the meta at a given key with the `getMeta()` method. The value should be returned in the same format that it was stored. For example, if an array is set, you will receive an array back when retrieving it.

```
<?php

$model->setMeta('age', 18);
$model->setMeta('approved', true);
$model->setMeta('accessed_at', Carbon::now());

//reload the model from the database
$model = $model->fresh();

$age = $model->getMeta('age'); //returns an integer
$approved = $model->getMeta('approved'); //returns a boolean
$accessDate = $model->getMeta('accessed_at'); //returns a Carbon instance

//etc.
```

You may pass a second parameter to the method in order to specify a default value to return if no meta had been set at that key.

```
<?php
$model->getMeta('status', 'draft');
```

Note: If a falsey value (e.g. `0`, `false`, `null`, `''`) has been manually set for the key, that value will be returned instead of the default value. The default value will only be returned if no meta exists at the key.

Once loaded, all meta attached to a model instance are cached in the model's `meta` relationship. As such, successive calls to `getMeta()` will not hit the database repeatedly.

Similarly, the unserialized value of each meta is cached once accessed. This is particularly relevant for attached *Eloquent Models* and similar database-dependant objects.

Setting a new value for a key automatically updates all caches.

2.3 Retrieving All Meta

To retrieve a collection of all meta attached to a model, expressed as key and value pairs, use `getAllMeta()`.

```
<?php
$meta = $model->getAllMeta();
```

2.4 Checking For Presence of Meta

You can check if a value has been assigned to a given key with the `hasMeta()` method.

```
<?php
if ($model->hasMeta('background-color')) {
        // ...
}
```

Note: This method will return `true` even if a falsey value (e.g. `0`, `false`, `null`, `''`) has been manually set for the key.

2.5 Deleting Meta

To remove the meta stored at a given key, use `removeMeta()`.

```
    <?php
$model->removeMeta('prefered_language');
```

To Remove all meta from a model, use `purgeMeta()`.

```
    <?php
$model->purgeMeta();
```

Attached meta is automatically purged from the database when a `Metable` model is manually deleted. Meta will *not* be cascaded if the model is deleted by the query builder.

```
<?php
$model->delete(); // will delete attached meta
MyModel::where(...)->delete() // will NOT delete attached meta
```

2.6 Eager Loading Meta

When working with collections of `Metable` models, be sure to eager load the meta relation for all instances together to avoid repeated database queries (i.e. N+1 problem).

Eager load from the query builder:

```
<?php
$models = MyModel::with('meta')->where(...)->get();
```

Lazy eager load from an Eloquent collection:

```
<?php
$models->load('meta');
```

You can also instruct your model class to *always* eager load the meta relationship by adding `'meta'` to your model's `$with` property.

```
<?php

class MyModel extends Model {
    use Metable;

    protected $with = ['meta'];
}
```

CHAPTER 3

Querying Meta

The `Metable` trait provides a number of query scopes to facilitate modifying queries based on the meta attached to your models

3.1 Checking for Presence of a key

To only return records that have a value assigned to a particular key, you can use `whereHasMeta()`. You can also pass an array to this method, which will cause the query to return any models attached to one or more of the provided keys.

```
<?php
$models = MyModel::whereHasMeta('notes')->get();
$models = MyModel::whereHasMeta(['queued_at', 'sent_at'])->get();
```

If you would like to restrict your query to only return models with meta for *all* of the provided keys, you can use `whereHasMetaKeys()`.

```
<?php
$models = MyModel::whereHasMetaKeys(['step1', 'step2', 'step3'])->get();
```

You can also query for records that does not contain a meta key using the `whereDoesntHaveMeta()`. It's signature is identical to that of ‘‘whereHasMeta()‘.

```
<?php
$models = MyModel::whereDoesntHaveMeta('notes')->get();
$models = MyModel::whereDoesntHaveMeta(['queued_at', 'sent_at'])->get();
```

3.2 Comparing value

You can restrict your query based on the value stored at a meta key. The `whereMeta()` method can be used to compare the value using any of the operators accepted by the Laravel query builder's `where()` method.

```php
<?php
// omit the operator (defaults to '=')
$models = MyModel::whereMeta('letters', ['a', 'b', 'c'])->get();

// greater than
$models = MyModel::whereMeta('name', '>', 'M')->get();

// like
$models = MyModel::whereMeta('summary', 'like', '%bacon%')->get();

//etc.
```

The `whereMetaIn()` method is also available to find records where the value is matches one of a predefined set of options.

```php
<?php
$models = MyModel::whereMetaIn('country', ['CAN', 'USA', 'MEX']);
```

The `whereMeta()` and `whereMetaIn()` methods perform string comparison (lexicographic ordering). Any non-string values passed to these methods will be serialized to a string. This is useful for evaluating equality (=) or inequality (<>), but may behave unpredictably with some other operators for non-string data types.

```php
<?php
// array value will be serialized before it is passed to the database
$model->setMeta('letters', ['a', 'b', 'c']);

// array argument will be serialized using the same mechanism
// the original model will be found.
$model = MyModel::whereMeta('letters', ['a', 'b', 'c'])->first();
```

Depending on the format of the original data, it may be possible to compare against subsets of the data using the SQL `like` operator and a string argument.

```php
<?php
$model->setMeta('letters', ['a', 'b', 'c']);

// check for the presence of one value within the json encoded array
// the original model will be found
$model = MyModel::whereMeta('letters', 'like', '%"b"%' )->first();
```

When comparing integer or float values with the <, <=, >= or > operators, use the `whereMetaNumeric()` method. This will cast the values to a number before performing the comparison, in order to avoid common pitfalls of lexicographic ordering (e.g. `'11'` is greater than `'100'`).

```php
<?php
$models = MyModel::whereMetaNumeric('counter', '>', 42)->get();
```

3.3 Ordering results

You can apply an order by clause to the query to sort the results by the value of a meta key.

```php
<?php
// order by string value
$models = MyModel::orderByMeta('nickname', 'asc')->get();
```

```
//order by numeric value
$models = MyModel::orderByMetaNumeric('score', 'desc')->get();
```

By default, all records matching the rest of the query will be ordered. Any records which have no meta assigned to the key being sorted on will be considered to have a value of `null`.

To automatically exclude all records that do not have meta assigned to the sorted key, pass `true` as the third argument. This will perform an inner join instead of a left join when sorting.

```
<?php
// sort by score, excluding models which have no score
$model = MyModel::orderByMetaNumeric('score', 'desc', true)->get();

//equivalent to, but more efficient than
$models = MyModel::whereHasMeta('score')
    ->orderByMetaNumeric('score', 'desc')->get();
```

3.4 A Note on Optimization

Laravel-Metable is intended a convenient means for handling data of many different shapes and sizes. It was designed for dealing with data that only a subset of all models in a table would have any need for.

For example, you have a Page model with a template field and each template needs some number of additional fields to modify how it displays. If you have X templates which each have up to Y fields, adding all of these as columns to pages table will quickly get out of hand. Instead, appending these template fields to the Page model as meta can make handling this use case trivial.

Laravel-Metable makes it very easy to append just about any data to your models. However, for sufficiently large data sets or data that is queried very frequently, it will often be more efficient to use regular database columns instead in order to take advantage of native SQL data types and indexes. The optimal solution will depend on your use case.

CHAPTER 4

Data Types

You can attach a number of different kinds of values to a `Metable` model. The data types that are supported by Laravel-Mediable out of the box are the following.

4.1 Scalar Values

The following scalar values are supported.

4.1.1 Array

Arrays of scalar values. Nested arrays are supported.

```
<?php
$metable->setMeta('information', [
        'address' => [
                'street' => '123 Somewhere Ave.',
                'city' => 'Somewhereville',
                'country' => 'Somewhereland',
                'postal' => '123456',
        ],
        'contact' => [
                'phone' => '555-555-5555',
                'email' => 'email@example.com'
        ]
]);
```

Warning: Laravel-Metable uses `json_encode()` and `json_decode()` under the hood for array serialization. This will cause any objects nested within the array to be cast to an array.

4.1.2 Boolean

```php
<?php
$metable->setMeta('accepted_promotion', true);
```

4.1.3 Integer

```php
<?php
$metable->setMeta('likes', 9001);
```

4.1.4 Float

```php
<?php
$metable->setMeta('precision', 0.755);
```

4.1.5 Null

```php
<?php
$metable->setMeta('linked_model', null);
```

4.1.6 String

```php
<?php
$metable->setMeta('attachment', '/var/www/html/public/attachment.pdf');
```

4.2 Objects

The following classes and interfaces are supported.

4.2.1 Eloquent Models

It is possible to attach another Eloquent model to a `Metable` model.

```php
<?php
$page = App\Page::where(['title' => 'Welcome'])->first();
$metable->setMeta('linked_model', $page);
```

When `$metable->getMeta()` is called, the attached model will be reloaded from the database.

It is also possible to attach a `Model` instance that has not been saved to the database.

```php
<?php
$metable->setMeta('related', new App\Page);
```

When `$metable->getMeta()` is called, a fresh instance of the class will be created (will not include any attributes).

4.2.2 Eloquent Collections

Similarly, it is possible to attach multiple models to a key by providing an instance of `Illuminate\Database\Eloquent\Collection` containing the models.

As with individual models, both existing and unsaved instances can be stored.

```
<?php
$users = App\User::where(['title' => 'developer'])->get();
$metable->setMeta('authorized', $users);
```

4.2.3 DateTime & Carbon

Any object implementing the `DateTimeInterface`. Object will be converted to a `Carbon` instance.

```
<?php
$metable->setMeta('last_viewed', \Carbon\Carbon::now());
```

4.2.4 Serializable

Any object implementing the PHP `Serializable` interface.

```
<?php
class Example implements \Serializable
{
        //...
}

$serializable = new Example;

$metable->setMeta('example', $serializable);
```

4.2.5 Plain Objects

Any other objects will be converted to `stdClass` plain objects. You can control what properties are stored by implementing the `JsonSerializable` interface on the class of your stored object.

```
<?php
$metable->setMeta('weight', new Weight(10, 'kg'));
$weight = $metable->getMeta('weight') // stdClass($amount = 10; $unit => 'kg');
```

Note: The `Plank\Metable\DataType\ObjectHandler` class should always be the last entry the `config/metable.php` datatypes array, as it will accept any object, causing any handlers below it to be ignored.

Warning: Laravel-Metable uses `json_encode()` and `json_decode()` under the hood for plain object serialization. This will cause any arrays within the object's properties to be cast to a `stdClass` object.

4.3 Adding Custom Data Types

You can add support for other data types by creating a new `Handler` for your class, which can take care of serialization. Only objects which can be converted to a string and then rebuilt from that string should be handled.

Define a class which implements the Plank\Metable\DataType\Handler interface and register it to the `'datatypes'` array in `config/metable.php`. The order of the handlers in the array is important, as Laravel-Metable will iterate through them and use the first entry that returns `true` for the `canHandleValue()` method for a given value. Make sure more concrete classes come before more abstract ones.