Laravel secrets

by Stefan Bauer & Bobby Bouwmann

Table of Contents

Models

Laravel is very popular because of Eloquent, the ORM layer. All the database queries and relations start with a model. A model is a class that represents the structure of the database table and its relationships to other tables. A model makes it possible to retrieve, insert, update, and delete records from the underlying database table.

In this chapter, we will dive into the model class's hidden options. You will learn how you can use them to keep your code clean and make tasks easy. We will mostly use the `Book` model as an example. This is an empty model class that we can create by running the following on the command line:

```
php artisan make:model Book
```

This will result in the following class.

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    //
}
```

To understand some concepts, you also need to know how the structure looks like in the database. For the `Book` model, we have a migration that looks like this:

```
class CreateBooksTable extends Migration
{
    public function up()
    {
        Schema::create('books', function (Blueprint $table) {
            $table->id();
            $table->foreignId('author_id')->constrained();
            $table->string('title');
            $table->text('description');
            $table->date('published_at');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('books');
    }
}
```

Let's dive in!

Attributes

Laravel makes it easy to retrieve the values of the model you are working with. The static magic methods of PHP are used to achieve that. This makes it possible to magically get a value of the model or a full relationship.

```
$book = Book::first();

$book->title; // Get an attribute of the model
$book->author; // Get the author(user) relationship of the model
$book->auhtor(); // Get the author relationship query builder object

$book->title = 'My new book title'; // Set an atribute on the model
```

The magic method calls a method on the model that retrieves the correct value or sets the model's value.

```
// Illuminate\Database\Eloquent\Model

public function __get($key)
{
    return $this->getAttribute($key);
}

public function __set($key, $value)
{
    $this->setAttribute($key, $value);
}
```

The `getAttribute` and `setAttribute` methods make it possible to easily add custom functionality. For example, accessors and mutators but also casting of attributes. More on that in the next chapters.

Since we can change and retrieve values from the model, the model should keep some kind of state. Internally this means a model keeps track of two arrays: `$attributes` and `$originial`. The names are pretty straightforward and explain what they do. Basically, whenever you update an attribute by setting a new value, the `$attributes` array is updated. This way, you can always go back to the original value.

The `$attributes` property is always used whenever you retrieve a property from the model because this holds the model's current state. Whenever you want to get an array presentation of your model or want the JSON presentation of your model, it will use the `$attributes` as the basis.

```
$book->toArray();
$book->toJson();

public function toArray()
{
    return array_merge(
        $this->attributesToArray(),
        $this->relationsToArray()
    );
}

public function toJson($options = 0)
{
    return json_encode($this->toArray(), $options);
}
```

As you can see in the above code block, the model has two methods: `toArray` and `toJson`. We simplified the methods a little bit, but it should give you the right image of what's going on.

The `toJson` method is utilizing the `toArray` method here. The `toArray` method is calling two methods that both return an array. The outcome is the merged results. The `attributesToArray` method will retrieve all the model values in the

`$attributes` array. After that, it will look for any date fields, accessors, and casts. From there, they are converted to the correct format and returned in the array.

```
 Book::first()->toArray();

[
    'id' => 1,
    'author_id' => '1',
    'title' => 'test',
    'description' => 'test',
    'created_at' => '2020-03-21T13:51:18.000000Z',
    'updated_at' => '2020-03-21T13:51:18.000000Z',
]
```

The `relationsToArray` method is called here as well. This method will convert the loaded models via de relationships to their array presentation as well. So the `toArray` method is called on all the models of the loaded relations. What does loaded mean here? Well, those are all the relationships that have been loaded on the model by using `with` or `load` on the model. The model itself also keeps a list of all these relationships that have been loaded in the `$relations` array.

Snippets

This chapter will give you some quick tips and tricks you might not find in the documentation. Most snippets are based on our own experience working with Laravel. The snippets are categorized by their main subject in the subchapters. Enjoy the snippets and apply them to your code.

Find Multiple Records by ID

We are sure you know the `find()` method to find a concrete record with the database's given ID. Correct? What if we want to retrieve multiple records of which you know the IDs? We can also pass various IDs in an array instead of a single ID.

> **Note:** In this case a collection is returned instead of the model.

```
$products = Product::find([1, 2, 3]);
```

Find Related IDs on a BelongsToMany Relationship

Imagine that there is a user. One user can have many roles, and one role can have many users. This is a classic many-to-many relationship. Each of the models now implements a respective method with a `BelongsToMany` relationship. At this point, we only look at the user model.

```
class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles()
    {
        return $this->belongsToMany('App\Models\Role');
    }
}
```

If we now want to receive all IDs of roles belonging to a particular user, the `pluck()` method is undoubtedly the right choice.

```
$user = User::find(1);
$roleIds = $user->roles()->pluck('id')->toArray();
```

There is an alternative that works the same way, but we do not need to include the actual column name. The method automatically uses the correct key of the corresponding pivot table. The method for this is `allRelatedIds()`.

```
$user = User::find(1);
$roleIds = $user->roles()->allRelatedIds()->toArray();
```

Clone a Model into a New Instance

If you want to copy an existing model, based on a database row, into a new non-existing instance, you can use the method `replicate`.

```
$product = Product::find(1);
$copy = $product->replicate();
```

In `$copy`, you now have an exact copy of your origin `$product`. The only difference is that it resets the default values like `created_at` and `updated_at`. As a second parameter, you can define the default that should be taken care of.

> **Note:** If you call `replicate`, a Laravel internal event gets fired called `eloquent.replicating`.

Determine If a Record Was Found, or a New Entry Was Created

From time to time, you create entries in the database only if they are not yet available. Fortunately, Laravel offers us an excellent function here. The method `$model->firstOrCreate()` either finds the first entry and returns it or creates a new entry and then returns it.

Sometimes it is necessary to determine if an entry was created, or an entry was found and returned. For this purpose, Laravel already provides built-in functionality.

```
$product = Product::firstOrCreate([
    'name' => 'Laravel Secrets',
    'category' => 'book',
]);

if ($product->wasRecentlyCreated()) {
    // A new record was created
} else {
    // An existing record was found
}
```