Chapter 12 – Product Model

Introduction to Eloquent

Eloquent is a Laravel object-relational mapper (ORM) that makes it super easy to interact with our database. When using Eloquent, each database table has a corresponding "Model" used to interact with that table. Eloquent models allow you to insert, retrieve, update, and delete records from the database tables. More information about Laravel Eloquent can be found here: https://laravel.com/docs/9.x/eloquent.

Laravel models are located in the *app/Http/Models* folder. In that folder, you will find a *User.php* file that contains a Laravel predefined *User* class model. We will focus on our *Product* model for now, so let's create it.

Creating Product model

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan make:model Product
```

You will see the *Product.php* file inside the *app/Models* folder. Let's analyze the *Product* model code.

Analyze Code

```
<?php

namespace App \ Models;

use Illuminate \ Database \ Eloquent \ Factories \ HasFactory;
use Illuminate \ Database \ Eloquent \ Model;

class Product extends Model
{
      use HasFactory;
}
```

It seems to be a little empty. We have a *Product* class that extends a Laravel Eloquent *Model* class. The Laravel Eloquent *Model* class will provide our *Product* class with a set of useful methods and attributes. Those methods will make it super easy to communicate with the database. Therefore, it uses a *HasFactory* trait (but we will skip it).

We will discuss some issues with these "empty" models later.

Important considerations about Eloquent

Let's check some considerations when using Laravel Eloquent.

- Eloquent will also assume that each model's corresponding database table has a primary key column named *id* . For all our migrations, we will use the *id* method which defines an *id* column.
- Eloquent will assume the *Product* model stores records in the *products* table (check the additional 's'). This convention applies to all models.
- By default, Eloquent expects *created_at* and *updated_at* columns to existing on your model's corresponding database table. Eloquent will automatically set these column's values when models are created or updated. For all our migrations, we will use the *timestamps* method which creates these columns.

Key methods

Let's discuss some key methods that Eloquent provides to our models.

- ***Product::all()*** : retrieve all product records.
- ***Product::find(1)*** : retrieve the product with id 1.
- ***Product::findOrFail(1)*** **:** it is similar to the previous one, but it will throw an exception if no result is found.
- ***Product::create(['name' => 'TV', ...])*** **:** create a new record in the database. You must pass an associative array with the data to be assigned, *id* is not required since it is autogenerated.
- ***Product::destroy(1)*** **:** remove the product with id 1.

We will use some previous methods in upcoming chapters. For now, let's modify our application to extract the product data from our MySQL database.

Chapter 13 – List Products with Database Data

To extract data from the MySQL database, we only need to modify the *ProductController* .

Modifying ProductController

In *app/Http/Controllers/ProductController.php* , make the following changes in **bold**.

Modify Bold Code

```
<?php

namespace App \ Http \ Controllers;

use App \ Models \ Product;
use Illuminate \ Http \ Request;

class ProductController extends Controller
{
   public static $products = [
      ["id"=>"1", "name"=>"TV", "description"=>"Best TV", "image" => "game.png", "price"=>"1000"],
      ["id"=>"2", "name"=>"iPhone", "description"=>"Best iPhone", "image" => "safe.png", "price"=>"999"],
      ["id"=>"3", "name"=>"Chromecast", "description"=>"Best Chromecast", "image" => "submarine.png", "price"=>"30"],
      ["id"=>"4", "name"=>"Glasses", "description"=>"Best Glasses", "image" => "game.png", "price"=>"100"]
   ];

      public function index()
      {
            $viewData = [];
            $viewData["title"] = "Products - Online Store";
            $viewData["subtitle"] =  "List of products";
            $viewData["products"] = Product::all();
            return view('product.index')->with("viewData", $viewData);
      }

      public function show($id)
      {
            $viewData = [];
            $product = Product::findOrFail($id);
            $viewData["title"] = $product["name"]." - Online Store";
            $viewData["subtitle"] =  $product["name"]." - Product information";
            $viewData["product"] = $product;
            return view('product.show')->with("viewData", $viewData);
      }
}
```

Let’s analyze the previous code by parts.

Analyze Code

```
use App \ Models \ Product;
```

We use the *Product* model, which connects to the database table.

Analyze Code

```
   public static $products = [
      ["id"=>"1", "name"=>"TV", "description"=>"Best TV", "image" => "game.png", "price"=>"1000"],
      ["id"=>"2", "name"=>"iPhone", "description"=>"Best iPhone", "image" => "safe.png", "price"=>"999"],
      ["id"=>"3", "name"=>"Chromecast", "description"=>"Best Chromecast", "image" => "submarine.png", "price"=>"30"],
      ["id"=>"4", "name"=>"Glasses", "description"=>"Best Glasses", "image" => "game.png", "price"=>"100"]
   ];
```

We remove the *products* dummy attribute since we don’t need it anymore. We will instead retrieve the products data from the database.

Analyze Code

```
$viewData["products"] = Product::all();
```

We have the *Product::all()* in the *index* method, which retrieves the products from the database.

Analyze Code

```
$product = Product::findOrFail($id);
```

In the *show* method, we have the *Product::findOrFail($id)* , which retrieves a specific product based on its *id* . *findOrFail* could throw a *ModelNotFoundException* (i.e., when passing an invalid id). If the *ModelNotFoundException* is not caught, a 404 HTTP response is automatically sent back to the client.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan serve
```

Go to the (“/products”) route, and you will see the products retrieved from our MySQL database. Try to visit a specific product with an *id* that does not exist (i.e., “/products/21”). The application will show a 404 error (see Fig. 13-1).

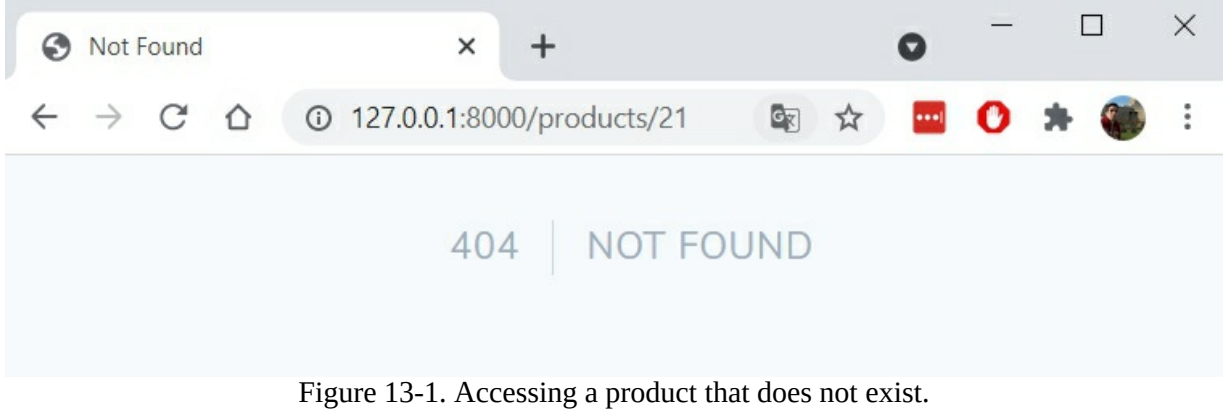

Figure 13-1. Accessing a product that does not exist.

Chapter 14 – Refactoring List Products

Many things can be improved in the previous code. For example, we will refactor our *Product* model, *ProductController* , and *product views* . These changes will make our code more maintainable, understandable, and clean. Again, many of these changes apply not only to a Laravel project but are also general tips that can be replicated in most MVC frameworks.

Model attributes

Open the *Product* model file and try to guess just by seeing that file which the *Product* model attributes are. We have a problem here. We cannot deduce the *Product* model attributes. We must look in our "create products table" migrations or open phpMyAdmin and go to the *products* table just to find that answer. In our opinion, the way the *Product* model is designed affects the project understandability. Many other MVC frameworks, such as Django (Python) or Spring (Java), explicitly define their models' product attributes, but that is not the Laravel case.

Let's refactor our *Product* model. In *app/Models/Product.php* , make the following changes in **bold**.

Modify Bold Code

```
<?php
namespace App \ Models;

use Illuminate \ Database \ Eloquent \ Factories \ HasFactory;
use Illuminate \ Database \ Eloquent \ Model;

class Product extends Model
{
    use HasFactory;
    /**
     * PRODUCT ATTRIBUTES
     * $this->attributes['id'] - int - contains the product primary key (id)
     * $this->attributes['name'] - string - contains the product name
     * $this->attributes['description'] - string - contains the product description
     * $this->attributes['image'] - string - contains the product image
     * $this->attributes['price'] - int - contains the product price
     * $this->attributes['created_at'] - timestamp - contains the product creation date
     * $this->attributes['updated_at'] - timestamp - contains the product update date
     */
}
```

Let's analyze the previous code.

- We removed the use of the *HasFactory* trait. Model factories are helpful to automate the creation of dummy model records in the database. For example, do you remember when we executed some SQL queries to insert some products into the database? Model factories can automate this process for us. However, model factories are out of the scope of this book. You can find more information about model factories here: https://laravel.com/docs/9.x/database-testing#defining-model-factories.
- Finally, we included a PHP comment block that specifies the available attributes for the *Product* model. A programmer who opens the *Product* model can easily find the available attributes.

We have declared the model's attributes in the form of *$this->attributes['id']* . It is because Laravel Eloquent stores the model's attributes in a class array attribute called *$attributes* .

Check the following code to understand how the Laravel Eloquent model's attributes work.

Analyze Code

```
$product = Product::findOrFail(1);
echo $product->name; # prints the product's name
echo $product["name"]; # prints the product's name
```

Laravel Eloquent provides two ways of accessing model attributes. The object attribute form (*$product->name*) and the associative array form (*$product["name"]*). Both options internally access the *Product* model and look for the *$this->attributes['name']* data. If that attribute does not exist, it returns *null* . Otherwise, it returns the value stored in the class array attribute (*$attributes*). Spoiler, we won't use any of these forms of accessing the model's attributes. We will use a better one, but first, let's see why we need a better way of accessing the model's attributes.

A project without getters and setters

Let's analyze our current code. It is an excerpt of our *ProductController.php* file.

Analyze Code

```
public function show($id)
{
    $viewData = [];
    $product = Product::findOrFail($id);
    $viewData["title"] = $product["name"]." - Online Store";
    $viewData["subtitle"] =  $product["name"]." - Product information";
    $viewData["product"] = $product;
    return view('product.show')->with("viewData", $viewData);
}
```

We are accessing the product's name through the associative array form (*$product["name"]*). Now, let's analyze the *product/show.blade.php* view.

Analyze Code

```
    <div class="card-body">
      <h5 class="card-title">
          {{ $viewData["product"]["name"] }} ({{ $viewData["product"]["price"] }})
      </h5>
      <p class="card-text">{{ $viewData["product"]["description"] }}</p>
      <p class="card-text"><small class="text-muted">Add to Cart</small></p>
    </div>
```

Again, we are accessing the product's name through the associative array form (*$viewData["product"]["name"]*). What is the problem with accessing the entity data this way? Imagine that your boss tells you, "We need to display all products' names in uppercase throughout the entire application". That is a big issue as we extract products' names over several different views and controllers. This simple requirement will require us to modify several views and controllers. For now, let's see what we must do to achieve that requirement.

We should modify the *ProductController.php* controller this way (**do not implement this change, it is used just to exemplify this scenario**):

Analyze Code

```
public function show($id)
{
    $viewData = [];
    $product = Product::findOrFail($id);
    $viewData["title"] = strtoupper($product["name"])." - Online Store";
    $viewData["subtitle"] =  strtoupper($product["name"])." - Product information";
    $viewData["product"] = $product;
    return view('product.show')->with("viewData", $viewData);
}
```

And the *product/show.blade.php* view this way (**do not implement this change, it is used just to exemplify this scenario**):

Analyze Code

```
    <div class="card-body">
      <h5 class="card-title">
          {{ strtoupper($viewData["product"]["name"]) }} ({{ $viewData["product"]["price"] }})
      </h5>
      <p class="card-text">{{ $viewData["product"]["description"] }}</p>
      <p class="card-text"><small class="text-muted">Add to Cart</small></p>
    </div>
```

There are two significant issues with this strategy. (i) We have many duplicate codes throughout the application (the *strtoupper* function is used over several places). And (ii) we must check all controllers, all views, and maybe dozens or hundreds of files to check where we need to apply the *strtoupper* function. It is not maintainable. So, let's refactor our code to implement a better strategy.

Project with getters and setters

Refactoring the Product model

First, let's refactor our *Product* model. In *app/Models/Product.php* , make the following changes in **bold**.

Modify Bold Code

```
...
class Product extends Model
{
    ...

    public function getId()
    {
        return $this->attributes['id'];
```

```
}

public function setId($id)
{
    $this->attributes['id'] = $id;
}

public function getName()
{
    return $this->attributes['name'];
}

public function setName($name)
{
    $this->attributes['name'] = $name;
}

public function getDescription()
{
    return $this->attributes['description'];
}

public function setDescription($description)
{
    $this->attributes['description'] = $description;
}

public function getImage()
{
    return $this->attributes['image'];
}

public function setImage($image)
{
    $this->attributes['image'] = $image;
}

public function getPrice()
{
    return $this->attributes['price'];
}

public function setPrice($price)
{
    $this->attributes['price'] = $price;
}

public function getCreatedAt()
{
    return $this->attributes['created_at'];
}

    public function setCreatedAt($createdAt)
    {
        $this->attributes['created_at'] = $createdAt;
    }

    public function getUpdatedAt()
    {
        return $this->attributes['updated_at'];
    }

    public function setUpdatedAt($updatedAt)
    {
        $this->attributes['updated_at'] = $updatedAt;
    }
}
```

For each *Product* attribute, we define its corresponding getter and setter. We will use getters and setters to access and modify our model attributes. It has many advantages which we will talk about later.

Refactoring the ProductController

In *app/Http/Controllers/ProductController.php* , make the following changes in **bold**.

Modify Bold Code

```
...
   public function show($id)
      {
            $viewData = [];
            $product = Product::findOrFail($id);
            $viewData["title"] = $product->getName()." - Online Store";
            $viewData["subtitle"] =  $product->getName()." - Product information";
            $viewData["product"] = $product;
            return view('product.show')->with("viewData", $viewData);
      }
}
```

Now, we access the product attributes through the corresponding getters and setters.

Refactoring the product/index view

In *resources/views/product/index.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
...
<div class="row">
   @foreach ($viewData["products"] as $product)
   <div class="col-md-4 col-lg-3 mb-2">
      <div class="card">
         <img src="{{ asset('/img/'.$product->getImage()) }}" class="card-img-top">
         <div class="card-body text-center">
            <a href="{{ route('product.show', ['id'=> $product->getId()]) }}"
               class="btn bg-primary text-white">{{ $product->getName() }}</a>
         </div>
      </div>
   </div>
   @endforeach
</div>
...
```

Like the previous controller, we access the product attributes through the corresponding getters.

Refactoring the product/show view

In *resources/views/product/show.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
...
   <div class="row g-0">
      <div class="col-md-4">
         <img src="{{ asset('/img/'.$viewData["product"]->getImage()) }}" class="img-fluid rounded-start">
      </div>
      <div class="col-md-8">
         <div class="card-body">
            <h5 class="card-title">
              {{ $viewData["product"]->getName() }} (${{ $viewData["product"]->getPrice() }})
         </h5>
            <p class="card-text">{{ $viewData["product"]->getDescription() }}</p>
            <p class="card-text"><small class="text-muted">Add to Cart</small></p>
         </div>
      </div>
   </div>
...
```

We access the *product* attributes through getters.

Analyzing getters and setters

For now, the application looks the same. We only modified the way we access model attributes. So, what is the advantage? Let's revisit the boss requirement: "we need to display all products' names in uppercase over the entire application".

In this case, we only need to modify the *Product* model file. Specifically, the *getName* method. Let's see the modification (**you can apply the following change or leave it as it is**).

Analyze Code

```
...
      public function getName()
      {
            return strtoupper($this->attributes['name']);
      }
...
```

If you run the application, you will see that all products' names appear in uppercase. We only required one single

change in one specific location. That is the power of the use of getters or setters. **The definition and use of getters and setters guarantee a unique access point to the model attributes**. That is part of what some people call encapsulation, one of the three pillars of object-oriented programming.

TIP: Always try to access your model attributes through getters and setters. **It will make it easier to add functionalities in the future**. You can even include a new rule saying that models' attributes must be accessed through their corresponding getters and setters (in your architectural rules document).

Quick discussion: Laravel provides another way to implement getters and setters. Laravel calls them Accessors and Mutators (you can read more about it here: https://laravel.com/docs/9.x/eloquent-mutators#accessors-and-mutators). We don't prefer this strategy for a single reason. When you use Accessors and Mutators and access to the model attributes, you continue using the original form (i.e., *$product->name*) versus the classic getter form (*$product->getName()*). We prefer the second one because it explicitly says that you access the data through a class method. It improves the code's understandability. Otherwise, programmers must go to the model file to check if an Accessor is implemented.

We will use classic getters and setters for the rest of the application. We hope you understand their importance now.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan serve
```

The application should be properly working. If you applied the modification in the *getName* method, you would see all products' names in uppercase (see Fig. 14-1).

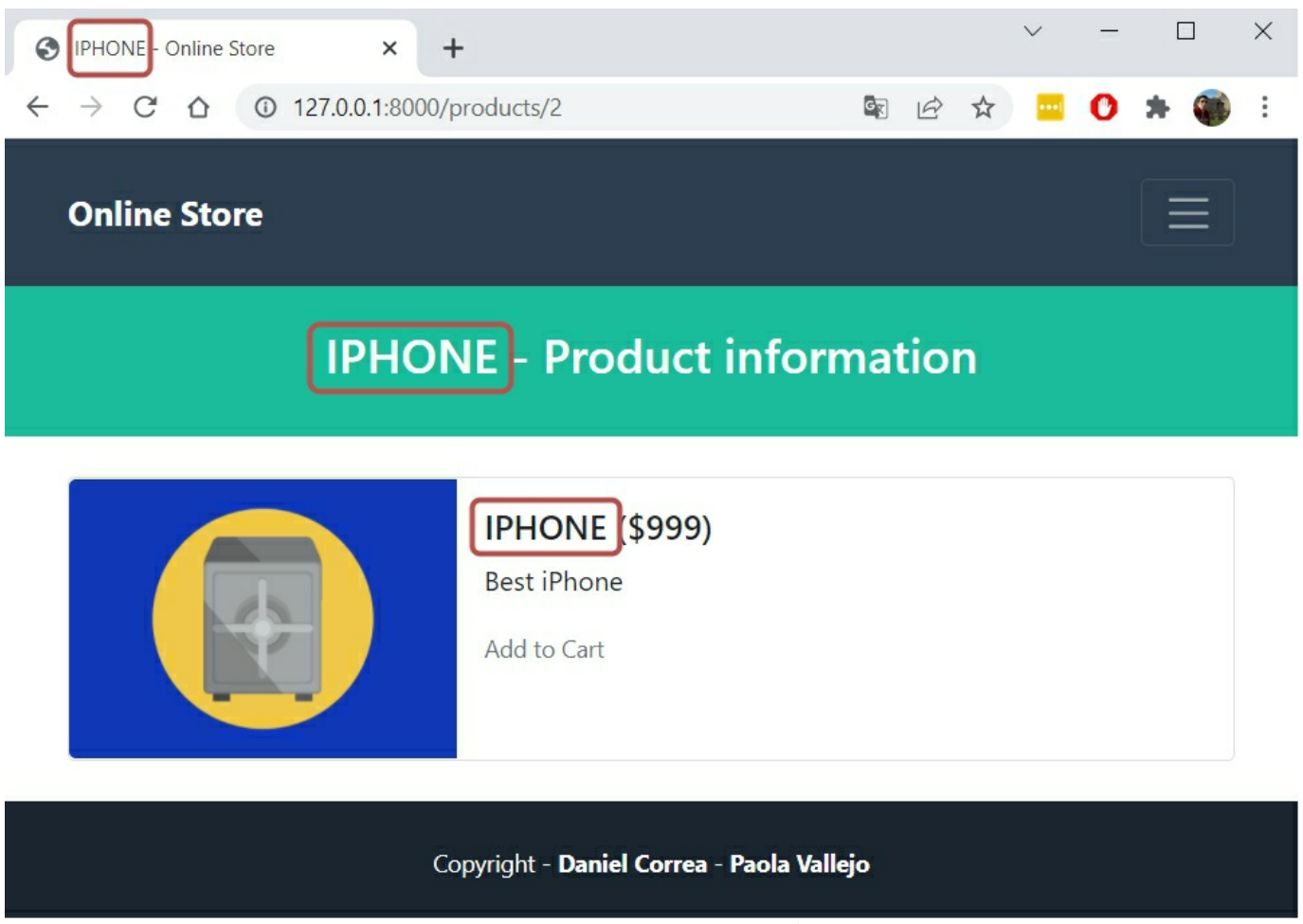

Figure 14-1. Accessing a product with the modified *getName* method.