Eloquent ORM

The essentials

> Laravel includes Eloquent, an object-relational mapper (ORM) that makes it enjoyable to interact with your database. When using Eloquent, each database table has a corresponding "Model" that is used to interact with that table. In addition to retrieving records from the database table, Eloquent models allow you to insert, update, and delete records from the table as well. (laravel.com)

Typically, when developers want to interact with a database table, they create a migration for each table that define the table's columns. They create a model that will interact with that table. When needed, they extend the model with extra attributes and methods to define relationships and other business logic.

Because these tasks are very common, Laravel again tries to help you as much as possible by assuming various aspects based on your naming of things. Therefore it is important to know these naming conventions.

Generating models

Models can be generated with the following Artisan command:

```
php artisan make:model FooBar
```

This command tells Laravel to create a `app\Models\FooBar.php` file that typically looks like this:

```
namespace App\Models; use
Illuminate\Database\Eloquent\Factories\HasFactory; use
Illuminate\Database\Eloquent\Model; class FooBar extends Model { use
HasFactory; }
```

It defines a class that extends the `Illuminate\Database\Eloquent\Model` class.

Model conventions

As mentioned earlier, Laravel expects (or actually is configured such) that the following is true:

- There should be a table present with the name `foo_bars`: the snake case plural name of the class
- The primary key is a numeric, auto incrementing column `id`
- The table has the two timestamp columns `created_at` and `updated_at` defined

> ⚠️ We highly recommend to follow these conventions as much as possible. Only deviate from this when explicitly asked to do, which will not happen in this course

Retrieving Models

Model classes are basically used as an advanced, more powerful query builder. The `Model` base class has a large collection of methods already implemented for this. This means that you can retrieve database rows by simply calling these methods from your Model class like:

```
$foobars = FooBar::where('flibble_floob', 'hufflepuf')->get();
```

All the Laravel query builder methods are allowed here.

Besides this, a more convenient method to retrieve the entire table is:

```
$allFoobars = FooBar::all();
```

It is also worth mentioning that queries return a collection of model class instances. So, each object in the result of the examples here are `FooBar` objects. This means that you can add methods to your class that can work with the models values.

From an OO perspective, a Model is therefore a class that represents a single row in a specific table of the database. It also contains static methods to retrieve objects from that same table, which functions basically the same as the Query Builder.

Inserting Models

Models can also be used to insert new rows. Eloquent makes this process, especially from an OO perspective, a breeze. Just create an object, populate its properties and then call some method to make it save itself to the database:

```
$newFooBar = new FooBar(); $newFooBar->snazzle_dazzle = 34; $newFooBar->flibble_floob = 'ephemeral'; $newFooBar->save();
```

Updating Models

This is also true for updating existing models. If you understand OOP, you can almost guess that you need to find an existing model instance, update its properties and the call a method so it can save itself.

```
$existingFooBar = FooBar::find(532); $existingFooBar->snazzle_dazzle = 78;
$existingFooBar->flibble_floob = 'whimsical'; $existingFooBar->save();
```

Deleting models

And finally, deleting an existing model:

```
$existingFooBar = FooBar::find(19); $existingFooBar->delete();
```

Digging deeper

The SHOULDs about this topic, described as links with some extra information to help understanding the content of the link

Mass assignment

In Laravel, mass assignment refers to the ability to assign multiple attributes of a model in a single line of code. It allows you to pass an array of data to the model's `make`, `create` or `update` method, enabling quick and convenient assignment of attributes.

For example, suppose you have a `User` model with attributes like `name`, `email`, and `password`. With mass assignment, you can assign values to these attributes like this:

```
// Creating a new user using mass assignment $user = User::create([ 'name'
=> 'John Doe', 'email' => 'john@example.com', 'password' => bcrypt('passwo
rd123'), ]);
```

In the above code, the `create` method takes an array where the keys correspond to the model's attributes. Laravel automatically handles assigning these attributes to the model and persists it in the database.

However, enabling mass assignment without proper safeguards could pose security risks. To prevent unintended mass assignment of sensitive attributes, Laravel provides a way to specify which attributes can be mass-assigned using the `$fillable` or `$guarded` properties within the model.

> ⚠️ Unless you know exactly what you are doing, we advise you to use `$fillable`. This minimizes the risk of unwanted fillable attributes as much as possible.

1. `$fillable`: You explicitly define an array listing the attributes that are allowed for mass assignment. Only these specified attributes can be filled using mass assignment.

Example:

```
class User extends Model { protected $fillable = ['name', 'email', 'passwo
rd']; }
```

1. `$guarded`: You specify an array of attributes that should not be mass-assigned. When using `$guarded`, all attributes not listed in the array are mass-assignable.

Example:

```
class User extends Model { protected $guarded = ['id', 'admin']; }
```

It's essential to use either `$fillable` or `$guarded` to protect your models from mass assignment vulnerabilities and explicitly define which attributes can be assigned en masse.

Derived attributes

Laravel calls them "value objects": transforming multiple model values into one. For example, if a user has attributes for `first_name` and `last_name`, you might want to add an extra attribute `full_name` that is not stored in the database, but transformed from the user's first and last name.

The simplest way of implementing this is by adding a method to you model:

```
class User extends Authenticatable { // ... /** * Returns the full name of this user * * @return string */ public function fullName(): string { return $this->name_first.' '.$this->name_last; } }
```

This can be called like: `$user->fullName()`. If you want to be more expressive, and want to do something like `$user->full_name`, check the Further Reading section.

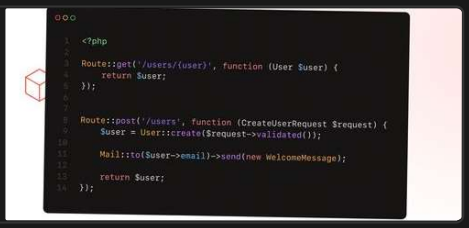
Further reading

The MAYs about this topic. Just inspiration and links, with the most important one here:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/eloquent



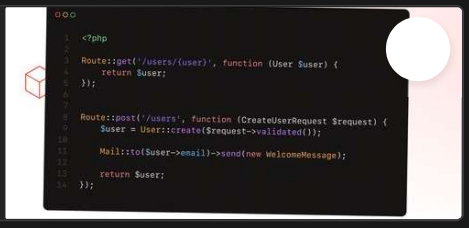
Derived attributes using accessors & mutators

As mentioned before, it is possible to define 'magic attributes' to an Eloquent model. Eloquent has a feature called **Mutators** to do this. You can read about this here:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/eloquent-mutators#accessors-an...



Local Query Scopes

It is possible to define commonly used queries that are used throughout your application as "Query Scopes". There are two ways of doing this and the most simple one is **Local Scopes.** You can read about them here:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/eloquent#local-scopes