Practical Laravel

Develop clean MVC web applications

Daniel Correa - Paola Vallejo

Practical Laravel

Develop clean MVC web applications

Daniel Correa – Paola Vallejo

Practical Books

Copyright © 2022 by Daniel Correa
All Rights Reserved

Practical Laravel

by Daniel Correa and Paola Vallejo
Copyright © 2022 by Daniel Correa. All rights reserved.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the author's prior written permission, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made to prepare this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author or its distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Technical editor: Andrés Felipe Pineda.

Code reviewer: Simón Flores.

First Edition: February 2022.

Contributors

About the author

Daniel Correa has been a researcher and a software developer for several years. Daniel has a Ph.D. in Computer Science; currently, he is a professor at Universidad EAFIT in Colombia. He is interested in software architectures, frameworks (such as Laravel, Django, Nest, Express, Vue, React, Angular, and many more), web development, and clean code.

Daniel is very active on Twitter. He shares tips about software development and reviews software engineering books. Contact Daniel on Twitter at **@danielgarax**.

About the co-author

Paola Vallejo is a professor and researcher at Universidad EAFIT in Colombia. She is interested in software architectures, software design principles, software design patterns, and clean code. Learn more about Paola's research interests at: https://scholar.google.com/citations?user=S8xNhVoAAAAJ.

About the technical editor

Andrés Felipe Pineda is a software developer with more than eight years developing full-stack applications mainly using PHP and Node. Andrés has worked as a professor in different universities in Medellín. His main areas of interest are design patterns, competitive programming, and databases. Andrés work in several educational communities to create tools and technologies for improving the students' performance in programming courses. Contact him at afpinedac@gmail.com.

About the code reviewer

Simón Flores is a Systems Engineering student with a great passion for web development. He is always searching for new ways to improve his skills (https://github.com/sflorezs1).

Table of Contents

Preface
Chapter 01 – Introduction
Chapter 02 – Online Store Running Example
Chapter 03 – Introduction to Laravel and Installation
Chapter 04 – Introduction to MVC applications
Chapter 05 – Layout View
Chapter 06 – Index and About Pages
Chapter 07 – Refactoring Index and About Pages
Chapter 08 – Use of a Coding Standard
Chapter 09 – List Products with Dummy Data
Chapter 10 – Configuration of MySQL Database
Chapter 11 – Product Migration
Chapter 12 – Product Model
Chapter 13 – List Products with Database Data
Chapter 14 – Refactoring List Products
Chapter 15 – Admin Panel
Chapter 16 – List Products in Admin Panel
Chapter 17 – Create Products
Chapter 18 – Create Products with Images
Chapter 19 – Edit and Delete Products
Chapter 20 – Refactoring Validations
Chapter 21 – Login System
Chapter 22 – Refactoring User
Chapter 23 – AdminAuthMiddleware
Chapter 24 – Introduction to Web Session
Chapter 25 – Shopping Cart
Chapter 26 – Orders and Items
Chapter 27 – Product Purchase
Chapter 28 – Orders Page
Chapter 29 – Deploying to the Cloud – Clever-Cloud – MySQL Database
Chapter 30 – Deploying to the Cloud – Heroku – Laravel Application
Chapter 31 – Continue your Laravel Journey

Preface

Laravel is a PHP web application framework with expressive and elegant syntax. We will use Laravel to develop an Online Store application which uses several Laravel features. The Online Store application will be the means to understand straightforward and complex Laravel concepts and how Laravel features can be used to implement real-world applications.

The main difference between this book and other similar books is that this book is not just about Laravel. Instead, this book is about a “clean” design and implementation of web applications using Laravel. By ‘clean’, we refer to an understandable, maintainable, usable, and well-divided application.

The authors have developed several applications over many frameworks, including Laravel, Django, Express, Flask, Nest, Spring, Vue, Angular, and React. Moreover, we are going to use that knowledge to create a clean design and clean code strategies that can be applied not just to Laravel, but to the design and implementation of most web applications using frameworks such as Django, Nest, Flask, Express, and more.

This book is written with brief explanations direct to the point. It includes tips, short discussions, and useful phrases found in other books that we have read to provide you with a practical approach that will make improve your coding skills.

This is a short book divided into 31 chapters, with six pages on average per chapter. It was designed not to overwhelm you. With this division, you will feel like you are making fast progress. We won’t cover all Laravel features, but some of the most important to develop MVC web applications.

We hope you enjoy this journey as we did when we wrote this book.

Who is this book for?

This book is for web developers or programmers who want to learn Laravel and improve their code skills. No previous knowledge of Laravel is required. However, basic programming knowledge is required. This book is also suitable for experienced Laravel developers. They can revise previous concepts and learn new clean code strategies.

Download the example code files

You can download the example code files from the GitHub repository https://github.com/PracticalBooks/Practical-Laravel. In it, you will find the code of each chapter. You can replicate this book’s code or download the code directly from GitHub. If there is an update to the code, it will be updated on the existing GitHub repository.

Questions and discussions

If you have questions about any aspect of this book or want to discuss something, we recommend you use the discussion zone of the GitHub repository (see Fig. P-1). In that way, you can learn from other questions, and we can learn from you. Besides, others in the community can answer your questions.

Figure P-1. Discussion zone of the GitHub repository.

Additionally, you can email your questions to practicalbooksco@gmail.com. Please mention the book title in the subject of your message.

Download colored images

We also provide a PDF file with colored images of the figures/diagrams used in this book. You can download it here: https://github.com/PracticalBooks/Practical-Laravel/blob/main/BookImages/Book%20images.pdf.

Getting book updates

If you want to receive book updates, please email us at practicalbooksco@gmail.com. We will also subscribe you to our mailing list.

Chapter 01 – Introduction

We will begin our journey to understand and apply many Laravel concepts and features to develop MVC web applications.

The book is divided into the following chapters. We will highlight the Laravel concepts we will learn and the features and tools we will use across the chapters.

- **Chapter 01 – Introduction.**
- **Chapter 02 – Online Store running example.**
- **Chapter 03 – Introduction to Laravel and Installation:** Laravel, XAMPP, Artisan, and Composer.
- **Chapter 04 – Introduction to MVC applications.**
- **Chapter 05 – Layout View:** Blade, Bootstrap, and views.
- **Chapter 06 – Index and About Pages:** views, routes, and controllers.
- **Chapter 07 – Refactoring Index and About Pages.**
- **Chapter 08 – Use of a Coding Standard:** PHP_CodeSniffer.
- **Chapter 09 – List Products with Dummy Data**
- **Chapter 10 – Configuration of MySQL Database:** MySQL and phpMyAdmin.
- **Chapter 11 – Product Migration:** migrations.
- **Chapter 12 – Product Model:** models and Eloquent.
- **Chapter 13 – List Products with Database Data.**
- **Chapter 14 – Refactoring List Products.**
- **Chapter 15 – Admin Panel.**
- **Chapter 16 – List Products in Admin Panel.**
- **Chapter 17 – Create Products:** forms and validations.
- **Chapter 18 – Create Products with Images:** storage.
- **Chapter 19 – Edit and Delete Products.**
- **Chapter 20 – Refactoring Validations.**
- **Chapter 21 – Login System:** laravel/ui.
- **Chapter 22 – Refactoring User:** Tinker.
- **Chapter 23 – AdminAuthMiddleware:** Middleware and authentication.
- **Chapter 24 – Introduction to Web Session:** Laravel sessions.
- **Chapter 25 – Shopping Cart.**
- **Chapter 26 – Orders and Items:** Eloquent relationships.
- **Chapter 27 – Product Purchase.**
- **Chapter 28 – Orders Page:** eager loading and debugbar.
- **Chapter 29 – Deploying to the Cloud – Clever-Cloud – MySQL Database:** Clever-Cloud.
- **Chapter 30 – Deploying to the Cloud – Heroku – Laravel Application:** Heroku.
- **Chapter 31 – Continue your Laravel Journey**

In this book, we will develop an Online Store. This Online Store will serve us to understand some of the more important Laravel concepts. Figures 1-1, 1-2, 1-3, and 1-4 show the kind of application we will develop.

Figure 1-1. List of products page.

Figure 1-2. Product page.

Figure 1-3. Shopping cart page.

Figure 1-4. Admin panel page.

Let’s start our journey!

Chapter 02 – Online Store Running Example

Using a running example is a common strategy in programming books. A running example is an example where we visit repeatedly throughout the book. It provides a practical way to illustrate the concepts of a methodology, process, tool, or technique. In this case, we define an Online Store running example.

Online Store is a web application where users place orders to buy products.

Let's define the application scope for the app.

- **Home page** will display a welcome message and some images.
- **About page** will display information about the online store and developers.
- **Products page** will display the available products information. In addition, you can click on a specific product and see its information.
- **Cart page** will display the products added to the cart and the total price to be paid. In addition, a user can remove products from the cart and make purchases.
- **Login page** will display a form to allow users to log in to the application.
- **Register page** will display a form to allow users to sign up for accounts.
- **My orders page** will display the orders placed by the logged in user.
- **Admin panel** will contain sections to manage the store's products (create, update, delete, and list them).

The Online Store will be implemented with Laravel (PHP), MySQL database, Bootstrap (a CSS framework), and Blade (a Laravel templating system). We will learn about these elements in the upcoming chapters.

Below is a class diagram illustrating the application scope and design (see Fig. 2-1). We have a *User* class with its data (id, name, email, password, etc.) which can place *Orders* . Each *Order* is composed of one or more *Items* that are related to a single *Product* . Each *Product* will have its corresponding data (id, name, description, image, etc.).

Figure 2-1. Online Store class diagram.

This book is not about class diagrams, so we won't explain other details in the class diagram. You will see a relationship between the code and this diagram as you advance through the book. This diagram serves as a blueprint for the construction of our application.

TIP: Designing a class diagram before starting to code helps us understand the application's scope and identify important data. It also helps us know how the application elements are related. You can share a diagram like this with your team or colleagues, obtain quick feedback, and make adjustments as needed. Since it is a diagram, changes can be made quickly. Else, when the project has been coded, the replacement cost will be higher to move data from one class to another. Let's check this phrase from *(2015 – Newman, S. - Building microservices)* book. *"I tend to do much of my thinking in the place where the cost of change and the cost of mistakes is as low as it can be: the whiteboard."*

Now that we considered the kind of application we want to build, let's next understand what Laravel is and how to install it.

Chapter 03 – Introduction to Laravel and Installation

Introduction to Laravel

Laravel is a free and open-source PHP framework that provides a set of tools and resources to build modern PHP web applications (https://laravel.com/). Laravel provides an elegant syntax, built-in features, and various compatible packages and extensions.

Laravel aims to make the development process a pleasing one for the developer without sacrificing application functionality. *"Happy developers make the best code"* (written in the Laravel documentation).

When writing this book, the latest version is Laravel 9, which we will use to build our Online Store application.

Note: a new Laravel version might be available at the time you are reading this book. We recommend you continue using Laravel 9 for this project. Once you complete this book, you can upgrade to the latest Laravel version. In this way, most of the code will remain reusable. Some others might require minor adjustments.

Requirements (XAMPP and Composer)

There are several ways to install Laravel. You can check some of them here: https://laravel.com/docs/9.x#your-first-laravel-project. We will use a local installation for this book, which requires installing XAMPP and Composer.

XAMPP

XAMPP is the most popular PHP development environment. XAMPP is a free, easy to install Apache distribution containing MySQL, PHP, and Perl. If you don't have XAMPP installed, go to https://www.apachefriends.org/download.html. Download and install it. **Be careful, you will need to download and install a XAMPP version which supports PHP 8** because Laravel 9 requires PHP 8.

Once XAMPP is installed, go to the Terminal, and execute the following command. **Note:** if *PHP is not recognized as an internal or external command*, it means you need to add the XAMPP PHP installation folder to your PATH environment variable. A simple search in Google will help you to solve this.

Execute in Terminal

```
php --version
```

If the installation was successful, you would see a result as presented in Fig 3-1. **Please check you have PHP 8.* installed**.

```
PS C:\Users\yo> php --version
PHP 8.0.11 (cli) (built: Sep 21 2021 18:26:06) ( ZTS Visual C++ 2019 x64 )
Copyright (c) The PHP Group
Zend Engine v4.0.11, Copyright (c) Zend Technologies
```

Figure 3-1. Checking PHP version.

Composer

Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on, and it will manage (install/update) them for you. If you don't have Composer installed, go to https://getcomposer.org/download/. Download and install it.

Once Composer is installed, go to the Terminal, and execute the following command.

Execute in Terminal

```
composer --version
```

If the installation was successful, you would see a result as presented in Fig 3-2.

```
PS C:\Users\yo> composer --version
Composer version 2.1.12 2021-11-09 16:02:04
```

Figure 3-2. Checking composer version.

Create a new Laravel Project (using Composer)

Open your Terminal, and in a location of your choice (you can use the *xampp/htdocs/* location if you want), execute the following:

Execute in Terminal

```
composer create-project laravel/laravel onlineStore "9.*" --prefer-dist
```

The previous command creates a new Laravel 9.* project inside the *onlineStore* folder. Next, in your Terminal, move to the *onlineStore* folder, and run the application with the following:

Execute in Terminal

```
cd onlineStore
php artisan serve
```

The *php artisan serve* command starts Laravel's server (see Fig. 3-3). **Artisan** is the command line interface included with Laravel. Artisan exists at the root of your application as the artisan script and provides helpful commands to assist you while you build your application. More information about Laravel Artisan can be found here: https://laravel.com/docs/9.x/artisan.

```
PS C:\xampp\htdocs\onlineStore> php artisan serve
Starting Laravel development server: http://127.0.0.1:8000
[Wed Feb  9 11:04:59 2022] PHP 8.0.11 Development Server (http://127.0.0.1:8000) started
```

Figure 3-3. Running Laravel project.

If the installation and setup were successful, you could open the Laravel development server link in your browser (http://127.0.0.1:8000/). You should see your Laravel 9 application as shown in Fig. 3-4.

Figure 3-4. Laravel 9 default page.

Note: you can stop the server with *Ctrl + C* (on Windows) or *Cmd + C* (on Mac).

Laravel Project Structure

Fig. 3-5 shows the Laravel project structure. We won't explain all the folders and files since we want to start developing our web applications quickly. We will explain some of the more important ones. The others will be covered in upcoming chapters.

Figure 3-5. Laravel Project structure.

- **app/Http/Controllers/***: we will place the app controllers here.
- **app/Models/*:** we will place the app models here.
- **database/migrations/*:** we will define the app migrations (the app's database schema definition) here.
- **public/*:** we will store our CSS, JavaScript, and images files here. The *public* folder also contains the *index.php* file, which is the entry point to the application.
- **resources/views/*:** we will place the app views here.
- **routes/web.php:** the *web.php* file will contain all the route definitions for the web application.
- **storage/app/public/*:** here, we will store the user-generated files, such as product images, that should be publicly accessible.
- **vendor/*:** The */vendor* folder contains all libraries downloaded from Composer. The libraries/dependencies are listed in the *composer.json* file.
- **.env:** contains some common configuration values that may differ based on whether your application is running locally or on a production web server. It includes information such as database name, database username, and database password, among others.
- **composer.json:** holds metadata relevant to the project and manages the project's dependencies, scripts, version, and many more.

Quick discussion: Laravel is an **opinionated framework**. It means that it comes with most of the parts you need to build an application. It defines a project structure, defines an architecture, contains a lot of libraries and helpers to deal with database management, authentication, web session, and so on. The advantage is that a developer can implement web applications very quickly. However, performance could not be the best, and you can have a significant number of folders and files which can be overwhelming to understand. On the other side, you have **unopinionated frameworks** (such as Express). Express (a Node.js framework) comes with limited functionalities, and even it does not define a project structure and an architecture. The advantage is that performance is increased. However, a web developer should take many critical decisions (such as defining the application architecture) and deal with the inclusion of third-party libraries (such as a library to connect and manage the database).

In the next chapter, we will discuss the application architectural pattern.

Chapter 04 – Introduction to MVC applications

There are different ways of designing and implementing web applications. For example, you can create an entire web application by placing your code in a single file. However, finding an error in such a file (which contains thousands of lines of code) is not an easy task. Other approaches split the code over different files and folders. You will even find approaches that split your application over different small applications distributed over several servers (distribution of these servers is not an easy task).

As you can see, structuring your code is not an easy task. That is the reason why developers and computer scientists have developed what are called software architectural patterns. **Software architectural patterns** are structural layouts used to solve commonly faced software design problems. With these patterns, startups and novice developers don't have to "reinvent the wheel" each time they start a new project. There are many architectural patterns, such as model-view-controller, layers, service-oriented, and micro-services. Each one has its advantages and disadvantages. Many are widely adopted. Still, one of the most used is the model-view-controller pattern.

Model-view-controller (MVC) is a software architectural pattern commonly used to develop web applications containing user interfaces. This pattern divides the application into three interconnected elements.

- **Model** contains the business logic of the application. For example, the Online Store application product data and its functions.
- **View** contains the application's user interface. For example, a view to register products or users.
- **Controller** acts as an interface between model and view elements. For example, a product controller collects information from a "create product" view and passes it to the product model to be stored in the database.

Laravel provides support for the MVC pattern thanks to the integration of the Blade templating engine. Other similar frameworks provide support to this popular pattern too. We will see this pattern in action (with actual code) later.

The MVC pattern provides some advantages: better code separation, multiple team members can work and collaborate simultaneously, finding an error is easier, and maintainability is improved. Fig. 4-1 shows the Online Store software architecture we will implement in this book. It can be a little overwhelming now, but you will understand the elements of this architecture when you finish this book. We will review the architecture in the final chapters.

Figure 4-1. Online Store software architecture.

Let's have a quick analysis of this architecture:

- On the left, we have clients (users of our application e.g., browsers in mobile/desktop devices). Clients connect to the application through the Hypertext Transfer Protocol (HTTP). HTTP gives users a way to interact with our web application.
- On the right, we have the server where we place our application code.
- All client interactions first pass for a route file called *web.php* (described in Chapter 6).
- The *web.php* file passes the interaction to a controller (described in Chapter 6).
- Controllers communicate with models (Chapter 12) and pass information to the views (described in Chapter 5), which are finally delivered to the clients as HTML, CSS, and JavaScript code.

We highlight the Model, View, and Controller layers in grey. We have four models (entities) corresponding to the classes defined in our class diagram (in Fig. 2-1). As mentioned, there are different approaches to implement web applications with Laravel. There are even different versions of MVC used in a Laravel application. In the following chapters, we will see the advantages of adopting the MVC architecture defined in Fig. 4-1.

Chapter 05 – Layout View

Introducing Blade

Blade is a powerful templating engine that is included with Laravel. Blade template files use the *.blade.php* file extension and are typically stored in the *resources/views* directory. In your blade files, you will have a mix of HTML code with Blade directives and Blade elements. Blade directives are convenient shortcuts for common PHP control structures, such as conditional statements and loops.

For example, the following code shows an excerpt of a simple view in Laravel using plain PHP.

Analyze Code

```
<?php if($records > 0) { ?>
   I have records!
<?php } else { ?>
   I don't have any records!
<?php } ?>
```

The same view, but with Blade directives, is presented next. It looks cleaner.

Analyze Code

```
@if (count($records) > 0)
   I have records!
@else
   I don't have any records!
@endif
```

Quick discussion: You don't need a templating engine in PHP projects. You can mix app logic code (PHP) with app view code (HTML) in any PHP file. However, this multi-language combination is not supported for other languages such as Java or Python. So, why do PHP frameworks use templating engines? The main reason is to avoid using PHP syntax or PHP tags inside your view files. Instead, you should use the templating engine directives or helpers. What is the advantage? The template engines limit the number of available functionalities implemented in views (to provide a proper code separation). For example, with PHP and HTML you can create a database connection or even a PHP class inside a view file (which is crazy because views should not be responsible for creating database connections or classes). So, template engines ensure that you won't make crazy things in your views. Our recommendation is: if you don't find a directive or helper for a functionality you need to implement in a view file, it is because that functionality should not be implemented in the view (maybe it should be implemented in a controller or in another file).

TIP: Do not use plain PHP code in your views. Blade allows it, but please do not do it. Blade contains a *@php* directive that will enable you to inject plain PHP code. However, only use it as your last resort. We have developed complex Laravel web applications without the use of that directive.

Introducing Bootstrap

Bootstrap is the most popular CSS framework for developing responsive and mobile-first websites (see Fig. 5-1). For Laravel projects, a developer can design the user interface from scratch if he wants to. But because this book is not about user interfaces, we will take advantage of CSS frameworks (such as Bootstrap) and use a starter template to create something that looks professional. You can find out more about Bootstrap at: https://getbootstrap.com/.

Figure 5-1. Bootstrap website.

Introducing Blade Layouts

Most web applications maintain the same general layout across various pages (common header, navigation bar, and footer). However, maintaining our application would be incredibly cumbersome if we had to repeat the entire header, navbar and footer HTML in every view. Fortunately, we can define this layout as a single Blade file and use it throughout our application.

Creating app.blade.php

To get started with Bootstrap and the Blade layout, we first create a folder called *layouts* under the *resources/views* directory. We then use the Bootstrap starter template to create our layout (see Fig. 5-2). The Bootstrap starter template can be found here: https://getbootstrap.com/docs/5.1/getting-started/introduction/.

Figure 5-2. Bootstrap starter template.

In *resources/views/layouts* , create a new file *app.blade.php* and fill it with the following code.

Add Entire Code

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.min.css" rel="stylesheet" crossorigin="anonymous" />
  <title>@yield('title', 'Online Store')</title>
</head>
<body>
  <!-- header -->
  <nav class="navbar navbar-expand-lg navbar-dark bg-secondary py-4">
    <div class="container">
      <a class="navbar-brand" href="#">Online Store</a>
      <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNavAltMarkup"
        aria-controls="navbarNavAltMarkup" aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
        <div class="navbar-nav ms-auto">
          <a class="nav-link active" href="#">Home</a>
          <a class="nav-link active" href="#">About</a>
        </div>
      </div>
    </div>
  </nav>


  <header class="masthead bg-primary text-white text-center py-4">
    <div class="container d-flex align-items-center flex-column">
      <h2>@yield('subtitle', 'A Laravel Online Store')</h2>
    </div>
  </header>
  <!-- header -->


  <div class="container my-4">
    @yield('content')
  </div>


  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/js/bootstrap.bundle.min.js" crossorigin="anonymous">
  </script>
</body>
</html>
```

The above code is based on the Bootstrap starter template code and the Bootstrap Navbar (https://getbootstrap.com/docs/5.1/components/navbar/). We modified the base code, including links in the header

(*Home* and *About*). The starter template includes a Bootstrap CSS file (*bootstrap.min.css*), and a Bootstrap JS file (*bootstrap.bundle.min.js*). We included three *@yield* Blade directives.

@yield is used as a marker. We will inject code in those markers from child Blade views (using the *@section* directive). *@yield* uses two parameters, the first is the marker identifier. The second is a default value that will be injected if a child view does not inject code for that marker.

Modifying welcome.blade.php

Delete all the existing code in *resources/views/welcome.blade.php* and fill it with the following code.

Replace Entire Code

```
@extends('layouts.app')
@section('title', 'Home Page - Online Store')
@section('content')
<div class="text-center">
  Welcome to the application
</div>
@endsection
```

The *welcome* view extends the *layouts.app* view. This view injects the *'Home Page - Online Store'* message in the *@yield('title')* of the *layouts.app* and injects an HTML div with a welcome message inside the *@yield('content')* of the *layouts.app* .

Running the app

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

Execute in Terminal

```
php artisan serve
```

Open the browser to http://127.0.0.1:8000/, and you will see the application with the new layout (see Fig. 5-3). If you reduce the browser window width, you will see a responsive navbar (thanks to the bootstrap starter template, navbar, and the inclusion of the bootstrap files, see Fig. 5-4).

Figure 5-3. Application home page with the layout.

Figure 5-4. Application home page with reduced browser window width.

Adding custom CSS styles and a Footer

Let's make our app interface more professional. We will include a custom CSS file and a footer in our layout.

Custom style (app.css)

Create a folder *css* under the *public/* directory. Then, in *public/css* , create a new file *app.css* and fill it with the following.

Add Entire Code

```
.bg-secondary {
  background-color: #2c3e50 !important;
}

.copyright {
  background-color: #1a252f;
}

.bg-primary {
  background-color: #1abc9c !important;
}

nav{
  font-weight: 700;
}

.img-card{
  height: 18vw;
  object-fit: cover;
}
```

We have some custom CSS styles in the previous file. We override some Bootstrap elements with our values and colors.

Modifying app.blade.php

Finally, in *resources/views/layouts/app.blade.php* , make the following changes in **bold** to include the previous CSS file and create the footer section.

Modify Bold Code

```
<!doctype html>
<html lang="en">
<head>
   ...
   <link href="{{ asset('/css/app.css') }}" rel="stylesheet" />
   <title>@yield('title', 'Online Store')</title>
</head>
<body>
   …
  <!-- footer -->
  <div class="copyright py-4 text-center text-white">
    <div class="container">
      <small>
        Copyright - <a class="text-reset fw-bold text-decoration-none" target="_blank"
          href="https://twitter.com/danielgarax">
          Daniel Correa
        </a> - <b>Paola Vallejo</b>
      </small>
    </div>
  </div>
  <!-- footer -->
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/js/bootstrap.bundle.min.js" crossorigin="anonymous">
  </script>
</body>
</html>
```

Laravel includes a variety of **global helper PHP functions**. For example, the *asset* function generates a URL for an asset using the current scheme of the request (HTTP or HTTPS). Since our *css/app.css* file is inside the *public* folder, it will be automatically deployed over our server link (i.e., http://127.0.0.1:8000/css/app.css). We used curly braces *{{ }}* to invoke the *asset* function. Curly braces are used in Blade files to display data passed to the view or invoke Laravel helpers. In the end, we created a footer section with the book's author names and links to their Twitter accounts.

Running the app

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

Execute in Terminal

```
php artisan serve
```

You will see our home page, with the refined layout (see Fig. 5-5). It looks more professional (at least for us). This

design was inspired by a free Bootstrap template called Freelancer. You can find more information about this template here: https://startbootstrap.com/theme/freelancer.

Figure 5-5. Application home page with the refined layout.

We will refine the welcome page and create an about page in the next chapter.

Chapter 06 – Index and About Pages

Index view

The home page currently just displays a welcome message. Let's create a new *index* view. In *resources/views/*, create a subfolder *home*. In *resources/views/home*, create a new file *index.blade.php* and fill it with the following code.

Add Entire Code

```
@extends('layouts.app')
@section('title', $viewData["title"])
@section('content')
<div class="row">
   <div class="col-md-6 col-lg-4 mb-2">
      <img src="{{ asset('/img/game.png') }}" class="img-fluid rounded">
   </div>
   <div class="col-md-6 col-lg-4 mb-2">
      <img src="{{ asset('/img/safe.png') }}" class="img-fluid rounded">
   </div>
   <div class="col-md-6 col-lg-4 mb-2">
      <img src="{{ asset('/img/submarine.png') }}" class="img-fluid rounded">
   </div>
</div>
@endsection
```

The first *@section* injects the content of the *viewData["title"]* variable. That variable will be defined in the *web* route file (presented later in this chapter) and passed to this view. Then, we define some divisions to display some images. We need to download these images and store them in our *public* folder. First, in the *public* folder, create a subfolder *img*. Then, download the following three images from this link https://github.com/PracticalBooks/Practical-Laravel/tree/main/Chapter06/onlineStore/public/img and store them inside the *public/img* folder (see Fig. 6-1).

Figure 6-1. Project structure after storing the images.

About view

Let's create the *About* view. In *resources/views/home*, create a new file *about.blade.php* and fill it with the following code.

Add Entire Code

```
@extends('layouts.app')
@section('title', $title)
@section('subtitle', $subtitle)
@section('content')
<div class="container">
   <div class="row">
      <div class="col-lg-4 ms-auto">
         <p class="lead">{{ $description }}</p>
      </div>
      <div class="col-lg-4 me-auto">
         <p class="lead">{{ $author }}</p>
      </div>
   </div>
</div>
@endsection
```

We have a simple view that displays a description of the application and some information about the author. We will pass four variables from a controller to this view later. Remember that Blade allows curly braces to display data passed to the view.

Introducing Laravel Routing

Laravel Routing is a mechanism used to route all your application requests to specific methods or functions which will deal with those specific requests. Laravel routes accept a URI (Uniform Resource Identifier) along with a closure. Closures are PHP's version of anonymous functions. A closure is a function you can pass around as an object, assign to a variable, or pass as a parameter to other functions and methods.

Laravel routes are defined in your route files (located in the *routes* directory).

- The *routes/web.php* file defines routes for your web interface. These are the routes that we will use in this book.
- The *routes/api.php* file defines routes for your API (if you have one). These are routes used in service-oriented architectures or REST APIs (which is out of the scope of this book).

Let's check a couple of routes to understand how they work. In *routes/web.php* , make the following changes in **bold**.

Modify Bold Code

```
…
Route::get('/', function () {
    return view('welcome');
    $viewData = [];
    $viewData["title"] = "Home Page - Online Store";
    return view('home.index')->with("viewData", $viewData);
});
Route::get('/about', 'App \ Http \ Controllers \ HomeController@about')->name("home.about");
```

In the above code, we presented two ways of defining Laravel routes.

- The first route connects the "/" URI with a closure that returns a view (in this case, the *home.index* view). *view()* is a Laravel helper which retrieves a view instance. Check how we pass the *viewData* variable to the *home.index* view by chaining the *with* method onto the *view* helper method.
- The second route connects the "/about" URI with the *HomeController about* method (created later). Besides, we define a custom route name by chaining the *name* method onto the route definition.

Introducing Laravel Controllers

Defining all your request handling logic inside in your route files' closures does not seem smart. You will end with hundreds or thousands of code lines inside the route files (which affects the project maintainability). A good strategy is to organize this behavior using "controller" classes. Controllers can group related request handling logic into a single class. For example, a *UserController* class might handle all incoming requests related to users, including showing, creating, updating, and deleting users.

In *app/Http/Controllers* , create a new file *HomeController.php* and fill it with the following code.

Add Entire Code

```
<?php


namespace App \ Http \ Controllers;
use Illuminate \ Http \ Request;


class HomeController extends Controller
{
   public function index()
   {
      $viewData = [];
      $viewData["title"] = "Home Page - Online Store";
      return view('home.index')->with("viewData", $viewData);
   }

   public function about()
   {
      $data1 = "About us - Online Store";
      $data2 = "About us";
      $description = "This is an about page ...";
      $author = "Developed by: Your Name";
      return view('home.about')->with("title", $data1)
         ->with("subtitle", $data2)
         ->with("description", $description)
         ->with("author", $author);


   }
}
```

We have the *HomeController* which extends the Laravel *Controller* class. We have the *index* method that won't be discussed in this chapter. Just ignore it for now, we will use it to exemplify an important scenario in the next chapter. Then, we have the *about* method connected to the ("/about") route in the *routes/web.php* file. This method defines a set of variables and passes them to the *home.about* view.

Recap, when a user goes to the ("/") route, the *home.index* view will be displayed (delivered in the *routes/web.php* file). Likewise, when a user goes to the ("/about") route, the *home.about* view will be displayed (delivered in the *HomeController about* method).

Running the app

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

Execute in Terminal

```
php artisan serve
```

Go to the ("/") route, and you will see the new home page (see Fig. 6-2). Then, go to the ("/about") route, and you will see the about page (see Fig. 6-3).

Figure 6-2. Online Store – Home page.

Figure 6-3. Online Store – About page.

We created a couple of Laravel views, a Laravel Controller, and modified the web routes. However, we have not linked the navbar menu to the new routes. **We have also purposely introduced some serious mistakes when defining the previous elements. Why? It is because we want to illustrate the concept of clean code.** In the next chapter, we will refactor these elements and apply some strategies about clean code in routes, controllers, and views.

Chapter 07 – Refactoring Index and About Pages

The code in the previous chapter can be further cleaned and improved. We have not defined rules for coding any of the previous elements. We will see how to refactor the code to make it cleaner and more maintainable. We will also provide general tips for handling routes, controllers, and views. Keep in mind that most of the principles presented in this chapter apply not only to a Laravel project but can be applied in other MVC frameworks (such as Django, Spring, Nest, Express, and more).

Refactoring routes

Previous routes

The previous chapter showed two routes defined with different approaches. Let's check the first one.

Analyze Code

```
Route::get('/', function () {
    $viewData = [];
    $viewData["title"] = "Home Page - Online Store";
    return view('home.index')->with("viewData", $viewData);
});
```

It is one of the worst approaches you can apply (no matter the framework you are using). If you put the application logic in the route file, you will end with hundreds or thousands of code lines in those files as your project evolves, and maintainability will be the hell. So, we don't recommend using this strategy even for small projects. Our recommended strategy will be presented next.

Now, let's analyze the second one.

Analyze Code

```
Route::get('/about', 'App \ Http \ Controllers \ HomeController@about')->name("home.about");
```

It looks better. Our route is now connected to a controller method. We also put a name on that route. We recommend a combination of the controller's name plus the controller's method name to define the route name.

New routes

Let's refactor our routes. In *routes/web.php* , remove the previous ("/") route, and replace it for the following route in **bold**.

Modify Bold Code

```
...
~~Route::get('/', function () {~~
    ~~$viewData = [];~~
    ~~$viewData["title"] = "Home Page - Online Store";~~
    ~~return view('home.index')->with("viewData", $viewData);~~
~~});~~
Route::get('/', 'App \ Http \ Controllers \ HomeController@index')->name("home.index");
Route::get('/about', 'App \ Http \ Controllers \ HomeController@about')->name("home.about");
```

Now, the web routes look cleaner and consistent.

Quick discussion: Did you check the routes we defined in the previous chapter? Now think that there is a new member in your project. And the new member must include a new route in the *routes/web.php* file. Which approach will the member use? The first one? The second one? That is a big problem. So, a good strategy is to use the same approach for all routes as we did in the above code. Now new members will replicate this approach. You can also complement this idea with the next tip.

TIP: As a software developer, a good strategy is to create a document with architectural rules and share that document with your team (if you have one). You can make that document in the project repository wiki (if you have one). Encourage all the members to read that document. A first rule that you can include in that document could be: "In the route files, any URI must be only connected to controller methods. Putting application logic inside a route file is NOT allowed". These simple rules will save you much time and many headaches. We always create a document like that for all our projects, and we encourage our students to do it in their projects.

Refactoring controllers

Previous controller

The previous chapter showed a controller with two methods that used different approaches. Let's check the *about* method.

Analyze Code

```
public function about()
{
   $data1 = "About us - Online Store";
   $data2 = "About us";
   $description = "This is an about page ...";
   $author = "Developed by: Your Name";
   return view('home.about')->with("title", $data1)
      ->with("subtitle", $data2)
      ->with("description", $description)
      ->with("author", $author);


}
```

Here we have three problems.

- Variable naming is a mess. Using names such as *data1* or *data2* is horrible, it says nothing. Instead of that, we can use *title* and *subtitle* .
- We have many *with* methods chained. Imagine if we have 20 variables to pass to the view. We don't have consistency. We send each variable to the view one by one, but we only pass one array variable to the view in the *index* method. We prefer the *index* method strategy as we will see it next.
- Finally, we have a blank line before the ending of the curly bracket. We need to define a consistent coding style guide. This one will be solved in the next chapter.

Quick discussion: Let's see the importance of variable naming with two quotes from the (*2019 - Thomas, D., & Hunt, A. - The Pragmatic Programmer: your journey to mastery*) book. *"The beginning of wisdom is the ability to call things by their right names. - Confucius." - "Why is naming important? Because good names make code easier to read, and you have to read it to change it."*

Now, let's analyze the *index* method (which was ignored in the previous chapter).

Analyze Code

```
public function index()
{
   $viewData = [];
   $viewData["title"] = "Home Page - Online Store";
   return view('home.index')->with("viewData", $viewData);
}
```

In this case, we only define one variable called *viewData* which contains all the data sent to the view. This variable is an associative array. With this approach, it does not matter if we pass one data to the view or dozens. In both cases, we just pass the associative array.

TIP: In your architectural rules document, you can include a new rule saying that controllers should only pass an associative array called *viewData* to the views. Laravel provides many ways to pass that data to the view, it does not matter which one you choose. However, it is essential to mention the selected approach in the rules document to be used consistently by all team members through the application.

New controller

Let's refactor our controller. In *app/Http/Controllers/HomeController.php* , make the following changes in **bold**.

Modify Bold Code

```
<?php
   ...
   public function about()
   {
      $viewData = [];
      $viewData["title"] = "About us - Online Store";
      $viewData["subtitle"] =  "About us";
      $viewData["description"] =  "This is an about page ...";
      $viewData["author"] = "Developed by: Your Name";
      return view('home.about')->with("viewData", $viewData);


   }
}
```

Now that we use the single variable strategy, both methods are consistent. We suggest storing the views following

the next approach. First, we create a subfolder with the controller's name inside the *resources/views* folder. In this case, the subfolder is called *home* . And if a controller method displays data, we create a file with the controller's method name inside the view controller subfolder. So, we have the *about.blade.php* file stored inside the *resources/views/home* subfolder. It makes easier to find specific views for specific controller methods.

Refactoring views

The previous chapter showed two views that display data a little differently. The *home/index* view won't be changed since it displays the data using the *viewData* strategy. However, we will need to modify the *home/about* view to match the single variable strategy previously defined.

Let's refactor the *about* view. In *resources/views/home/about.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
@extends('layouts.app')
@section('title', $viewData["title"])
@section('subtitle', $viewData["subtitle"])
@section('content')
<div class="container">
  <div class="row">
    <div class="col-lg-4 ms-auto">
      <p class="lead">{{ $viewData["description"] }}</p>
    </div>
    <div class="col-lg-4 me-auto">
      <p class="lead">{{ $viewData["author"] }}</p>
    </div>
  </div>
</div>
@endsection
```

As you can see, we now access the data through the single *viewData* associative array. We will use this strategy across the entire application, making our views more consistent.

Quick discussion: The previous data can also be placed directly over the previous views. We mean, you don't need to define those texts as variables in the controller and send them to the view. Instead, you can place the text directly in the views as we did in our *welcome.blade.php* view. We did it that way to illustrate and explain some Laravel elements. There is a better option that is out of this book's scope (which is what we prefer). That option is called Localization. In Localization, you move away those texts from controllers and views and place them in the *resources/lang* folder. Localization allows you even to retrieve strings in various languages. It is not difficult to implement. You can use this link to get more info about it https://laravel.com/docs/9.x/localization#introduction, search in Google, or let us know if you need a good example (use the discussion zone of the book repository).

Updating links in Header

Now that we have the proper routes, controller, and views, let's include the links in the header. In *resources/views/layouts/app.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
...
<body>
<!-- header -->
 <nav class="navbar navbar-expand-lg navbar-dark bg-secondary py-4">
   <div class="container">
     <a class="navbar-brand" href="{{ route('home.index') }}">Online Store</a>
     <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNavAltMarkup"
       aria-controls="navbarNavAltMarkup" aria-expanded="false" aria-label="Toggle navigation">
       <span class="navbar-toggler-icon"></span>
     </button>
     <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
       <div class="navbar-nav ms-auto">
         <a class="nav-link active" href="{{ route('home.index') }}">Home</a>
         <a class="nav-link active" href="{{ route('home.about') }}">About</a>
       </div>
         </div>
      </div>
   </nav>
...
```

We use the *route* helper method in the previous layout, which generates a URL for a given named route. We used the names of the routes defined for the ("/") route (*home.index*) and the ("/about") route (*home.about*).

Running the app

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

Execute in Terminal

```
php artisan serve
```

Now, you can navigate between the *Home* page and the *About* page by using the links in the navigation bar (see Fig. 7-1).

Figure 7-1. Online Store – About page.

Note: you can find the application code at the GitHub repository in https://github.com/PracticalBooks/Practical-Laravel.

Chapter 08 – Use of a Coding Standard

A **coding standard** is a set of rules and agreements used when writing source code in a particular programming language. Using a coding standard provides some advantages, such as:

- It gives a uniform appearance to the codes written by different programmers.
- It improves the readability and maintainability of the code and reduces complexity.
- It helps in code reuse and helps to detect errors easily.
- It promotes good programming practices and increases the efficiency of the programmers.

One of the most used PHP coding standards is *PSR-2* . It is a standard established by PHP-FIG (PHP Framework Interop Group, more information about *PSR-2* can be found at https://www.php-fig.org/psr/psr-2/). Some tools help programmers to automatically check their code against these coding standards. In this book, we will use PHP_CodeSniffer, which is presented next.

Introducing PHP_CodeSniffer

PHP_CodeSniffer is a set of two PHP scripts. The *phpcs* script tokenizes PHP, JavaScript, and CSS files to detect violations based on a defined coding standard. The *phpcbf* script automatically corrects coding standard violations. PHP_CodeSniffer is an essential development tool that ensures your code remains clean and consistent. More information about PHP_CodeSniffer can be found here: https://github.com/squizlabs/PHP_CodeSniffer.

Installing and configuring PHP_CodeSniffer

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

Execute in Terminal

```
composer require --dev squizlabs/php_codesniffer
```

The previous command installs the PHP_CodeSniffer library in our *composer.json* file, which downloads and extracts the library files in the *vendor* folder.

We will apply a quick configuration of PHP_CodeSniffer. In the project root directory, create a new file *phpcs.xml* and fill it with the following code.

Add Entire Code

```
<?xml version="1.0"?>
<ruleset name="PHP_CodeSniffer">
  <description>The coding standard for the Online Store project</description>
  <rule ref="PSR2"/>
   <file>app/</file>
   <file>config/</file>
   <file>database/</file>
   <file>routes/</file>
   <exclude-pattern>*/migrations/*</exclude-pattern>
</ruleset>
```

This file establishes that PHP_CodeSniffer will use *PSR-2* as the coding style standard. It also defines that when we run PHP_CodeSniffer, it will find errors in the *app/** , *config/** , *database/** , and *routes/** folders. We exclude locations that include a *migrations* folder.

Running PHP_CodeSniffer

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

Execute in Terminal

```
./vendor/bin/phpcs
```

The previous command executes PHP_CodeSniffer based on the custom configuration defined in the *phpcs.xml* file (see Fig. 8-1). Our case showed 17 errors, all with an “X” mark, which means that PHP_CodeSniffer can handle them and fix them (if you want). Some of those errors are related to spaces, indentation, and rules defined in the *PSR2* coding style guide. For example, there was an annoying blank line at the end of the *about* method of the *HomeController* (before the curly brace closing). So, PHP_CodeSniffer marked it as an error.

```
PS C:\xampp\htdocs\onlineStore> ./vendor/bin/phpcs

FILE: ...xampp\htdocs\onlineStore\app\Http\Controllers\HomeController.php
----------------------------------------------------------------------
FOUND 17 ERRORS AFFECTING 17 LINES
----------------------------------------------------------------------
  1 | ERROR | [x] End of line character is invalid; expected "\n" but
    |       |     found "\r\n"
  3 | ERROR | [x] There must be one blank line after the namespace
    |       |     declaration
  8 | ERROR | [x] Line indented incorrectly; expected 4 spaces, found
    |       |     2
  9 | ERROR | [x] Line indented incorrectly; expected at least 4
    |       |     spaces, found 2
```

Figure 8-1. Running PHP CodeSniffer.

To automatically fix it, we need to execute the PHP_CodeSniffer *phpcbf* file with the following command (see Fig. 8-2).

Execute in Terminal

```
./vendor/bin/phpcbf
```

After that, it shows the errors fixed. This strategy is better than a manual fix. For the rest of the book, we will use PHP_CodeSniffer to improve the code quality.

```
PS C:\xampp\htdocs\onlineStore> ./vendor/bin/phpcbf

PHPCBF RESULT SUMMARY
----------------------------------------------------------------------
FILE                                                  FIXED  REMAINING
----------------------------------------------------------------------
...lineStore\app\Http\Controllers\HomeController.php  17     1
----------------------------------------------------------------------
A TOTAL OF 17 ERRORS WERE FIXED IN 1 FILE
----------------------------------------------------------------------

Time: 209ms; Memory: 8MB
```

Figure 8-2. Fixing style errors with PHP CodeSniffer.

Final remarks

- There are other tools that you can use to extend the PHP_CodeSniffer capabilities. For example, Larastan. Larastan is a static analysis tool for Laravel Projects (https://github.com/nunomaduro/larastan), this tool even discovers bugs in your code without running it.
- PHP_CodeSniffer does not work well with blade files. A formatting Visual Studio Code plugin called *Format HTML in PHP* can help you to format your Blade views.
- PHP_CodeSniffer does not automatically fix all errors. For example, there is a standard of using camelCase coding style to name methods. If you change the *HomeController about* method to *About*, PHP_CodeSniffer will mark that error, but it won't fix it automatically. You must fix this error manually.

TIP: There is a good story about "The Broken Window Theory" described in the (*2019 - Thomas, D., & Hunt, A. - The Pragmatic Programmer: your journey to mastery*) book. You can search it in Google. From that story, we want to highlight the next tip. *Don't leave "broken windows" (e.g., bad designs, wrong decisions, or poor code) unrepaired. Fix each one as soon as it is discovered.* The previous three chapters showed many broken windows, which fortunately were fixed.

TIP: Always use a coding standard tool, formatter, static code analysis tool, or even a combination of them in your projects. It will save you much time and improve the code quality. In addition, you will find linters available for most programming languages. Besides, include a rule in your architectural rules document mentioning that all code changes should be previously verified by using these tools. You can even automate this process (with a pipeline or CI/CD strategy). However, it is out of the scope of this book. You can check an article that Daniel wrote for a Laravel 8 project https://faun.pub/configure-laravel-8-for-ci-cd-with-jenkins-and-github-part-1-58b9be304292.

Chapter 09 – List Products with Dummy Data

In this chapter, we will implement functionality to list all products and to be able to click those products and display their data in a separate section.

Modifying routes

Let's start by including a couple of routes to list products and show the data of a single product. In *routes/web.php* , make the following changes in **bold**.

Modify Bold Code

```
…
Route::get('/', 'App \ Http \ Controllers \ HomeController@index')->name("home.index");
Route::get('/about', 'App \ Http \ Controllers \ HomeController@about')->name("home.about");
Route::get('/products', 'App \ Http \ Controllers \ ProductController@index')->name("product.index");
Route::get('/products/{id}', 'App \ Http \ Controllers \ ProductController@show')->name("product.show");
```

We will list all the application products in the first route ("/products"). The second route will be used to show the data of a single product. "/products/{id}" takes a parameter called *id* . This parameter is the product *id* to identify which product data to show. For example, if we access "/products/1", the application will display the data of the product with id=1.

If you noticed, both routes are connected to the *ProductController* . So, let's implement it.

ProductController

In *app/Http/Controllers* , create a new file *ProductController.php* and fill it with the following code.

Add Entire Code

```
<?php

namespace App \ Http \ Controllers;

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";
            $viewData["products"] = ProductController::$products;
            return view('product.index')->with("viewData", $viewData);
      }

      public function show($id)
      {
            $viewData = [];
            $product = ProductController::$products[$id-1];
            $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

```
public static $products = [
      ["id"=>"1", "name"=>"TV", "description"=>"Best TV", "image" => "game.png", "price"=>"1000"],
      …
]
```

$products is an array that contains a set of products with its data. In the array index 0, we have the product with id=1 (the "TV"). We have four dummy products. Currently, they are stored as a static attribute of the

ProductController class. We will later retrieve product data from a MySQL database in upcoming chapters.

Analyze Code

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

The *index* method gets the array of products and sends them to the *product.index* view to be displayed.

Analyze Code

```
public function show($id)
{
    $viewData = [];
    $product = ProductController::$products[$id-1];
    $viewData["title"] = $product["name"]." - Online Store";
    $viewData["subtitle"] =  $product["name"]." - Product information";
    $viewData["product"] = $product;
    return view('product.show')->with("viewData", $viewData);
}
```

The *show* method gets an *$id* as a parameter (the *id* is collected from the URL). We extract the product data with that *id* (check the **bold** line). We subtract one unit since we stored the product with id=1 in the *ProductController::$products* array index 0, the product with id=2 in the *ProductController::$products* array index 1, and so on. We then send the product data to the *product.show* view.

Product views

Let's first implement the *product.index* view, and then, the *product.show* view.

Product index view

In *resources/views/*, create a subfolder *product*. Then, in *resources/views/product*, create a new file *index.blade.php* and fill it with the following code.

Add Entire Code

```
@extends('layouts.app')
@section('title', $viewData["title"])
@section('subtitle', $viewData["subtitle"])
@section('content')
<div class="row">
  @foreach ($viewData["products"] as $product)
  <div class="col-md-4 col-lg-3 mb-2">
    <div class="card">
      <img src="{{ asset('/img/'.$product["image"]) }}" class="card-img-top img-card">
      <div class="card-body text-center">
        <a href="{{ route('product.show', ['id'=> $product["id"]]) }}"
          class="btn bg-primary text-white">{{ $product["name"] }}</a>
      </div>
    </div>
  </div>
  @endforeach
</div>
@endsection
```

The important part of the previous code is the *@foreach*. *@foreach* is a Blade directive which allows us to iterate over a list. We iterate through each product and display the product image and name. More information about Blade directives can be found here: https://laravel.com/docs/9.x/blade#blade-directives.

Analyze Code

```
<a href="{{ route('product.show', ['id'=> $product["id"]]) }}"
          class="btn bg-primary text-white">{{ $product["name"] }}</a>
```

Finally, we put a link to the product name. The link will route to the *product.show* route (defined previously in *routes/web.php*) and it requires a parameter to be sent. In this case, we send the product id of the current iterated product.

Product show view

In *resources/views/product*, create a new file *show.blade.php* and fill it with the following code.

Add Entire Code

```
@extends('layouts.app')
```

```
@section('title', $viewData["title"])
@section('subtitle', $viewData["subtitle"])
@section('content')
<div class="card mb-3">
   <div class="row g-0">
      <div class="col-md-4">
         <img src="{{ asset('/img/'.$viewData["product"]["image"]) }}" class="img-fluid rounded-start">
      </div>
      <div class="col-md-8">
         <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>
      </div>
   </div>
</div>
@endsection
```

We show the product image, name, and description in the above code. Remember, we are using dummy products. This will change in upcoming chapters.

TIP: In the last examples, we have defined a structure to store our controllers, controllers' methods, routes' names, and views. For example, the *product.show* route is linked to the *ProductController show* method, which displays the *product/show* view. Try to use this strategy across the entire project as it facilitates finding the views of the corresponding controllers' methods and vice versa.

Updating links in Header

Now, let's include the products link in the header. In *resources/views/layouts/app.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
...
            <div class="navbar-nav ms-auto">
               <a class="nav-link active" href="{{ route('home.index') }}">Home</a>
               <a class="nav-link active" href="{{ route('product.index') }}">Products</a>
               <a class="nav-link active" href="{{ route('home.about') }}">About</a>
            </div>
...
```

Running the app

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

Execute in Terminal

```
php artisan serve
```

Now, you can go to the *Products* page (see Fig. 9-1) and navigate to a specific product (see Fig. 9-2).

Figure 9-1. Online Store – Products page.

Figure 9-2. Online Store – iPhone product page.

Chapter 10 – Configuration of MySQL Database

Introduction to MySQL

MySQL is the most popular Open-Source SQL database management system developed, distributed, and supported by Oracle.

- **MySQL is a database management system**. A database is a structured collection of data. It may be anything from a simple shopping list to a picture gallery or the vast amounts of information in a corporate network. To add, access, and process data stored in a computer database, you need a database management system such as **MySQL**.
- **MySQL databases are relational.** A relational database stores data in separate tables rather than putting all the data in one big storeroom. The database structures are organized into physical files optimized for speed. MySQL provides a logical model with objects such as databases, tables, views, rows, and columns, which offers a flexible programming environment.

MySQL tables

A table is used to organize data in rows and columns. It is used for both storing and displaying records in a structured format. It is like worksheets in a spreadsheet application. The **columns** specify the data type, whereas the **rows** contain the actual data. Below is how you could imagine a MySQL table (see Fig. 10-1).

id	name	description	image	price
1	TV	Best tv	game.png	1000
2	iPhone	Best iPhone	safe.png	999
3	Chromecast	Best Chromecast	submarine.png	30
4	Glasses	Best Glasses	game.png	100

Figure 10-1. Product table.

Introduction to Laravel databases

Almost every modern web application interacts with a database. Laravel makes interacting with databases straightforward across a variety of supported databases. Laravel provides support for four databases:

•MySQL 5.7+
•PostgreSQL 9.6+
•SQLite 3.8.8+
•SQL Server 2017+

In this project, we will work with MySQL. We will use the MySQL module that XAMPP (also WAMP) provides.

Configuring our database

Execute XAMPP, *start* the Apache Module, *start* the MySQL module, and click the *MySQL Admin* button (of the MySQL module). It takes us to the phpMyAdmin application (see Fig. 10-2).

Figure 10-2. Starting MySQL module in XAMPP.

Note: If you are using WAMP or another similar application, the phpMyAdmin application can be commonly accessed through the next route: http://localhost/phpmyadmin/.

In the phpMyAdmin application, enter your username and password. Default values are “root” (for the username)

and an empty password (see Fig. 10-3).

Figure 10-3. XAMPP phpMyAdmin application.

Once logged in to phpMyAdmin, click the *Databases* tab. Enter the database name "online_store", and click *Create* button (see Fig. 10-4).

Figure 10-4. Database creation.

We will use the user, password, and database name in the next chapter.

Chapter 11 – Product Migration

Introduction to Laravel migrations

Now, we can just go to the phpMyAdmin application and create tables inside the *online_store* database (such as the product table). It can be done through the fill of a form. It is the **traditional way**. However, it has an issue, it does not allow us to have version control of our database tables and queries. If you have ever had to tell a teammate to manually add a table column to their local database schema after pulling in your changes from source control (such as GitHub), you have faced this issue.

Laravel **migrations** are like version control for our database. They solve the previous issue allowing us to define and share the application's database schema definition.

We won't create database tables in the traditional way. Instead, we will do it through Laravel migrations.

Product migration

Let's create our product migration. In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan make:migration create_products_table
```

The previous command creates a products table migration file inside the *database/migrations* folder. Each migration file name contains a timestamp that allows Laravel to determine the order of the migrations. In our case, it created a file named *2022_02_11_153916_create_products_table.php* .

If you look at the generated file, it contains a migration class. Migration classes contain two methods: *up* and *down* . The *up* method is used to add new tables, columns, or indexes to your database. In contrast, the *down* method should reverse the operations performed by the *up* method.

Now, open the previously generated file. Delete all the existing code and fill it with the following code.

Replace Entire Code

```
<?php


use Illuminate \ Database \ Migrations \ Migration;
use Illuminate \ Database \ Schema \ Blueprint;
use Illuminate \ Support \ Facades \ Schema;


return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('products', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->text('description');
            $table->string('image');
            $table->integer('price');
            $table->timestamps();
        });
    }


    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('products');
    }
};
```

Let's analyze the previous code by parts.

Analyze Code

```
    public function up()
```

```
{
    Schema::create('products', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->text('description');
            $table->string('image');
            $table->integer('price');
            $table->timestamps();
    });
}
```

The *up* method will create a new database table called *products* . By default, Laravel suggests creating the table names in plural. It is due to how the Laravel ORM "Eloquent" system works (discussed later). The *up* method defines the creation of the *products* table with five columns (*id* , *name* , *description* , *image* , and *price*). Therefore, a *timestamps* method will add two columns (*created_at* and *updated_at*). We also used five column types (*id* , *string* , *text* , *integer* , and *timestamps*). More information about available column types can be found here: https://laravel.com/docs/9.x/migrations#available-column-types.

Analyze Code

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

The *down* method contains the opposite of the *up* method. It drops the *products* table. We will see how to execute the migration classes *up* and *down* methods later. But first, we need to update our database credentials in our Laravel project.

TIP: Always define your database schema using migrations or similar approaches. Remember, they work as version control of your databases. Most web application frameworks provide these kinds of features. For example, Django provides Django Migrations which work like the previous approach.

Modifying the .env file

To execute the migrations, we need to modify the *.env* file (located at the project root folder). In the *.env* file, make the following changes in **bold**. You need to set the *DB_DATABASE* , *DB_USERNAME* , and *DB_PASSWORD* . If you have a different database name, username, or password, make the corresponding changes.

Modify Bold Code

```
...
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=online_store
DB_USERNAME=root
DB_PASSWORD=
...
```

Executing the migrations

To run the migrations, in the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan migrate
```

The previous command executes the migrations defined in the *database/migrations* folder. Five migrations are executed. The last one is our "create products table" migration. The other migrations correspond to Laravel default migrations. We will take advantage of some of them in upcoming chapters. If everything works as expected, you should see a result as presented in Fig. 11-1.

```
PS C:\xampp\htdocs\onlineStore> php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table (26.54ms)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table (24.25ms)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated:  2019_08_19_000000_create_failed_jobs_table (24.61ms)
Migrating: 2019_12_14_000001_create_personal_access_tokens_table
Migrated:  2019_12_14_000001_create_personal_access_tokens_table (34.73ms)
Migrating: 2022_02_11_153916_create_products_table
Migrated:  2022_02_11_153916_create_products_table (11.84ms)
```

Figure 11-1. Execution of Laravel migrations.

Verifying the migrations in phpMyAdmin

It is time to verify that the migrations were properly applied in our database. Go to the phpMyAdmin application and click over the *online_store* database (see Fig. 11-2).

Figure 11-2. Accessing our database from phpMyAdmin.

As you can see, our *products* table appears listed. Migrations worked!

Inserting products

Let's insert four products into our database. For now, we will insert it manually (through SQL queries). Later, we will insert products through a form in an upcoming chapter.

In phpMyAdmin, click the *online_store* database, click the *SQL* tab, paste the following SQL queries, and click *go* (see Fig. 11-3).

Execute in Database

```
INSERT INTO products (id, name, description, image, price, created_at, updated_at) VALUES (NULL, 'TV', 'Best TV', 'game.png', '1000', '2021-10-01 00:00:00', '2021-10-01 00:00:00');
INSERT INTO products (id, name, description, image, price, created_at, updated_at) VALUES (NULL, 'iPhone', 'Best iPhone', 'safe.png', '999', '2021-10-01 00:00:00', '2021-10-01 00:00:00');
INSERT INTO products (id, name, description, image, price, created_at, updated_at) VALUES (NULL, 'Chromecast', 'Best Chromecast', 'submarine.png', '30', '2021-10-01 00:00:00', '2021-10-01 00:00:00');
INSERT INTO products (id, name, description, image, price, created_at, updated_at) VALUES (NULL, 'Glasses', 'Best Glasses', 'game.png', '100', '2021-10-01 00:00:00', '2021-10-01 00:00:00');
```

Figure 11-3. Inserting products through phpMyAdmin.

Let's check that the products were successfully inserted. In phpMyAdmin, click the *online_store* database, and click the *products* table. Hopefully, you will see the four products inserted (see Fig. 11-4).

Figure 11-4. Products displayed in phpMyAdmin.

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.

Chapter 15 – Admin Panel

Many web applications have an admin section that administrators or moderators can access to manage the application data, such as registering products, managing sales, generating reports, and managing users. Commonly, this section is called an **administration panel**. Due to its nature, this section is secured by a login or authentication system. For now, we will create a public administration panel, which any visitor can access. We will later implement a login system to secure and verify that only allowed users (admins) can access this section.

To create the admin panel, we need to include a layout, a controller, a view, new files, and a new route. So, let's begin.

Admin Layout

Commonly, administration panels look different from the main pages. They are quite minimal and many of them display information like a spreadsheet. Let's start our admin panel construction by defining a new layout. This layout will be used across the admin panel pages.

In *resources/views/layouts* , create a new file *admin.blade.php* and fill it with the following code.

Add Entire Code

```
<!doctype html>
<html lang="en">

<head>
   <meta charset="utf-8" />
   <meta name="viewport" content="width=device-width, initial-scale=1" />
   <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.min.css" rel="stylesheet"
      crossorigin="anonymous" />
   <link href="{{ asset('/css/admin.css') }}" rel="stylesheet" />
   <title>@yield('title', 'Admin - Online Store')</title>
</head>

<body>
   <div class="row g-0">
      <!-- sidebar -->
      <div class="p-3 col fixed text-white bg-dark">
         <a href="{{ route('admin.home.index') }}" class="text-white text-decoration-none">
            <span class="fs-4">Admin Panel</span>
         </a>
     <hr />
     <ul class="nav flex-column">
      <li><a href="{{ route('admin.home.index') }}" class="nav-link text-white">- Admin - Home</a></li>
      <li><a href="#" class="nav-link text-white">- Admin - Products</a></li>
      <li>
        <a href="{{ route('home.index') }}" class="mt-2 btn bg-primary text-white">Go back to the home page</a>
      </li>
     </ul>
   </div>
      <!-- sidebar -->
      <div class="col content-grey">
      <nav class="p-3 shadow text-end">
         <span class="profile-font">Admin</span>
         <img class="img-profile rounded-circle" src="{{ asset('/img/undraw_profile.svg') }}">
      </nav>

      <div class="g-0 m-5">
         @yield('content')
      </div>
   </div>
</div>

<!-- footer -->
<div class="copyright py-4 text-center text-white">
<div class="container">
  <small>
    Copyright - <a class="text-reset fw-bold text-decoration-none" target="_blank"
      href="https://twitter.com/danielgarax">
      Daniel Correa
    </a>
  </small>
   </div>
</div>
<!-- footer -->
```

```
 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/js/bootstrap.bundle.min.js" crossorigin="anonymous">
 </script>
</body>


</html>
```

We have a new layout. This layout presents a different structure and contains a sidebar with three links.

- **"- Admin - Home"** links to the admin panel home page.
- **"- Admin - Products"** links to the admin panel product management page (this page will be implemented in the next chapter).
- **"Go back to the home page"** links back to the online store home page.

The *admin.blade.php* layout imports an *admin.css* file and a new image. Let's add these elements to our project.

Creating the admin.css file

In *public/css* , create a new file *admin.css* and fill it with the following code.

Add Entire Code

```
.copyright {
   background-color: #1a252f;
}

.fixed {
   -ms-flex: 0 0 250px;
   flex: 0 0 250px;
}

.content-grey {
   background-color: #f8f9fc;
}

hr {
   margin-top: 0.8em;
   margin-bottom: 0.8em;
}

.img-profile {
   height: 2rem;
   width: 2rem;
}

.profile-font {
   color: #858796 !important;
   font-size: 80%;
   font-weight: 400;
}

.card-header {
   background-color: #f8f9fc;
   border-bottom: 1px solid #e3e6f0;
}
```

We have some custom CSS classes. Most of them are used in the *admin* layout.

Adding the undraw profile image

Download *undraw_profile.svg* file from this link https://github.com/PracticalBooks/Practical-Laravel/blob/main/Chapter15/onlineStore/public/img/undraw_profile.svg and store it inside the *public/img* folder. Alternatively, store your own admin profile image.

AdminHomeController

In *app/Http/Controllers/* , create a subfolder *Admin* . Then, in *app/Http/Controllers/Admin* , create a new file *AdminHomeController.php* and fill it with the following code.

Add Entire Code

```
<?php


namespace App \ Http \ Controllers \ Admin;


use App \ Http \ Controllers \ Controller;
use Illuminate \ Http \ Request;
```

```
class AdminHomeController extends Controller
{
    public function index()
    {
        $viewData = [];
        $viewData["title"] = "Admin Page - Admin - Online Store";
        return view('admin.home.index')->with("viewData", $viewData);
    }
}
```

We have a simple *index* method that displays the *admin.home.index* view.

Admin index view

Let's create the admin index view. In *resources/views/* , create a subfolder *admin* . Next, create a subfolder *home* in *resources/views/admin* . Finally, in *resources/views/admin/home* , create a new file *index.blade.php* and fill it with the following code.

Add Entire Code

```
@extends('layouts.admin')
@section('title', $viewData["title"])
@section('content')
<div class="card">
  <div class="card-header">
    Admin Panel - Home Page
  </div>
  <div class="card-body">
    Welcome to the Admin Panel, use the sidebar to navigate between the different options.
  </div>
</div>
@endsection
```

We have a simple view that displays a "welcome to the admin panel" message. Note that this view extends the new *admin* layout (not the *app* layout).

Modifying routes

Let's include a new route to the admin home page. In *routes/web.php* , make the following changes in **bold**.

Modify Bold Code

```
…
Route::get('/products', 'App \ Http \ Controllers \ ProductController@index')->name("product.index");
Route::get('/products/{id}', 'App \ Http \ Controllers \ ProductController@show')->name("product.show");
Route::get('/admin', 'App \ Http \ Controllers \ Admin \ AdminHomeController@index')->name("admin.home.index");
```

The new route ("/admin") will display the new admin home page (which is rendered inside the *AdminHomeController index* method).

Running the app

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

Execute in Terminal

```
php artisan serve
```

Now go to the ("/admin") route, and you will see the new Admin Panel (see Fig. 15-1).

Figure 15-1. Online Store – Admin Panel.

Note: you can find the application code at the GitHub repository in https://github.com/PracticalBooks/Practical-Laravel

Chapter 16 – List Products in Admin Panel

Let's create the admin product management section. For now, we will create an index page that lists all products.

Modifying routes

Let's include a new route to the admin product index page. In *routes/web.php* , make the following changes in **bold**.

Modify Bold Code

```
…
Route::get('/admin', 'App \ Http \ Controllers \ Admin \ AdminHomeController@index')->name("admin.home.index");
Route::get('/admin/products', 'App \ Http \ Controllers \ Admin \ AdminProductController@index')->name("admin.product.index");
```

The new route ("/admin/products") will be the entry point to manage our products.

AdminProductController

In *app/Http/Controllers/Admin* , create a new file *AdminProductController.php* and fill it with the following code.

Add Entire Code

```
<?php


namespace App \ Http \ Controllers \ Admin;


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


class AdminProductController extends Controller
{
	public function index()
	{
		$viewData = [];
		$viewData["title"] = "Admin Page - Products - Online Store";
		$viewData["products"] = Product::all();
		return view('admin.product.index')->with("viewData", $viewData);
	}
}
```

We have an *index* method that collects the products data and displays the *admin.products.index* view.

Admin product index view

In *resources/views/admin* , create a subfolder *product* . Then, in *resources/views/admin/product* , create a new file *index.blade.php* and fill it with the following code.

Add Entire Code

```
@extends('layouts.admin')
@section('title', $viewData["title"])
@section('content')
<div class="card">
   <div class="card-header">
      Manage Products
   </div>
   <div class="card-body">
      <table class="table table-bordered table-striped">
         <thead>
            <tr>
               <th scope="col">ID</th>
               <th scope="col">Name</th>
               <th scope="col">Edit</th>
               <th scope="col">Delete</th>
            </tr>
         </thead>
         <tbody>
            @foreach ($viewData["products"] as $product)
            <tr>
               <td>{{ $product->getId() }}</td>
               <td>{{ $product->getName() }}</td>
               <td>Edit</td>
               <td>Delete</td>
            </tr>
            @endforeach
         </tbody>
      </table>
```

```
  </div>
</div>
@endsection
```

We have a table which displays the products' ids and names. This table was designed based on Bootstrap Tables (https://getbootstrap.com/docs/5.1/content/tables/). Later, we will include a link to edit and delete specific products.

Updating links in Admin layout

Now that we have the proper admin product route, let's include it in the *admin* layout. In *resources/views/layouts/admin.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
...
    <!-- sidebar -->
      ...
      <ul class="nav flex-column">
    <li><a href="{{ route('admin.home.index') }}" class="nav-link text-white">- Admin - Home</a></li>
    <li><a href="{{ route('admin.product.index') }}" class="nav-link text-white">- Admin - Products</a></li>
    <li>
      <a href="{{ route('home.index') }}" class="mt-2 btn bg-primary text-white">Go back to the home page</a>
    </li>
      </ul>
    </div>
    <!-- sidebar -->
...
```

Running the app

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

Execute in Terminal

```
php artisan serve
```

Now go to the ("/admin/products") route, and you will see the new Admin Product page (see Fig. 16-1). Remember that you will need to have MySQL running. Otherwise, you will get a database connection error.

Figure 16-1. Online Store – Admin Panel – Products.

Chapter 17 – Create Products

Now, we will focus on the admin panel system to create products.

Modifying routes

In *routes/web.php* , make the following changes in **bold**.

Modify Bold Code

```
…
Route::get('/admin/products', 'App \ Http \ Controllers \ Admin \ AdminProductController@index')->name("admin.product.index");
Route::post('/admin/products/store', 'App \ Http \ Controllers \ Admin \ AdminProductController@store')->name("admin.product.store");
```

The new route ("/admin/products/store") will collect and store the newly created products' data. It uses a *post* HTTP method since the controller method will collect information for a form.

Modifying admin/product/index.blade.php view

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

Modify Bold Code

```
@extends('layouts.admin')
@section('title', $viewData["title"])
@section('content')
<div class="card mb-4">
   <div class="card-header">
      Create Products
   </div>
   <div class="card-body">
      @if($errors->any())
      <ul class="alert alert-danger list-unstyled">
         @foreach($errors->all() as $error)
         <li>- {{ $error }}</li>
         @endforeach
      </ul>
      @endif

      <form method="POST" action="{{ route('admin.product.store') }}">
         @csrf
         <div class="row">
            <div class="col">
               <div class="mb-3 row">
                        <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Name:</label>
                        <div class="col-lg-10 col-md-6 col-sm-12">
                           <input name="name" value="{{ old('name') }}" type="text" class="form-control">
                        </div>
               </div>
            </div>
            <div class="col">
               <div class="mb-3 row">
                        <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Price:</label>
                        <div class="col-lg-10 col-md-6 col-sm-12">
                           <input name="price" value="{{ old('price') }}" type="number" class="form-control">
                        </div>
               </div>
            </div>
         </div>
         <div class="mb-3">
            <label class="form-label">Description</label>
            <textarea class="form-control" name="description" rows="3">{{ old('description') }}</textarea>
         </div>
         <button type="submit" class="btn btn-primary">Submit</button>
      </form>
   </div>
</div>
<div class="card">
   <div class="card-header">
      Manage Products
...
```

Let's analyze the previous code by parts.

Analyze Code

```
      @if($errors->any())
      <ul class="alert alert-danger list-unstyled">
         @foreach($errors->all() as $error)
         <li>- {{ $error }}</li>
```

```
        @endforeach
    </ul>
    @endif
```

Laravel provides an *$errors* variable which is available in all views. This variable allows access to errors reported by the application. For example, errors are found when submitting a form with incomplete or invalid inputs. In the previous code, we are displaying all those errors.

Analyze Code

```
<form method="POST" action="{{ route('admin.product.store') }}">
    @csrf
    …
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Next, we have an HTML *form*. This *form* specifies a POST method and links the form with the *admin.product.store* route (the route that was defined in the previous section). POST method is used to send data to the application server. Laravel requires to include a hidden CSRF token field for all our HTML forms. That token protects us against CSRF attacks (more info here: https://owasp.org/www-community/attacks/csrf). To create that token, we use the *@csrf* Blade directive. Finally, we have the *submit* button to submit the form.

Analyze Code

```
        <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Name:</label>
        <div class="col-lg-10 col-md-6 col-sm-12">
            <input name="name" value="{{ old('name') }}" type="text" class="form-control">
        </div>
```

The rest of the code shows form inputs, buttons, and a text area. This form and elements were designed based on Bootstrap Forms (https://getbootstrap.com/docs/5.1/forms/overview/). They show an input to enter the product name. We are also using as the input value this code *value="{{ old('name') }}"*. The *old* Laravel helper is used to repopulate a previously collected input value if we find some errors.

Note: we are not yet collecting the product image. We will do so in the next chapter.

Modifying AdminProductController

Next, we create the *store* method in *AdminProductController*. In *app/Http/Controllers/Admin/AdminProductController.php*, make the following changes in **bold**.

Modify Bold Code

```
...
class AdminProductController extends Controller
{
    ...

    public function store(Request $request)
    {
        $request->validate([
                "name" => "required|max:255",
                "description" => "required",
                "price" => "required|numeric|gt:0",
                'image' => 'image',
        ]);

        $newProduct = new Product();
        $newProduct->setName($request->input('name'));
        $newProduct->setDescription($request->input('description'));
        $newProduct->setPrice($request->input('price'));
        $newProduct->setImage("game.png");
        $newProduct->save();

        return back();
    }
}
```

Let’s analyze the previous code by parts.

Analyze Code

```
    public function store(Request $request)
    {
        $request->validate([
                "name" => "required|max:255",
                "description" => "required",
                "price" => "required|numeric|gt:0",
                'image' => 'image',
```

```
        ]);
```

store function receives a *$request* object. This object allows us to interact with the HTTP request handled by our application. It also allows us to retrieve the inputs, cookies, and files submitted with the request.

Next, we use the *validate* method provided by the *$request* object. If the validation rules pass, the code will keep executing normally. However, an exception will be thrown if validation fails, and the proper error response will automatically be sent back to the user. Those errors will be displayed in the *product/index.blade.php* view through the *$errors* global variable.

In our case, *name* and *description* are required, and the *name* has a maximum length of 255 characters. Besides, *price* is required, must be numeric and greater than zero. Then, we are validating the *image* input contains only image files (this will be used in the next chapter). You can find the available validation rules here: https://laravel.com/docs/9.x/validation#available-validation-rules.

Analyze Code

```
        $newProduct = new Product();
        $newProduct->setName($request->input('name'));
        $newProduct->setDescription($request->input('description'));
        $newProduct->setPrice($request->input('price'));
        $newProduct->setImage("game.png");
        $newProduct->save();
```

Next, we create a *newProduct* instance. Then, we set the *newProduct* attributes based on values collected from the form. We use the *request->input* method to retrieve the form inputs. We also set a default image (*game.png*). In the end, we invoke the *save* method, which inserts the object data into the database. The model's *created_at* and *updated_at* timestamps will automatically be set when the *save* method is called, so there is no need to set them manually.

Note: setting one by one each model attribute presents an issue. When a model has many attributes, it implies manually calling several setters. We will see an alternative at the end of this chapter.

Analyze Code

```
        return back();
```

Finally, we invoke the *back* helper redirecting the user to the previous location.

TIP: If you have required inputs, include the *required* attribute to the HTML inputs (https://www.w3schools.com/tags/att_input_required.asp). This way, we can also validate those inputs using the client browser. We didn't implement them because we wanted to show how the Laravel validations work. But you should refactor all required inputs with the previous strategy.

Running the app

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

Execute in Terminal

```
php artisan serve
```

Now go to the ("/admin/products") route, and you will see the new form. First, complete the form with invalid data (see Fig. 17-1). Then, when you complete and submit the form with valid data (see Fig. 17-2), you will see the new product listed (see Fig. 17-3).

Figure 17-1. Submitting a form with invalid data.

Figure 17-2. Online Store – Refined Admin Panel – Products.

Figure 17-3. New product listed.

Another product store alternative

There is another product store alternative. **We will not codify this alternative into the application.** If you want to apply this alternative, we suggest you make an entire copy of the original project and apply the next changes over the project copy.

Let's discuss how the alternative works. Here is an excerpt of the *app/Http/Controllers/Admin/AdminProductController.php* , with some changes.

Analyze Code

```
public function store(Request $request)
{
```

```
        $request->validate([
                "name" => "required|max:255",
                "description" => "required",
                "price" => "required|numeric|gt:0",
                'image' => 'image',
        ]);

        $newProduct = new Product();
        $newProduct->setName($request->input('name'));
        $newProduct->setDescription($request->input('description'));
        $newProduct->setPrice($request->input('price'));
        $newProduct->setImage("game.png");
        $newProduct->save();

        $creationData = $request->only(["name","description","price"]);
        $creationData["image"] = "game.png";
        Product::create($creationData);

        return back();
    }
```

As you can see, we removed the product instance creation and the setters. Instead of that, we created an associative array called *creationData* with the *name* , *description* , and *price* collected from the form (through the *$request->only* method). Then, we added the *image* data to the array with a default image value (*game.png*). And, we invoked the *Product::create* method. To which we passed the *creationData* array, which contains the product data. The *create* method (which accepts an array of attributes) creates a model and inserts it into the database.

The advantage of this option is that we don't need to assign one by one each attribute. However, the disadvantage of this option is that it never invokes the classic setters that we defined in the *Product* model. Sometimes, setters are helpful to add functionalities to modify the way we want to store the model attributes.

Note: if you want to implement the previous strategy, you will also need to modify the *Product* model. You will need to include the following code in the *app/Models/Product.php* file. Remember, **we will not codify this alternative into the application.**

Analyze Code

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

    protected $fillable = [
        'name',
        'description',
        'price',
        'image',
    ];

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

Laravel protects us against mass assignment vulnerabilities. By default, we cannot create a new product by invoking the *create* method and passing an array with multiple data that refers to our model attributes. To deal with this problem, we must define an attribute called *fillable* in our model. Then, we specify which specific attributes can be assigned by the *create* method to create a new product. In this case, we will allow assigning *name* , *description* , *price* , and *image* data to create new products.

Note for advanced Laravel developers: there is an alternative to continue using the *Product::create* method, which at the same time invokes our classic setters. This option requires to create a "PHP trait" that overrides two Laravel methods, *hasSetMutator* and *setMutatedAttributeValue* and requires the inclusion of that trait in the models. Here is the repository link with that custom solution: https://github.com/PracticalBooks/Practical-Laravel/tree/main/Chapter17-Advanced/onlineStore and here you can find the specific code changes: https://github.com/PracticalBooks/Practical-Laravel/commit/b32a66463cbbd95789ebc17cf04f46a40ccc9ee7. We won't explain this strategy in detail. However, if you have any questions or a better solution, remember to use the discussion zone of the book repository.

Chapter 18 – Create Products with Images

In the previous chapter, we created all our products with a default *game.png* image. Let's see how to use the Laravel Storage system to upload our products' images.

Laravel Storage

Laravel provides a powerful filesystem abstraction thanks to the Flysystem PHP package by Frank de Jonge. The Laravel Flysystem integration provides simple drivers for working with local filesystems, SFTP, and Amazon S3 (https://laravel.com/docs/9.x/filesystem).

Laravel provides a class called *Storage* . This class contains a set of methods which allow creating, deleting, and moving files and directories. It also allows defining the kind of disk we want to interact with (i.e., local disk or Amazon S3).

Let's modify our admin product form to include selecting a product image, and later we will use the Laravel *Storage* class to store our images.

Modifying admin/product/index view

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

Modify Bold Code

```
@extends('layouts.admin')
@section('title', $viewData["title"])
@section('content')
...
    <form method="POST" action="{{ route('admin.product.store') }}" enctype="multipart/form-data">
      @csrf
      <div class="row">
      ...
      </div>
      <div class="row">
        <div class="col">
          <div class="mb-3 row">
                <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Image:</label>
                <div class="col-lg-10 col-md-6 col-sm-12">
                    <input class="form-control" type="file" name="image">
                </div>
          </div>
        </div>
        <div class="col">
           
        </div>
      </div>
      <div class="mb-3">
        <label class="form-label">Description</label>
...
```

We include an *enctype="multipart/form-data"* attribute in our form. This attribute is used in form elements that have a file upload. Then, we add a file upload input to allow admins to select product images.

Modifying AdminProductController

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

Modify Bold Code

```
<?php
...
use Illuminate \ Http \ Request;
use Illuminate \ Support \ Facades \ Storage;

class AdminProductController extends Controller
{
...
    public function store(Request $request)
    {
        ...
        $newProduct->save();

        if ($request->hasFile('image')) {
            $imageName = $newProduct->getId().".".$request->file('image')->extension();
            Storage::disk('public')->put(
                $imageName,
```

```
                    file_get_contents($request->file('image')->getRealPath())
                );
                $newProduct->setImage($imageName);
                $newProduct->save();
        }

        return back();
    }
}
```

We include the Laravel *Storage* library. Then, we check if an image was uploaded by using the *$request->hasFile* method. If an image was uploaded, we get the *newProduct id* and use it as a base to define the *imageName* . For example, if we create a new product and it is inserted with the id *8* , and we also upload an image for that product. Then, we will update the product image value with the product *id* , plus a *dot* , plus the image *extension* (i.e., *8.png*).

We also set the *Storage* with a *public* disk. This disk is intended for files that are going to be publicly accessible (as our product images). The *public* disk stores files in *storage/app/public* folder by default. The *put* method will store our product images over the *public* disk.

To make these files accessible from the web, we must create a "symbolic link" from *public/storage* to *storage/app/public* . Then, in the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan storage:link
```

Then, you will see something like the result in Fig. 18-1. We need to create this "symbolic link" because once we run our application (with *php artisan serve*), the users only can access files located inside *public/* folder. The rest of folders and files cannot be accessed.

```
PS C:\xampp\htdocs\onlineStore> php artisan storage:link
The [C:\xampp\htdocs\onlineStore\public\storage] link has been connected to [C:\xampp\htdocs\onlineStore\storage\app/public].
The links have been created.
```

Figure 18-1. Creating the storage symbolic link.

Modifying product/index and product/show views

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

Modify Bold Code

```
...
  @foreach ($viewData["products"] as $product)
  ...
        <img src="{{ asset('/storage/'.$product->getImage()) }}" class="card-img-top">
...
```

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

Modify Bold Code

```
...
    <div class="col-md-4">
        <img src="{{ asset('/storage/'.$viewData["product"]->getImage()) }}" class="img-fluid rounded-start">
    </div>
...
```

We are accessing the product images through the *storage* folder path.

Running the app

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

Execute in Terminal

```
php artisan serve
```

Go to the ("/admin/products") route, and you will see the new form (see Fig. 18-2). Then, complete the form, upload an image, and create a new product.

Figure 18-2. Online Store – Create products form with image selection.

Access to the ("/products") route. You will see the new product with its corresponding image. However, the old product images are not loading (see Fig. 18-3). It is because we changed the path from which we are loading the images. Don't worry. In the next chapter, we will implement a way to edit our products and upload proper images for the rest of our products.

Figure 18-3. Online Store – List products with their uploaded images.

Quick discussion: There is a tremendous advantage of using Laravel Storage for uploading product images. Before, we manually downloaded and added the products' images to the *public/img* folder. This manual task does not scale well. It requires access to the application code (but not all admins will have it). Now, we automated this process with the product registration form and Laravel Storage.

Chapter 19 – Edit and Delete Products

Let's start with deleting products since it is relatively simple. And then, we will implement the edit products functionality.

Deleting products

To delete a product, we will need to modify a set of files.

Modifying AdminProductController

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

Modify Bold Code

```
...
    public function delete($id)
    {
        Product::destroy($id);
        return back();
    }
}
```

We include a new *delete* method at the end of the *AdminProductController* class. This method invokes the model *destroy* method, which takes a primary key (*id*) as a parameter and deletes the corresponding object from the database. In the end, we redirect to the previous user location.

Modifying routes

In *routes/web.php* , make the following changes in **bold**.

Modify Bold Code

```
…
Route::post('/admin/products/store', 'App \ Http \ Controllers \ Admin \ AdminProductController@store')->name("admin.product.store");
Route::delete('/admin/products/{id}/delete', 'App \ Http \ Controllers \ Admin \ AdminProductController@delete')-
>name("admin.product.delete");
```

The new route ("/admin/products/{id}/delete") will invoke the *AdminProductController delete* method passing the *id* of the object to be deleted. It uses a *delete* HTTP method since the controller method will delete data.

Modifying layout/admin

In *resources/views/layout/admin.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
<html lang="en">


<head>
   <meta charset="utf-8" />
   <meta name="viewport" content="width=device-width, initial-scale=1" />
   <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.min.css" rel="stylesheet"
    crossorigin="anonymous" />
 <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.5.0/font/bootstrap-icons.css">
...
```

We add a link to a CSS file (Bootstrap icons). This CSS file allows us to use specific icons and fonts inside our views. For example, we will have icons to edit and delete our products. More information about Bootstrap icons can be found at: https://icons.getbootstrap.com/.

Modifying admin/product/index view

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

Modify Bold Code

```
...
        @foreach ($viewData["products"] as $product)
        <tr>
          <td>{{ $product->getId() }}</td>
          <td>{{ $product->getName() }}</td>
          <td>Edit
                <button class="btn btn-primary">
                    <i class="bi-pencil"></i>
                </button>
          </td>
      <td>Delete
```

```
        <form action="{{ route('admin.product.delete', $product->getId())}}" method="POST">
          @csrf
          @method('DELETE')
          <button class="btn btn-danger">
            <i class="bi-trash"></i>
          </button>
        </form>
            </td>
          </tr>
          @endforeach
...
```

We removed the “Edit” and “Delete” text and replaced them with a couple of buttons. Inside those buttons, we use Bootstrap icons (a *pencil* icon to edit a product and a *trash* icon to delete a product). We completed the delete functionality by wrapping a form around the delete icon. This form will invoke the *admin.product.delete* route, and will pass the current product *id* . Check that we use the *@method('DELETE')* Blade directive since the route uses the *delete* method.

Quick discussion: Laravel names controllers which handle CRUD (create-read-update-delete) operations as **Resource Controllers**. Laravel suggests using of some appropriated HTTP verbs (methods) to handle different actions for those controllers. Check:
- Use **GET** to list all elements, list a specific element, show a form to create an element, and show a form to edit an element.
- Use **POST** to store a new element into the database.
- Use **DELETE** to delete an element.
- Use **PUT/PATCH** to update an element.

Running the app

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

Execute in Terminal

```
php artisan serve
```

Now go to the (“/admin/products”) route, and you will see the new form (see Fig. 19-1). Now, try to delete a product.

Figure 19-1. Online Store – Refined Admin Panel – Products.

Editing products

Like the previous delete functionality, we will need to modify a set of files.

Modifying AdminProductController

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

Modify Bold Code

```
…

    public function edit($id)
    {
        $viewData = [];
        $viewData["title"] = "Admin Page - Edit Product - Online Store";
        $viewData["product"] = Product::findOrFail($id);
        return view('admin.product.edit')->with("viewData", $viewData);
```

```
    }

    public function update(Request $request, $id)
    {
        $request->validate([
                "name" => "required|max:255",
                "description" => "required",
                "price" => "required|numeric|gt:0",
                'image' => 'image',
        ]);

        $product = Product::findOrFail($id);
        $product->setName($request->input('name'));
        $product->setDescription($request->input('description'));
        $product->setPrice($request->input('price'));

        if ($request->hasFile('image')) {
                $imageName = $product->getId().".".$request->file('image')->extension();
                Storage::disk('public')->put(
                        $imageName,
                        file_get_contents($request->file('image')->getRealPath())
                );
                $product->setImage($imageName);
        }

        $product->save();
        return redirect()->route('admin.product.index');
    }
}
```

Let’s analyze the previous code by parts.

Analyze Code

```
    public function edit($id)
    {
        $viewData = [];
        $viewData["title"] = "Admin Page - Edit Product - Online Store";
        $viewData["product"] = Product::findOrFail($id);
        return view('admin.product.edit')->with("viewData", $viewData);
    }
```

We have an *edit* method that searches for a product based on its *id* , and sends it to the *admin.product.edit* view. It is the product we are going to edit.

Analyze Code

```
    public function update(Request $request, $id)
    {
        $request->validate([
                "name" => "required|max:255",
                "description" => "required",
                "price" => "required|numeric|gt:0",
                'image' => 'image',
        ]);

        $product = Product::findOrFail($id);
        $product->setName($request->input('name'));
        $product->setDescription($request->input('description'));
        $product->setPrice($request->input('price'));

        if ($request->hasFile('image')) {
                $imageName = $product->getId().".".$request->file('image')->extension();
                Storage::disk('public')->put(
                        $imageName,
                        file_get_contents($request->file('image')->getRealPath())
                );
                $product->setImage($imageName);
        }

        $product->save();
        return redirect()->route('admin.product.index');
    }
```

Then, we have the *update* method. It is like the *store* method.

1 We collect the *request* and the *id* of the product to be updated.
2 We search for a product based on that *id* , and to that product, we set the new *name* , *price* , and *description* . That data is collected in a form that we will show later.

3 We set the new product *image* value if a new *image* was uploaded.
4 Finally, we *save* the new product data, and we *redirect* to the *admin.product.index* route (here is where we list all products).

Modifying routes

In *routes/web.php* , make the following changes in **bold**.

Modify Bold Code

```
…
Route::delete('/admin/products/{id}/delete', 'App \ Http \ Controllers \ Admin \ AdminProductController@delete')->name("admin.product.delete");
Route::get('/admin/products/{id}/edit', 'App \ Http \ Controllers \ Admin \ AdminProductController@edit')->name("admin.product.edit");
Route::put('/admin/products/{id}/update', 'App \ Http \ Controllers \ Admin \ AdminProductController@update')-
>name("admin.product.update");
```

We have two new routes. The (“/admin/products/{id}/edit”) will be used to show a form in which we enter the new product data. It uses a *get* HTTP method since the controller method will show a form to edit a product. The (“/admin/products/{id}/update”) will be used to update the product data. It uses a *put* HTTP method since the controller method will update product data.

Modifying admin/product/index view

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

Modify Bold Code

```
@extends('layouts.admin')
@section('title', $viewData["title"])
@section('content')
...
        @foreach ($viewData["products"] as $product)
        <tr>
          <td>{{ $product->getId() }}</td>
          <td>{{ $product->getName() }}</td>
          <td>
                <a class="btn btn-primary" href="{{route('admin.product.edit', ['id'=> $product->getId()])}}">
                    <button class="btn btn-primary">
                        <i class="bi-pencil"></i>
                    </button>
                </a>
          </td>
...
```

Now, we link the pencil icon with the *admin.product.edit* route and pass in the current product *id* .

Creating admin/product/edit view

In *resources/views/admin/product* , create a new file *edit.blade.php* and fill it with the following code.

Add Entire Code

```
@extends('layouts.admin')
@section('title', $viewData["title"])
@section('content')
<div class="card mb-4">
<div class="card-header">
  Edit Product
</div>
<div class="card-body">
  @if($errors->any())
  <ul class="alert alert-danger list-unstyled">
    @foreach($errors->all() as $error)
    <li>- {{ $error }}</li>
    @endforeach
  </ul>
  @endif

  <form method="POST" action="{{ route('admin.product.update', ['id'=> $viewData['product']->getId()]) }}"
    enctype="multipart/form-data">
    @csrf
    @method('PUT')
    <div class="row">
      <div class="col">
        <div class="mb-3 row">
          <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Name:</label>
          <div class="col-lg-10 col-md-6 col-sm-12">
            <input name="name" value="{{ $viewData['product']->getName() }}" type="text" class="form-control">
          </div>
        </div>
      </div>
```

```
    <div class="col">
      <div class="mb-3 row">
        <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Price:</label>
        <div class="col-lg-10 col-md-6 col-sm-12">
          <input name="price" value="{{ $viewData['product']->getPrice() }}" type="number" class="form-control">
        </div>
      </div>
          </div>
        </div>
        <div class="row">
          <div class="col">
            <div class="mb-3 row">
                    <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Image:</label>
                    <div class="col-lg-10 col-md-6 col-sm-12">
                        <input class="form-control" type="file" name="image">
                    </div>
            </div>
          </div>
          <div class="col">
             
          </div>
        </div>
        <div class="mb-3">
          <label class="form-label">Description</label>
          <textarea class="form-control" name="description"
            rows="3">{{ $viewData['product']->getDescription() }}</textarea>
        </div>
        <button type="submit" class="btn btn-primary">Edit</button>
      </form>
    </div>
  </div>
@endsection
```

The previous view is like the form created in the *admin/product/index* view but with minor differences. We use the PUT method (since we are updating a resource) and populate the input values with the product attributes.

Running the app

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

Execute in Terminal

```
php artisan serve
```

Go to the ("/admin/products") route, click an edit button of a specific product, and you will see the edit form (see Fig. 19-2). Now, you can edit products. We suggest editing all products to replace the missing images (see Fig. 19-3).

Figure 19-2. Editing a product.

Figure 19-3. List products with proper images.

Chapter 20 – Refactoring Validations

We have some duplicated codes in our *AdminProductController* class. For example, we have the same validations in the *store* and *update* methods. Placing validations in the controllers is not always a good strategy. If you need to make validations for values related with model attributes, we suggest moving them away from controllers. Because that is data that could be validated across multiple controllers and classes (as seen in the previous example).

There are multiple strategies to deal with model validations. For example, you can use the *request->validate* method, create separated Laravel *Form Requests* classes, and create custom *Validators* (https://laravel.com/docs/9.x/validation). We will implement a simple strategy. It will work well for our needs, and indeed, much better than the strategy of the previous chapter. We will move model validations to the corresponding model files and invoke those validations from the controllers. Let's see this strategy in action.

Refactoring Product model

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

Modify Bold Code

```
<?php
...
class Product extends Model
{
    ...
     * $this->attributes['updated_at'] - timestamp - contains the product update date
     */

    public static function validate($request)
    {
        $request->validate([
                "name" => "required|max:255",
                "description" => "required",
                "price" => "required|numeric|gt:0",
                'image' => 'image',
        ]);
    }
...
```

We moved the duplicated controller validations to a single place. We placed them inside a static method called *validate* inside our *Product* model class. Now, we have a unique representation of our *Product* validations, which will improve our code maintainability.

TIP: Always try to move your model validations away from the controllers. You can place them in methods inside your models or into separated classes. Even you can include a new rule (in your architectural rules document) saying that models' validations must be done in a model method called *validate* or in the location of your preference (but not in controllers).

Refactoring AdminProductController

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

Modify Bold Code

```
<?php
...
class AdminProductController extends Controller
{
    ...
    public function store(Request $request)
    {
        Product::validate($request);
        $request->validate([
                "name" => "required|max:255",
                "description" => "required",
                "price" => "required|numeric|gt:0",
                'image' => 'image',
        ]);
        ...
    }
    ...
    public function update(Request $request, $id)
    {
        Product::validate($request);
        $request->validate([
```

```
                    ~~"name" => "required|max:255",~~
                    ~~"description" => "required",~~
                    ~~"price" => "required|numeric|gt:0",~~
                    ~~'image' => 'image',~~
        ~~]);~~
        ...
    }
...
```

We removed the previous validation logic, which was inside our *store* and *update* methods. Instead of that, now we invoke the *validate* method of our *Product* model.

Finally, you can run the application and check that the validations are working as expected.

Chapter 21 – Login System

Currently, anyone can access the admin panel and create, edit, and delete products. In a real-world setting, this obviously should not be the way. So, let's implement a login system.

Laravel authentication systems

Most frameworks provide a library or package to handle authentications. Even if it is an unopinionated framework, you can find third-party libraries for these types of systems.

Laravel 6.* and 7.* provided an official authentication system called *laravel/ui* (https://laravel.com/docs/7.x/authentication). *laravel/ui* is a straightforward authentication system built on the Bootstrap CSS framework (https://github.com/laravel/ui). This library was created by Taylor Otwell (the creator of Laravel). Later, this library provided support to create an authentication system with Vue and React. However, *laravel/ui* is no longer the official authentication system for Laravel 8.* and 9.*.

Laravel 9.* have three official authentication systems (https://laravel.com/docs/9.x/starter-kits):

- **Breeze:** is a simple authentication system based on Blade templates styled with Tailwind CSS.
- **Breeze & Inertia:** Breeze also offers an Inertia.js frontend implementation powered by Vue or React.
- **Jetstream:** augments functionalities with more robust features and additional frontend technology stacks. Jetstream is designed using Tailwind CSS and offers your choice of Livewire or Inertia.js driven frontend scaffolding.

If we want to use some of the three previous options to implement the authentication system, we will need to use additional CSS frameworks or JavaScript frameworks (such as React or Vue) which is out of the scope of this book. Fortunately, *laravel/ui* continues to be available and supports for Laravel 9.* applications.

Installing laravel/ui authentication system

Installing laravel/ui

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

Execute in Terminal

```
composer require laravel/ui
```

The previous command includes the *laravel/ui* library over our *composer.json* file. And it installs that library in our *vendor* folder.

We also need to generate the frontend scaffolding and the login system. In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan ui bootstrap --auth
```

You will be asked to replace the *app* layout and the *HomeController* , type "no" to both questions (see Fig. 21-1).

```
PS C:\xampp\htdocs\onlineStore> php artisan ui bootstrap --auth
Bootstrap scaffolding installed successfully.
Please run "npm install && npm run dev" to compile your fresh scaffolding.

 The [layouts/app.blade.php] view already exists. Do you want to replace it? (yes/no) [no]:
 > no

 The [HomeController.php] file already exists. Do you want to replace it? (yes/no) [no]:
 > no

Authentication scaffolding generated successfully.
```

Figure 21-1. Configuring laravel/ui.

laravel/ui creates:

- The *app/Http/Controllers/Auth* folder that includes some authentication controllers such as *LoginController* and *RegisterController* .
- The *resources/views/auth* folder that includes some authentication views such as *login* and *register* . Those views are generated with simple HTML and Bootstrap code.
- A modification over the *web.php* file that includes the *Auth* routes and a */home* route.

Customizing the authentication system

Let’s make some changes to finalize the inclusion of the authentication system over our Online Store application.

Modifying web.php

In *routes/web.php* , remove the following line in **bold** (at the end of the file).

Modify Bold Code

```
...

Auth::routes();

Route::get('/home', [App/Http/Controllers/HomeController::class, 'index'])->name('home');
```

laravel/ui creates a default (“/home”) route. However, our application does not have a (“/home”) route, so we remove it.

Modifying RouteServiceProvider

In *app/Providers/RouteServiceProvider.php* , we replace the *HOME* attribute for the next one in **bold**.

Modify Bold Code

```
<?php
...
class RouteServiceProvider extends ServiceProvider
{
    ...
    public const HOME = '/';
...
```

The value of the *HOME* attribute is used by Laravel auth to redirect users after login. Since we don’t have a (“/home”) route, we replace it with the main route (“/”).

Modifying app.blade.php

In *resources/views/layouts/app.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
...
        <div class="navbar-nav ms-auto">
          <a class="nav-link active" href="{{ route('home.index') }}">Home</a>
          <a class="nav-link active" href="{{ route('product.index') }}">Products</a>
          <a class="nav-link active" href="{{ route('home.about') }}">About</a>
          <div class="vr bg-white mx-2 d-none d-lg-block"></div>
          @guest
          <a class="nav-link active" href="{{ route('login') }}">Login</a>
          <a class="nav-link active" href="{{ route('register') }}">Register</a>
          @else
          <form id="logout" action="{{ route('logout') }}" method="POST">
                <a role="button" class="nav-link active"
                     onclick="document.getElementById('logout').submit();">Logout</a>
                @csrf
          </form>
          @endguest
        </div>
...
```

Blade provides *@auth* and *@guest* directives to determine if the current user is authenticated(*@auth*) or is a guest(*@guest*). If the user is a guest, we will show the *register* and *login* links. On the other hand, if the user is authenticated, we will show the *logout* link. All these links are connected to *auth* routes.

Modifying buttons in views/auth/login and views/auth/register

In *resources/views/auth/login.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
…
                <div class="col-md-8 offset-md-4">
                    <button type="submit" class="btn bg-primary text-white">
                        {{ __('Login') }}
                    </button>

                    @if (Route::has('password.request'))
                        <a class="btn bg-primary text-white" href="{{ route('password.request') }}">
                            {{ __('Forgot Your Password?') }}
```

```
                                        </a>
                                    @endif
                                </div>
…
```

In *resources/views/auth/register.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
…
                                <div class="form-group row mb-0">
                                    <div class="col-md-6 offset-md-4">
                                        <button type="submit" class="btn bg-primary text-white">
                                            {{ __('Register') }}
                                        </button>
                                    </div>
                                </div>
…
```

Running the app

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

Execute in Terminal

```
php artisan serve
```

Go to the ("/register") route, you will see the registration form and the new links over the navigation bar (see Fig. 21-2). First, create a new user. Then, you will be redirected to the home page as an authenticated user. Then, you will see the *logout* link (see Fig. 21-3). Finally, you can click the *logout* link and test the *login* system with your user credentials (see Fig. 21-4).

Figure 21-2. Online Store – Register page.

Figure 21-3. Online Store – Home page for authenticated users.

Figure 21-4. Online Store – Login page.

Users are registered in the *users* table. This table was created the first time we ran Laravel Migrations. Laravel also includes a *User* model. We will analyze this model in the next chapter.

Note: you can find the application code at the GitHub repository in https://github.com/PracticalBooks/Practical-Laravel.

Chapter 22 – Refactoring User

We successfully implemented our login system in the last chapter. In this chapter, we will apply some changes to the User model to improve our Online Store application and manage user roles.

Alter user migration

Let's create a migration to add two columns to the *users* table. In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan make:migration alter_users_table
```

The previous command creates a new migration inside the *database/migrations* folder. Open the generated file (something like *2022_02_12_140820_alter_users_table.php*) and replace the *up* and *down* methods with the following.

Modify Bold Code

```
<?php
...
return new class extends Migration
{
...
    public function up()
    {
        Schema::table('users', function (Blueprint $table) {
                $table->string('role')->default('client');
                $table->integer('balance');
        });
    }
...
    public function down()
    {
        Schema::table('users', function (Blueprint $table) {
                $table->dropColumn(['role']);
                $table->dropColumn(['balance']);
        });
    }
}
```

The *up* method adds two columns to the *users* table. The *role* column will be used to know if a user is an *admin* or *client* . *client* will be the default value. *admin* users will be able to access the admin panel. In the *balance* column, we will store the user's balance. The user's balance represents the user's money inside the application (we will use it for the purchase functionality).

The *down* method removes the *role* and *balance* columns.

Let's execute the previous migration, go to the project directory, and in the Terminal, execute the following command (see Fig. 22-1).

Execute in Terminal

```
php artisan migrate
```

Figure 22-1. Execution of alter users table migration.

Refactoring User model

Let's refactor the default Laravel User model. In *app/Models/User.php* , make the following changes in **bold**.

Modify Bold Code

```
<?php
...
class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;

    /**
     * USER ATTRIBUTES
     * $this->attributes['id'] - int - contains the user primary key (id)
     * $this->attributes['name'] - string - contains the user name
```

```
 * $this->attributes['email'] - string - contains the user email
 * $this->attributes['email_verified_at'] - timestamp - contains the user email verification date
 * $this->attributes['password'] - string - contains the user password
 * $this->attributes['remember_token'] - string - contains the user password
 * $this->attributes['role'] - string - contains the user role (client or admin)
 * $this->attributes['balance'] - int - contains the user balance
 * $this->attributes['created_at'] - timestamp - contains the user creation date
 * $this->attributes['updated_at'] - timestamp - contains the user update date
 */


...

protected $fillable = [
    'name',
    'email',
    'password',
    'balance',
];

...

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 getEmail()
{
    return $this->attributes['email'];
}

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

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

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

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

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

public function getBalance()
{
    return $this->attributes['balance'];
}
```

```
    public function setBalance($balance)
    {
        $this->attributes['balance'] = $balance;
    }

    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;
    }
}
```

We included the *User* attributes block of comments and added some getters and setters. We also added the *balance* in the *fillable* attribute.

TIP: We made some improvements to the *User* model. Now, a developer can know the available *User* attributes. In addition, we leave the code a little cleaner than we found it. The Boy Scouts of America have a simple rule that we can apply to our profession. *Leave the campground cleaner than you found it (2008 - Martin, R. C.- Clean Code).*

Modifying the RegisteredUserController

Finally, let's modify the auth *RegisterController* . In *app/Http/Controllers/Auth/RegisterController.php* , make the following changes in **bold**.

Modify Bold Code

```
<?php
...
class RegisterController extends Controller
{
...
    protected function create(array $data)
    {
        return User::create([
                'name' => $data['name'],
                'email' => $data['email'],
                'password' => Hash::make($data['password']),
                'balance' => 5000,
        ]);
    }
...
```

We modified the way users are created. When a new user is created, we will set the user's *balance* to 5000. It is just an example. Later, you can improve this application to connect with a payment system and properly update the user's *balance* .

Running the app

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

Execute in Terminal

```
php artisan serve
```

Now go to the ("/register") route and create a new user (remember you must be logged out to access the "/register" route). You will see in the database that the new user has *5000* in *balance* and a *client* role (see Fig. 22-2).

created_at	updated_at	role	balance
2022-02-12 13:53:31	2022-02-12 13:53:31	client	0
2022-02-12 14:15:54	2022-02-12 14:15:54	client	5000

Figure 22-2. New columns and rows in the users table.

Note: remember to update the old users' balances if you want.

Defining admins

If you check your users table, you will see that all users are clients. We don't have admins. Here we have two options.

- Go to phpMyAdmin, click the *online_store* database, click the *users* table, browse a specific user, and edit the user's *role* (change it from *client* to *admin*). Check Fig. 22-3.
- Use Laravel Tinker and create a user from scratch.

created_at	updated_at	role	balance
2022-02-12 13:53:31	2022-02-12 13:53:31	admin	0
2022-02-12 14:15:54	2022-02-12 14:15:54	client	5000

Figure 22-3. Editing a user's role.

Laravel Tinker

Laravel **Tinker** is a powerful REPL (Read–Eval–Print Loop) for the Laravel framework (https://laravel.com/docs/9.x/artisan#tinker). Tinker allows you to interact with your Laravel application on the command line.

Let's use Laravel Tinker. In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan tinker
```

You will see a command line where we will create an admin user. Type the following commands, and you will get a result like in Fig. 22-4.

Execute in Terminal

```
$user = new App \ Models \ User();
$user->setName('Daniel');
$user->setEmail('daniel@danielgara.com');
$user->setPassword(bcrypt('passwordVerySecret'));
$user->setBalance(5000);
$user->setRole('admin');
$user->save();
exit;
```

Figure 22-4. Create a user with Laravel Tinker.

We created a new user, hashed the password with *bcrypt()* for security, and saved the user to the database. Now, you can log in to the application with the previous credentials.

TIP: You can use Tinker to make quick verifications. For example, in Tinker, you can enter the following command (*$products = App\Models\Product::all();*) to check the stored products in the database.

Chapter 23 – AdminAuthMiddleware

Let's restrict the access to the admin panel just for admins. Laravel provides a set of options to operationalize this functionality. A couple of them are called *gates* and *policies* (https://laravel.com/docs/9.x/authorization). We won't use them. Instead, we will use another approach, a Laravel Middleware.

Laravel Middleware

Laravel **middleware** provides a convenient mechanism for inspecting and filtering HTTP requests entering your application. For example, Laravel includes a middleware that verifies your application's user is authenticated. If the user is not authenticated, the middleware will redirect the user to your application's login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application. Additional middleware can be written to perform a set of tasks besides authentication. For example, a logging middleware might log all incoming requests to your application.

We will create a middleware to verify that only *admin* users can access the admin panel. Otherwise, we will redirect them to the home page.

AdminAuthMiddleware

To create a new middleware, in the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan make:middleware AdminAuthMiddleware
```

Let's modify the new middleware. In *app/Http/Middleware/AdminAuthMiddleware.php* , make the following changes in **bold**.

Modify Bold Code

```
<?php
…
use Closure;
use Illuminate \ Http \ Request;
use Illuminate \ Support \ Facades \ Auth;


class AdminAuthMiddleware
{
	...
	public function handle(Request $request, Closure $next)
	{
		if (Auth::user() && Auth::user()->getRole() == 'admin') {
			return $next($request);
		} else {
			return redirect()->route('home.index');
		}
	}
}
```

The middleware checks if the user is authenticated and is an *admin* . If that is true, the application continues its normal execution flow. Otherwise, we redirect to the *home.index* route.

Registering AdminAuthMiddleware

We will use *AdminAuthMiddleware* to restrict access to admin routes. So, we will need to register the middleware in our application's *app/Http/Kernel.php* file. In *app/Http/Kernel.php* , make the following changes in **bold**.

Modify Bold Code

```
…
	protected $routeMiddleware = [
		'auth' => \  App \ Http \ Middleware \ Authenticate::class,
		'auth.basic' => \  Illuminate \ Auth \ Middleware \ AuthenticateWithBasicAuth::class,
		'cache.headers' => \  Illuminate \ Http \ Middleware \ SetCacheHeaders::class,
		'can' => \  Illuminate \ Auth \ Middleware \ Authorize::class,
		'guest' => \  App \ Http \ Middleware \ RedirectIfAuthenticated::class,
		'password.confirm' => \  Illuminate \ Auth \ Middleware \ RequirePassword::class,
		'signed' => \  Illuminate \ Routing \ Middleware \ ValidateSignature::class,
		'throttle' => \  Illuminate \ Routing \ Middleware \ ThrottleRequests::class,
		'verified' => \  Illuminate \ Auth \ Middleware \ EnsureEmailIsVerified::class,
		'admin' => \  App \ Http \ Middleware \ AdminAuthMiddleware::class,
	];
}
```

We included the *AdminAuthMiddleware* in the *$routeMiddleware* attribute. This attribute contains the available

middleware for our route system.

Modifying web.php

Let's connect the previous middleware with the routes we want to restrict. In *routes/web.php* , make the following changes in **bold**.

Modify Bold Code

```
…
Route::get('/products/{id}', 'App \ Http \ Controllers \ ProductController@show')->name("product.show");


Route::middleware('admin')->group(function () {
    Route::get('/admin', 'App \ Http \ Controllers \ Admin \ AdminHomeController@index')->name("admin.home.index");
    Route::get('/admin/products', 'App \ Http \ Controllers \ Admin \ AdminProductController@index')->name("admin.product.index");
    Route::post('/admin/products/store', 'App \ Http \ Controllers \ Admin \ AdminProductController@store')->name("admin.product.store");
    Route::delete('/admin/products/{id}/delete', 'App \ Http \ Controllers \ Admin \ AdminProductController@delete')->name("admin.product.delete");
    Route::get('/admin/products/{id}/edit', 'App \ Http \ Controllers \ Admin \ AdminProductController@edit')->name("admin.product.edit");
    Route::put('/admin/products/{id}/update', 'App \ Http \ Controllers \ Admin \ AdminProductController@update')->name("admin.product.update");
});


Auth::routes();
```

We grouped all ("/admin/*") routes around the new middleware. It means that any *guest* or *client user* who tries to access any ("/admin/*") route will be redirected to the home page. Only *admin* users can access those routes.

Running the app

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

Execute in Terminal

```
php artisan serve
```

Now go to the ("/admin") as a *guest* or *client user* , and you will be redirected to the home page.

Chapter 24 – Introduction to Web Session

Do you know how the login systems works? How does the application know that I'm connected to the application? How does the application know that it should show the logout button for a logged user, but to my friend who is not connected, it should show the login button? For how long will the application know that I am connected?

In this chapter, we will try to solve the previous questions and understand why web session is important in the construction of web applications.

HTTP protocol limitations

For now, we have relied on the HTTP protocol to communicate with our Online Store. For example, if we want to get products information, we access http://127.0.0.1:8000/products. To log in, we access http://127.0.0.1:8000/login. Each of our requests uses the HTTP protocol as a means of communication.

However, the HTTP protocol has some limitations. HTTP is a **stateless** protocol, meaning that the server does not keep any data (state) between two requests. In HTTP, every request creates a new connection, and each new request knows nothing about any prior requests. That is, it holds no state data.

But if you remember, when you are logged into the application, the following requests will show you the logout button (which means that the application is using a mechanism to identify you) to keep state data. It is because Laravel increases the HTTP protocol capabilities using Laravel sessions.

Laravel sessions

Laravel sessions provide a way to store information about the user across multiple requests. User information is typically placed in a persistent store that can be accessed from subsequent requests. By default, Laravel uses a *file* session driver. This driver stores the session data in files inside the *storage/framework/sessions* folder.

Let's understand how web sessions work.

Web sessions

A web session is a series of contiguous actions by a visitor on a website within a given time frame. Each framework provides its mechanism to implement sessions (to track the visitors' actions). Figure 24-1 shows how Laravel sessions work.

Figure 24-1. Typical Laravel sessions operation.

Let's analyze this process in detail:

1 The user goes to the home page (http://127.0.0.1:8000/). Laravel checks if this is the first time the user accesses the application. If that is true, Laravel generates a session id for the user (current request). Laravel also creates a new file (inside the *storage/framework/sessions* folder) linked to the generated session id (see Fig. 24-2). Laravel stores the user state data in the previous file.

2 Laravel sends an HTTP response to the user with the home page (with the corresponding HTML, CSS, and JavaScript code), which will be rendered in the user browser. Laravel sends the session id inside the HTTP response. This session id will be stored inside a cookie (called *laravel_session*) in the user browser (see Fig. 24-3). The user browser will send the cookie information (the session id) in subsequent user requests to the server.

3 The user goes to the login page (http://127.0.0.1:8000/login).

4 Laravel sends an HTTP response to the user with the login form.

5 The user completes the login form and clicks the login button. Laravel verifies the user data, and if the data is correct, Laravel updates the session id for logged user. It also updates the session file including some keys indicating that the user is logged. Note: sometimes Laravel removes and re-generates both session id and session file. This new session id will be stored inside the *laravel_session* cookie in the user browser. Finally, Laravel redirects the user to the home page.

6 Laravel sends an HTTP response to the user with the home page. Since the user is logged (based on the information from the session file), it displays the logout button.

7 All subsequent requests will display the navigation menu with the logout button since the user requests include the session id. Laravel verifies that the corresponding session file contains the keys indicating that the user is logged. It works this way until the user clicks logout (which generates a new session file and new session id) or until the session expires (by default, it is 120 minutes).

Figure 24-2. Generated session file.

Figure 24-3. Generated cookie (Chrome DevTools).

We have seen how the Laravel session works. But sessions are not only used for login systems. Sessions can also store flash messages, CSRF tokens, the last page visited by the user, and even products added to a cart. In addition, you can use sessions to store temporary data that is useful to track the current user. Data that you don't need to

persist permanently. If you need to store data permanently, you should use Laravel Eloquent instead of Laravel sessions.

In the next chapter, we will see how to use Laravel sessions to design a shopping cart.

Chapter 25 – Shopping Cart

Let’s use web sessions to implement a shopping cart.

CartController

In *app/Http/Controllers* , create a new file *CartController.php* and fill it with the following code.

Add Entire Code

```
<?php

namespace App \ Http \ Controllers;

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

class CartController extends Controller
{
	public function index(Request $request)
	{
		$total = 0;
		$productsInCart = [];

		$productsInSession = $request->session()->get("products");
		if ($productsInSession) {
			$productsInCart = Product::findMany(array_keys($productsInSession));
			$total = Product::sumPricesByQuantities($productsInCart, $productsInSession);
		}

		$viewData = [];
		$viewData["title"] = "Cart - Online Store";
		$viewData["subtitle"] =  "Shopping Cart";
		$viewData["total"] = $total;
		$viewData["products"] = $productsInCart;
		return view('cart.index')->with("viewData", $viewData);
	}

	public function add(Request $request, $id)
	{
		$products = $request->session()->get("products");
		$products[$id] = $request->input('quantity');
		$request->session()->put('products', $products);

		return redirect()->route('cart.index');
	}

	public function delete(Request $request)
	{
		$request->session()->forget('products');
		return back();
	}
}
```

Let’s analyze the previous code by parts.

Analyze Code

```
	public function add(Request $request, $id)
	{
		$products = $request->session()->get("products");
		$products[$id] = $request->input('quantity');
		$request->session()->put('products', $products);

		return redirect()->route('cart.index');
	}
```

The *add* method receives the *request* (which receives the quantity of product) and the product *id* (the *id* of the product to be added to the cart). Then, we get the products stored in the session through the *request->session()->get("products")* method. The first time, *request->session()->get("products")* won’t exist, so we assign it to an empty object. Next, we include in *products* variable the collected product *id* with its *quantity* (*id* as key, *quantity* as value). We then update the products stored in the session (with the use of the *request->session()->put* method). Finally, we redirect the user to the *cart.index* route.

Analyze Code

```
    public function delete(Request $request)
    {
        $request->session()->forget('products');
        return back();
    }
```

The *delete* method receives the *request* and removes the products stored in the session for that request (using the *request->session()->forget* method). Then, we return to the previous route.

Analyze Code

```
    public function index(Request $request)
    {
        $total = 0;
        $productsInCart = [];

        $productsInSession = $request->session()->get("products");
        if ($productsInSession) {
                $productsInCart = Product::findMany(array_keys($productsInSession));
                $total = Product::sumPricesByQuantities($productsInCart, $productsInSession);
        }

        $viewData = [];
        $viewData["title"] = "Cart - Online Store";
        $viewData["subtitle"] =  "Shopping Cart";
        $viewData["total"] = $total;
        $viewData["products"] = $productsInCart;
        return view('cart.index')->with("viewData", $viewData);
    }
```

The *index* method defines a *total* variable with a zero value and an empty *productsInCart* array. First, we check if the current *request* has products stored in session. If there are *productsInSession* , we extract the related products from the database. In this case, we use the model *findMany* method, which receives an array with primary keys and returns a collection of objects. We send *array_keys($productsInSession)* to this method, remember we store the products id as keys and the quantities as values. Then, we update the *total* value by invoking the *Product::sumPricesByQuantities* method (which will be implemented next). Finally, we send the *total* and *products* to the *cart.index* view.

More information about Laravel session methods can be found at: https://laravel.com/docs/9.x/session.

Modifying Product model

In *app/Models/Product.php* , make the following changes in **bold**.

Modify Bold Code

```
...
    public static function sumPricesByQuantities($products, $productsInSession)
    {
        $total = 0;
        foreach ($products as $product) {
                $total = $total + ($product->getPrice()*$productsInSession[$product->getId()]);
        }

        return $total;
    }

    public function getId()
...
```

We include a new static method called *sumPricesByQuantities* . *sumPricesByQuantities* receives the Eloquent products' models added in the cart and the information of products stored in session. It iterates over the products and calculates the *total* to be paid (based on the *price* of each product and its corresponding *quantity*). It then returns the *total* to be paid.

Modifying web.php

In *routes/web.php* , make the following changes in **bold**.

Modify Bold Code

```
…
Route::get('/products/{id}', 'App \ Http \ Controllers \ ProductController@show')->name("product.show");

Route::get('/cart', 'App \ Http \ Controllers \ CartController@index')->name("cart.index");
Route::get('/cart/delete', 'App \ Http \ Controllers \ CartController@delete')->name("cart.delete");
Route::post('/cart/add/{id}', 'App \ Http \ Controllers \ CartController@add')->name("cart.add");
```

```
...
```

We included three new routes which are linked to the corresponding *CartController* methods.

Modifying app.blade.php

In *resources/views/layouts/app.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
...
        <a class="nav-link active" href="{{ route('home.index') }}">Home</a>
        <a class="nav-link active" href="{{ route('product.index') }}">Products</a>
        <a class="nav-link active" href="{{ route('cart.index') }}">Cart</a>
        <a class="nav-link active" href="{{ route('home.about') }}">About</a>
...
```

We added a new link to the cart page.

Modifying product/show view

In *resources/views/product/show.blade.php* , make the following changes in **bold** (replace the "Add to cart" paragraph).

Modify Bold Code

```
...
    <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>~~
      <p class="card-text">
      <form method="POST" action="{{ route('cart.add', ['id'=> $viewData['product']->getId()]) }}">
        <div class="row">
            @csrf
            <div class="col-auto">
              <div class="input-group col-auto">
                <div class="input-group-text">Quantity</div>
                <input type="number" min="1" max="10" class="form-control quantity-input" name="quantity" value="1">
              </div>
            </div>
            <div class="col-auto">
              <button class="btn bg-primary text-white" type="submit">Add to cart</button>
            </div>
        </div>
      </form>
      </p>
    </div>
...
```

We add a new form where the user enters the product's quantity to the cart. This form is linked to the *cart.add* route.

Cart index view

In *resources/views/* , create a subfolder *cart* . Then, in *resources/views/cart* , create a new file *index.blade.php* and fill it with the following code.

Add Entire Code

```
@extends('layouts.app')
@section('title', $viewData["title"])
@section('subtitle', $viewData["subtitle"])
@section('content')
<div class="card">
  <div class="card-header">
    Products in Cart
  </div>
  <div class="card-body">
    <table class="table table-bordered table-striped text-center">
      <thead>
        <tr>
          <th scope="col">ID</th>
          <th scope="col">Name</th>
          <th scope="col">Price</th>
          <th scope="col">Quantity</th>
        </tr>
      </thead>
      <tbody>
```

```
            @foreach ($viewData["products"] as $product)
            <tr>
                <td>{{ $product->getId() }}</td>
                <td>{{ $product->getName() }}</td>
                <td>${{ $product->getPrice() }}</td>
                <td>{{ session('products')[$product->getId()] }}</td>
            </tr>
            @endforeach
        </tbody>
    </table>
    <div class="row">
        <div class="text-end">
            <a class="btn btn-outline-secondary mb-2"><b>Total to pay:</b> ${{ $viewData["total"] }}</a>
            <a class="btn bg-primary text-white mb-2">Purchase</a>
            <a href="{{ route('cart.delete') }}">
                <button class="btn btn-danger mb-2">
                        Remove all products from Cart
                </button>
            </a>
        </div>
    </div>
  </div>
</div>
@endsection
```

This view is like the *admin/product/index* view. We iterate and display the products added in session. We also use the *session* global helper to access the products' quantities. We then display the *total* to be paid, a *purchase* button (which doesn't do anything yet), and a button to remove all products from the cart linking the *cart.delete* route.

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, click a specific product, and you will see the new *product/show* view (see Fig. 25-1). Next, you can add some products to the cart and visit the ("/cart") route. It will show you the total to be paid and a button to remove all products from the cart (see Fig. 25-2).

Figure 25-1. Product show view with add to cart button.

Figure 25-2. Online Store – Shopping cart page.

Chapter 26 – Orders and Items

To complete our shopping cart, we must be able to make purchases. However, we need to set in place some things before adding the purchase functionality.

Orders and Items

Let's take Fig. 26-1 as a base. Though a simplified invoice, it is helpful to show the data we need to store.

Order #1

Date: 2021-10-14

Total: $1060

User id - name: 1 - Daniel

Qty	Product id - name	Price
1	1 - TV	1000
2	3 - Chromecast	30

Figure 26-1. Example of an invoice.

For the **Order**, we need to store:

- **Id:** in the example, it is *#1* .
- **Date:** in the example, it is *2021-10-14* .
- **Total:** in the example, it is *1060* .
- **User id:** in the example, it is *1* .
- **User name:** in the example, it is *Daniel* . Username won't be stored since we can retrieve it with user *id* .

Orders are composed of items (see the internal table in Fig. 26-1). So, for each **Item**, we need to store:

- **Quantity:** in the example, it is *1* for the first item. It means that the user is buying one TV.
- **Product id:** in the example, it is *1* for the first item. It means that the user is buying the product with *id 1* .
- **Product name:** in the example, it is *TV* for the first item. Product name won't be stored since we can retrieve it based on product *id* .
- **Price:** in the example, it is *1000* for the first item. We will store this price since it is common for products to change their price. Also, storing the price in the item table will allow us to know at which price the user bought each product.
- **Id:** we will include the item *id* to trace each item.

Now that we understand how these simplified invoices work, let's create the proper models and migrations.

Order migration

Let's create the order migration. In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan make:migration create_orders_table
```

The previous command creates a new migration inside the *database/migrations* folder. Open the generated file (something like *2022_02_12_152713_create_orders_table.php*) and replace the *up* and *down* methods with the following.

Modify Bold Code

```
<?php
...
return new class extends Migration
{
...
    public function up()
    {
        Schema::create('orders', function (Blueprint $table) {
                $table->id();
```

```
                $table->integer('total');
                $table->unsignedBigInteger('user_id');
                $table->foreign('user_id')->references('id')->on('users');
                $table->timestamps();
        });
    }
...
    public function down()
    {
        Schema::dropIfExists('orders');
    }
}
```

The *up* method will create a new database table called *orders* . The *orders* table will contain *id* , *total* , *user_id* , and the corresponding timestamps. Note that we create a foreign key for the *user_id* column, which references the *id* column of the *users* table. More information about foreign key constraints can be found at: https://laravel.com/docs/9.x/migrations#foreign-key-constraints.

The *down* method removes the *orders* table.

Item migration

Let's create the item migration. In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
php artisan make:migration create_items_table
```

The previous command creates a new migration inside the *database/migrations* folder. Open the generated file (something like *2022_02_12_152850_create_items_table.php*) and replace the *up* and *down* methods with the following.

Modify Bold Code

```
<?php
...
return new class extends Migration
{
...
    public function up()
    {
        Schema::create('items', function (Blueprint $table) {
                $table->id();
                $table->integer('quantity');
                $table->integer('price');
                $table->unsignedBigInteger('order_id');
                $table->foreign('order_id')->references('id')->on('orders');
                $table->unsignedBigInteger('product_id');
                $table->foreign('product_id')->references('id')->on('products');
                $table->timestamps();
        });
    }
...
    public function down()
    {
        Schema::dropIfExists('items');
    }
}
```

The *up* method will create a new database table called *items* . The *items* table will contain *id* , *quantity* , *price* , *order_id* , *user_id* , and the corresponding timestamps. Note that we create a foreign key for the *order_id* column, which references the *id* column of the *orders* table. And a foreign key for the *product_id* column, which references the *id* column of the *products* table.

The *down* method removes the *items* table.

Let's execute the previous two migrations. Go to the project directory, and in the Terminal, execute the following command (see Fig. 26-2).

Execute in Terminal

```
php artisan migrate
```

Figure 26-2. Execution of orders and items tables migrations.

Order model

In *app/Models* , create a new file *Order.php* and fill it with the following code.

Add Entire Code

```
<?php

namespace App \ Models;

use Illuminate \ Database \ Eloquent \ Model;
use App \ Models \ User;
use App \ Models \ Item;

class Order extends Model
{
	/**
	 * ORDER ATTRIBUTES
	 * $this->attributes['id'] - int - contains the order primary key (id)
	 * $this->attributes['total'] - string - contains the order name
	 * $this->attributes['user_id'] - int - contains the referenced user id
	 * $this->attributes['created_at'] - timestamp - contains the order creation date
	 * $this->attributes['updated_at'] - timestamp - contains the order update date
	 * $this->user - User - contains the associated User
	 * $this->items - Item[] - contains the associated items
	 */

	public static function validate($request)
	{
		$request->validate([
				"total" => "required|numeric",
				"user_id" => "required|exists:users,id",
		]);
	}

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

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

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

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

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

	public function setUserId($userId)
	{
		$this->attributes['user_id'] = $userId;
	}

	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;
    }

    public function user()
    {
        return $this->belongsTo(User::class);
    }

    public function getUser()
    {
        return $this->user;
    }

    public function setUser($user)
    {
        $this->user = $user;
    }

    public function items()
    {
        return $this->hasMany(Item::class);
    }

    public function getItems()
    {
        return $this->items;
    }

    public function setItems($items)
    {
        $this->items = $items;
    }
}
```

The *Order* model contains a block of comments with its attributes, getters and setters, and a validation. However, we apply four new things.

- We use an additional Laravel validation called *exists* . It verifies that the *user_id* corresponds to an existing *user id* .
- We create a method called *user* . This method represents an **Eloquent relationship** (https://laravel.com/docs/9.x/eloquent-relationships). Defining relationships as methods provides powerful method chaining and querying capabilities that we will use later. In this case, we use the *belongsTo* method, which indicates that the *Order* belongs to a specific *User* . Eloquent determines the foreign key name by examining the name of the relationship method and suffixing the method name with *_id* . So, in this case, Eloquent assumes that the *Order* model has a *user_id* column.
- We create a method called *items* . Again, this is an Eloquent relationship. We use the *hasMany* method, which indicates that the *Order* is parent to one or more child *Item* models (this is called One-To-Many relationship). Eloquent will automatically determine the proper foreign key column for the *Item* model. By convention, Eloquent will take the "snake case" name of the parent model and suffix it with *_id* . So, in this example, Eloquent will assume the foreign key column on the *Item* model is *order_id* .
- We create the getters and setters for the Eloquent relationships. However, we are not accessing these elements with the *$this->attributes* form. It is because, once the relationship is defined, Eloquent creates dynamic attributes based on those relationships. So, to get the *user* of a specific *Order* , we will need to use the dynamic attribute *$this->user* inside the *Order* class.

Once both relationship methods have been defined, we can access the related collections by using the *getUser()* method or *getItems()* method (Eloquent will provide dynamic attributes based on those relationships). We will see an example later.

Item model

In *app/Models* , create a new file *Item.php* and fill it with the following code.

Add Entire Code

```php
<?php

namespace App \ Models;

use Illuminate \ Database \ Eloquent \ Model;
use App \ Models \ Order;
use App \ Models \ Product;

class Item extends Model
{
	/**
	 * ITEM ATTRIBUTES
	 * $this->attributes['id'] - int - contains the item primary key (id)
	 * $this->attributes['quantity'] - int - contains the item quantity
	 * $this->attributes['price'] - int - contains the item price
	 * $this->attributes['order_id'] - int - contains the referenced order id
	 * $this->attributes['product_id'] - int - contains the referenced product id
	 * $this->attributes['created_at'] - timestamp - contains the item creation date
	 * $this->attributes['updated_at'] - timestamp - contains the item update date
	 * $this->order - Order - contains the associated Order
	 * $this->product - Product - contains the associated Product
	 */

	public static function validate($request)
	{
		$request->validate([
				"price" => "required|numeric|gt:0",
				"quantity" => "required|numeric|gt:0",
				"product_id" => "required|exists:products,id",
				"order_id" => "required|exists:orders,id",
		]);
	}

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

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

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

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

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

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

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

	public function setOrderId($orderId)
	{
		$this->attributes['order_id'] = $orderId;
	}

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

```
    }

    public function setProductId($productId)
    {
        $this->attributes['product_id'] = $productId;
    }

    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;
    }

    public function order()
    {
        return $this->belongsTo(Order::class);
    }

    public function getOrder()
    {
        return $this->order;
    }

    public function setOrder($order)
    {
        $this->order = $order;
    }

    public function product()
    {
        return $this->belongsTo(Product::class);
    }

    public function getProduct()
    {
        return $this->product;
    }

    public function setProduct($product)
    {
        $this->product = $product;
    }
}
```

The *Item* model contains a block of comments with its attributes, getters and setters, and a validation. Like the previous one, we put some validations related to the *Item* foreign keys and two Eloquent relationships to the *Order* and *Product* models.

Modifying User model

In *app/Models/User.php* , make the following changes in **bold**.

Modify Bold Code

```
<?php
...
use Laravel \ Sanctum \ HasApiTokens;
use App \ Models \ Order;

class User extends Authenticatable
{
    /**
```

```
     * USER ATTRIBUTES
     ...
     * $this->attributes['balance'] - int - contains the user balance
     * $this->attributes['created_at'] - timestamp - contains the user creation date
     * $this->attributes['updated_at'] - timestamp - contains the user update date
     * $this->orders - Order[] - contains the associated orders
     */
     ...

    public function orders()
    {
        return $this->hasMany(Order::class);
    }

    public function getOrders()
    {
        return $this->orders;
    }

    public function setOrders($orders)
    {
        $this->orders = $orders;
    }
}
```

We added the *orders* attribute. Now, we can access from a *User* model to its corresponding *orders* .

Modifying Product model

In *app/Models/Product.php* , make the following changes in **bold**.

Modify Bold Code

```
<?php
...
use Illuminate \ Database \ Eloquent \ Model;
use App \ Models \ Item;

class Product extends Model
{
    /**
     * PRODUCT ATTRIBUTES
     …
     * $this->attributes['created_at'] - timestamp - contains the product creation date
     * $this->attributes['updated_at'] - timestamp - contains the product update date
     * $this->items - Item[] - contains the associated items
    */
     …

    public function items()
    {
        return $this->hasMany(Item::class);
    }

    public function getItems()
    {
        return $this->items;
    }

    public function setItems($items)
    {
        $this->items = $items;
    }
}
```

We added the *items* attribute. Now, we can access from a *Product* model to its corresponding *items* .

Analyzing class diagram

Fig. 26-3 shows our initial class diagram. We have implemented all the elements in that diagram.

- We implemented the migrations for all the classes.
- We implemented all the classes' attributes.
- We implemented all the classes' associations.
- We implemented all the classes' getters and setters.
- We didn't need to implement CRUD methods since we inherited them from the Eloquent model class.

Figure 26-3. Original Online Store class diagram.

Now, we are ready to implement the purchase system.

Chapter 27 – Product Purchase

Let's make some changes to implement the product purchase.

Modifying web.php

In *routes/web.php* , make the following changes in **bold**.

Modify Bold Code

```
…
Route::post('/cart/add/{id}', 'App \ Http \ Controllers \ CartController@add')->name("cart.add");


Route::middleware('auth')->group(function () {
   Route::get('/cart/purchase', 'App \ Http \ Controllers \ CartController@purchase')->name("cart.purchase");
});


Route::middleware('admin')->group(function () {
…
```

We create the ("cart/purchase") route. This route will be only available for authenticated users.

Modifying cart/index view

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

Modify Bold Code

```
...
      <div class="row">
         <div class="text-end">
            <a class="btn btn-outline-secondary mb-2"><b>Total to pay:</b> ${{ $viewData["total"] }}</a>
            @if (count($viewData["products"]) > 0)
            <a href="{{ route('cart.purchase') }}" class="btn bg-primary text-white mb-2">Purchase</a>
            <a href="{{ route('cart.delete') }}">
               <button class="btn btn-danger mb-2">
                        Remove all products from Cart
               </button>
            </a>
            @endif
         </div>
      </div>
   </div>
</div>
@endsection
```

We modified the *cart.index* view. We only show the *Purchase* link and the "Remove all products from Cart" button if the user has products in session. And we link the *Purchase* link with the *cart.purchase* route.

Cart purchase view

In *resources/views/cart* , create a new file *purchase.blade.php* and fill it with the following code.

Add Entire Code

```
@extends('layouts.app')
@section('title', $viewData["title"])
@section('subtitle', $viewData["subtitle"])
@section('content')
<div class="card">
   <div class="card-header">
      Purchase Completed
   </div>
   <div class="card-body">
      <div class="alert alert-success" role="alert">
         Congratulations, purchase completed. Order number is <b>#{{ $viewData["order"]->getId() }}</b>
      </div>
   </div>
</div>
@endsection
```

It is a simple view that displays a message and shows the order id. This view will be displayed when the user completes the purchase.

Modifying ProductController

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

Modify Bold Code

```
<?php
…
use App \ Models \ Product;
use App \ Models \ Order;
use App \ Models \ Item;
use Illuminate \ Http \ Request;
use Illuminate \ Support \ Facades \ Auth;

class CartController extends Controller
{
	...

	public function purchase(Request $request)
	{
		$productsInSession = $request->session()->get("products");
		if ($productsInSession) {
			$userId = Auth::user()->getId();
			$order = new Order();
			$order->setUserId($userId);
			$order->setTotal(0);
			$order->save();

			$total = 0;
			$productsInCart = Product::findMany(array_keys($productsInSession));
			foreach ($productsInCart as $product) {
				$quantity = $productsInSession[$product->getId()];
				$item = new Item();
				$item->setQuantity($quantity);
				$item->setPrice($product->getPrice());
				$item->setProductId($product->getId());
				$item->setOrderId($order->getId());
				$item->save();
				$total = $total + ($product->getPrice()*$quantity);
			}
			$order->setTotal($total);
			$order->save();

			$newBalance = Auth::user()->getBalance() - $total;
			Auth::user()->setBalance($newBalance);
			Auth::user()->save();

			$request->session()->forget('products');

			$viewData = [];
			$viewData["title"] = "Purchase - Online Store";
			$viewData["subtitle"] =  "Purchase Status";
			$viewData["order"] =  $order;
			return view('cart.purchase')->with("viewData", $viewData);
		} else {
			return redirect()->route('cart.index');
		}
	}
}
```

Let’s analyze the previous method by parts.

Analyze Code

```
	public function purchase(Request $request)
	{
		$productsInSession = $request->session()->get("products");
		if ($productsInSession) {
			$userId = Auth::user()->getId();
			$order = new Order();
			$order->setUserId($userId);
			$order->setTotal(0);
			$order->save();
			…
		} else {
			return redirect()->route('cart.index');
		}
	}
```

We define the *purchase* method that is the most complex in this book. In the beginning, we check if the user has products in session. If there are no products in session, we redirect the user to the *cart.index* route. If there are products, we create an *Order* with the logged *user id* and a purchase total of *0* (we will update this value later). We create this *Order* because we need to access the *Order id* to create items.

Analyze Code

```
$total = 0;
$productsInCart = Product::findMany(array_keys($productsInSession));
foreach ($productsInCart as $product) {
	$quantity = $productsInSession[$product->getId()];
	$item = new Item();
	$item->setQuantity($quantity);
	$item->setPrice($product->getPrice());
	$item->setProductId($product->getId());
	$item->setOrderId($order->getId());
	$item->save();
	$total = $total + ($product->getPrice()*$quantity);
}
```

Then, we iterate through the *productsInCart* . For each product in *productsInCart* , we create a new *Item* , set the corresponding *quantity* (based on the values stored in session), *price* , *product id* , and *order id* . We then *save* the item and update the *total* value.

Analyze Code

```
$order->setTotal($total);
$order->save();

$newBalance = Auth::user()->getBalance() - $total;
Auth::user()->setBalance($newBalance);
Auth::user()->save();

$request->session()->forget('products');

$viewData = [];
$viewData["title"] = "Purchase - Online Store";
$viewData["subtitle"] =  "Purchase Status";
$viewData["order"] =  $order;
return view('cart.purchase')->with("viewData", $viewData);
```

We update the *order total* and *save* it. Then, we calculate and set the new user's *balance* . We then remove the products in session, and show the *cart.purchase* view with the *order* .

Note: we have not verified if the user has enough money to purchase. Try to include that validation in the previous code. You can use the discussion zone of the book repository to show us your solution.

Running the app

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

Execute in Terminal

```
php artisan serve
```

Now, log in to the application, go to the ("/products") route, and add some products to the cart (see Fig. 27-1). Click the *Purchase* button, and the application will show a confirmation message (see Fig. 27-2). Next, you can open phpMyAdmin and check the *orders* table (see Fig 27-3), and *items* table (see Fig. 27-4) contains the new data.

Figure 27-1. Shopping cart with some products.

Figure 27-2. Purchase completed message.

id	total	user_id	created_at	updated_at
1	1059	1	2022-02-12 15:58:56	2022-02-12 15:58:56

Figure 27-3. New order row included in the database.

id	quantity	price	order_id	product_id	created_at	updated_at
1	1	999	1	2	2022-02-12 15:58:56	2022-02-12 15:58:56
2	2	30	1	3	2022-02-12 15:58:56	2022-02-12 15:58:56

Figure 27-4. New item rows included in the database.

Chapter 28 – Orders Page

Let's finish our Online Store application. We will create a page where users can list their orders.

Modifying web.php

In *routes/web.php* , make the following changes in **bold**.

Modify Bold Code

```
…
Route::middleware('auth')->group(function () {
   Route::get('/cart/purchase', 'App \ Http \ Controllers \ CartController@purchase')->name("cart.purchase");
   Route::get('/my-account/orders', 'App \ Http \ Controllers \ MyAccountController@orders')->name("myaccount.orders");
});
…
```

We create the ("my-account/orders") route. This route will be only available for authenticated users.

Modifying app.blade.php

In *resources/views/layouts/app.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
...
            <a class="nav-link active" href="{{ route('register') }}">Register</a>
            @else
            <a class="nav-link active" href="{{ route('myaccount.orders') }}">My Orders</a>
...
```

We added a new link to My Orders page.

MyAccountController

In *app/Http/Controllers* , create a new file *MyAccountController.php* and fill it with the following code.

Add Entire Code

```
<?php


namespace App \ Http \ Controllers;


use Illuminate \ Http \ Request;
use App \ Models \ Order;
use Illuminate \ Support \ Facades \ Auth;


class MyAccountController extends Controller
{
    public function orders()
    {
        $viewData = [];
        $viewData["title"] = "My Orders - Online Store";
        $viewData["subtitle"] =  "My Orders";
        $viewData["orders"] = Order::where('user_id', Auth::user()->getId())->get();
        return view('myaccount.orders')->with("viewData", $viewData);
    }
}
```

The *orders* method collects the orders based on the authenticated user id and displays the *myaccount.orders* view.

MyAccount orders view

In *resources/views/* , create a subfolder *myaccount* . Then, in *resources/views/myaccount* , create a new file *orders.blade.php* and fill it with the following code.

Add Entire Code

```
@extends('layouts.app')
@section('title', $viewData["title"])
@section('subtitle', $viewData["subtitle"])
@section('content')
@forelse ($viewData["orders"] as $order)
<div class="card mb-4">
  <div class="card-header">
    Order #{{ $order->getId() }}
  </div>
  <div class="card-body">
    <b>Date:</b> {{ $order->getCreatedAt() }}<br />
```

```
    <b>Total:</b> ${{ $order->getTotal() }}<br />
    <table class="table table-bordered table-striped text-center mt-3">
      <thead>
        <tr>
          <th scope="col">Item ID</th>
          <th scope="col">Product Name</th>
          <th scope="col">Price</th>
          <th scope="col">Quantity</th>
        </tr>
      </thead>
      <tbody>
        @foreach ($order->getItems() as $item)
        <tr>
          <td>{{ $item->getId() }}</td>
          <td>
                <a class="link-success" href="{{ route('product.show', ['id'=> $item->getProduct()->getId()]) }}">
                  {{ $item->getProduct()->getName() }}
                </a>
          </td>
          <td>${{ $item->getPrice() }}</td>
          <td>{{ $item->getQuantity() }}</td>
        </tr>
        @endforeach
      </tbody>
    </table>
  </div>
@empty
<div class="alert alert-danger" role="alert">
  Seems to be that you have not purchased anything in our store =(.
</div>
@endforelse
@endsection
```

We display the user orders with their respective information. For this view, we don't use the *@foreach* Blade directive. Instead of that, we use the *@forelse* directive. *@forelse* allows defining a *@empty* directive in which we display a message in case the user doesn't have any orders.

If the user has no orders, we use the *@empty* directive to show a message saying, "Seems to be that you have not purchased anything in our store". Otherwise, we show each order's *id* , creation date , and *total* (using the respective getters). Then, we have a second *@foreach* . This time, we iterate through the items of each order (using the *getItems()* method). We show each item's *id* , *price* , and *quantity* . Finally, we access the product properties of each item (using the *getProduct()* method) and display the product *name* and link to the respective product page.

Note the way we are easily accessing the order relationships (i.e., *$item->getProduct()->getName()*). That's the power of Eloquent relationships. Now, we can navigate through the entire class diagram just with some getters.

Power of the Eloquent relationships

Let's check another example to understand the power of the Eloquent relationships. Previously, we purchased an iPhone product. Then, open Laravel Tinker in our Terminal (*php artisan tinker*) and execute the following command.

Analyze Code

```
Product::where('name', 'iPhone')->first()->getItems()[0]->getOrder()->getUser()->getName();
```

1. We look for the *first product* , which contains "iPhone" as its name.
2. We obtain the *items* in which that product was purchased.
3. We access the *first item* .
4. We obtain the *order* of the previous *item* .
5. We obtain the *user* of the previous *order* .
6. We access the *name* of the previous *user* .

In summary, we are obtaining the user's *name* who bought the first "iPhone" in our store. Fig. 28-1 shows this name. It was Daniel. Can you believe it?

```
PS C:\xampp\htdocs\onlineStore> php artisan tinker
Psy Shell v0.11.1 (PHP 8.0.11 — cli) by Justin Hileman
>>> Product::where('name', 'iPhone')->first()->getItems()[0]->getOrder()->getUser()->getName();
[!] Aliasing 'Product' to 'App\Models\Product' for this Tinker session.
=> "Daniel Correa"
>>> exit;
Exit:  Goodbye
```

Figure 28-1. Use of Laravel tinker.

Running the app

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

Execute in Terminal

```
php artisan serve
```

Now, log in to the application, go to the ("/my-account/orders") route, and you will see your orders (see Fig. 28-2). If you have no orders (see Fig. 28-3), make a purchase, and return to the orders page.

Figure 28-2. Online Store – Orders page.

Figure 28-3. Online Store – Orders page with no orders.

Lazy loading and eager loading

The use of Eloquent relationships has a silent cost. The related models are "**lazy loaded**" when we access Eloquent relationships as attributes. It means the relationship data is not loaded until you first access the attribute. It presents an important issue. For example, suppose we have ten orders on the orders page. In this case, Laravel will make one database query to extract the ten orders (that's not an issue). However, in our *orders.blade.php* we created a *@forelse* which iterates over each order and extracts the corresponding items. Since Eloquent uses lazy loading, for each order, Laravel will make an extra database query to collect the information of the items related to each order (another ten queries). And, since we are extracting the product name of each item, it will execute another set of queries (minimum another ten queries). So, in this scenario, we will need to execute at least 21 queries to show the

orders page (that's crazy). Lazy loading affects our application performance. Fortunately, Eloquent can reduce the number of queries and "**eager load**" relationships using the *with* method (we will see it in action later).

Debugging our application queries

"Laravel Debugbar" library allows us to debug our Laravel application (https://github.com/barryvdh/laravel-debugbar). We won't teach you how to install it (you can read it from the previous link, it is only to execute a composer command). But we will show the difference when you use eager loading over lazy loading.

Fig. 28-4 shows the debug of the orders page (with our current lazy loading scenario). We had three orders for this user, and in this case, the debug bar shows that nine queries were executed.

Figure 28-4. Online Store – Orders page with debug bar and lazy loading.

Using eager loading

Let's improve our application code and use eager loading for the orders page. In *app/Http/Controllers/MyAccountController.php* , make the following changes in **bold**.

Modify Bold Code

```
…
    public function orders()
    {
        $viewData = [];
        $viewData["title"] = "My Orders - Online Store";
        $viewData["subtitle"] =  "My Orders";
        $viewData["orders"] = Order::with(['items.product'])->where('user_id', Auth::user()->getId())->get();
        ~~$viewData["orders"] = Order::where('user_id', Auth::user()->getId())->get();~~
        return view('myaccount.orders')->with("viewData", $viewData);
    }
…
```

The previous code specifies which relationships should be eager loaded using the *with* method. In this case, we load the *orders* with their corresponding *items* and the *products* of the related *items* . Let's reload the page and check the new results (see Fig. 28-5).

Figure 28-5. Online Store – Orders page with debug bar and eager loading.

We reduced the number of queries to four. But that's not the best part. Now, if we have ten or 50 orders, we will always execute just four queries because we loaded the relationships in advance (eager loading).

TIP: Always try to install a debug bar or mechanism to check the executed queries of each application section. Remember to use eager loading to reduce the quantity of executed queries. If that doesn't work, try to use a cache mechanism or another solution.

We have finished our application. Congratulations! We hope you better understand our architecture diagram and its elements (see Fig. 28-6).

Figure 28-6. Online Store software architecture.

Let's have a quick final analysis.

- We integrated MVC into our software architecture.
- We have proper consistency between the architecture diagram, class diagram, and the developed code. For example, the architecture diagram shows four models corresponding to the class diagram classes. Those models are coded adequately into the *app/Http/model* Laravel folder.
- We defined a set of rules across the book chapters. Note that some of those rules are consistent with the architecture diagram. For example, all requests pass for the *web.php* file, connecting to controllers. Hopefully, a developer who reads the documentation, or analyzes the architecture diagram, will understand that it is not allowed to invoke models from the *web.php* file or display a view.

What an exciting journey! We have completed our Online Store with an MVC architecture. Let's have a final tip and discussion, before deploying our application to the cloud.

TIP: A final quote from the (*2019 - Thomas, D., & Hunt, A. - The Pragmatic Programmer: your journey to mastery*) book. "Critically Analyze What You Read and Hear". That video you saw on YouTube, that post you read in StackOverFlow, this paragraph that you are reading in this book. Does it make sense? Is it a good strategy? Does it apply to your context? Is it true?

Quick discussion: Have you seen the "Harry Potter and the Half-Blood Prince" film? In the film, Harry Potter was enrolled in a potions class. The class had an "Advanced Potion-Making" textbook that contained a variety of recipes for various potions. Harry had not bought his own textbook. So, the potions professor loaned one of the older books (left behind by previous students) to Harry. This book belonged to Severus Snape, whose nickname was the "Half-Blood Prince". Snape improved many of the potions by means of including procedures and scribbling notes in the margins. Harry tried Snape's methods and achieved the best results in the class. We all should be like the Half-Blood Prince, trying to critically analyze everything and make our own judgements. Book authors make mistakes, experts make mistakes, we all make mistakes.

There is still room for improvement.

Chapter 29 – Deploying to the Cloud – Clever-Cloud – MySQL Database

Now, we will learn how to deploy our application to the cloud using a free cloud server.

Introduction to Clever-Cloud

Clever Cloud is a 'Platform as a Service' company that helps developers deploy and run their apps over the cloud. In addition, Clever Cloud offers free service trials, such as the MySQL service. We will deploy, run, and monitor our MySQL databases and tables using this service.

Creating a Clever Cloud account

Let's create a Clever Cloud account. Go to https://api.clever-cloud.com/v2/session/signup and complete the "Sign up with Email" information (see Fig. 29-1).

Figure 29-1. Clever Cloud Sign up page.

Validate your email. After that, access https://console.clever-cloud.com/, click *Personal space* , *Create…* , and click *an add-on* . After that, select *MySQL* , choose the *DEV plan* option (the first one), which costs 0.00€, and click *next* . Then, put *onlineStore* as the add-one name, leave the region as it appears, and click *next* (see Fig. 29-2). We have created our MySQL cloud service.

Figure 29-2. Clever Cloud MySQL add-on creation.

After the MySQL creation, we will see a dashboard (see Fig. 29-3), which contains our "Database Credentials" and a link to "phpMyAdmin". We will use this information next.

Figure 29-3. Clever Cloud MySQL add-on creation.

Note: if for any reason, Clever Cloud disables the *DEV free plan* , just create a new topic in the discussion zone of the book repo, and we will make a tutorial with an alternative platform.

Cloning our application

Let's make a copy of our application code. Copy all the content of your *onlineStore* folder to a new folder *onlineStoreCloud* (in a location of your choice).

Modifying the .env file

To execute the migrations over our cloud database, we must modify the *.env* file (located at the cloud project root folder).

Go to your *onlineStoreCloud* folder, and in the *.env* file, make the following changes in **bold**. Replace the *DATABASE_CREDENTIALS_HOST* , *DATABASE_CREDENTIALS_DATABASE_NAME* , *DATABASE_CREDENTIALS_USER* , and *DATABASE_CREDENTIALS_PASSWORD* with your values from the "Clever Cloud addon dashboard" presented previously.

Modify Bold Code with Your Values

```
...
DB_CONNECTION=mysql
DB_HOST=DATABASE_CREDENTIALS_HOST
DB_PORT=3306
DB_DATABASE=DATABASE_CREDENTIALS_DATABASE_NAME
DB_USERNAME=DATABASE_CREDENTIALS_USER
DB_PASSWORD=DATABASE_CREDENTIALS_PASSWORD
...
```

You will get something like the following.

Analyze Code

```
...
DB_CONNECTION=mysql
DB_HOST=example-mysql.services.clever-cloud.com
DB_PORT=3306
DB_DATABASE=bhexamplejlml1ycqdln
DB_USERNAME=uqfksexampleme3dv
DB_PASSWORD=z5jCE59QexamplemXv74DF...
```

Executing the migrations

To run the migrations, go to the **cloud project** directory, and in the Terminal, execute the following command (see Fig. 29-4).

Execute in Terminal

```
php artisan migrate
```

```
PS C:\xampp\htdocs\onlineStore> php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table (694.42ms)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table (716.37ms)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated:  2019_08_19_000000_create_failed_jobs_table (765.13ms)
Migrating: 2019_12_14_000001_create_personal_access_tokens_table
Migrated:  2019_12_14_000001_create_personal_access_tokens_table (1,131.34ms)
Migrating: 2022_02_11_153916_create_products_table
Migrated:  2022_02_11_153916_create_products_table (351.93ms)
Migrating: 2022_02_12_140820_alter_users_table
Migrated:  2022_02_12_140820_alter_users_table (345.62ms)
Migrating: 2022_02_12_152713_create_orders_table
Migrated:  2022_02_12_152713_create_orders_table (739.33ms)
Migrating: 2022_02_12_152850_create_items_table
Migrated:  2022_02_12_152850_create_items_table (1,114.81ms)
```

Figure 29-4. Execution of Laravel migrations over the cloud database.

Besides, we must recreate the "symbolic link" from *public/storage* to *storage/app/public*. To do this, first, manually remove the *public/storage* folder, go to the cloud project directory, and in the Terminal, execute the following command.

Execute in Terminal

```
php artisan storage:link
```

Now, if you go to the Clever Cloud "phpMyAdmin" tab, you will see the tables properly created (see Fig. 29-5).

Figure 29-5. Accessing phpMyAdmin over the Clever Cloud database.

Exporting and importing our data

We currently have empty tables. Let's copy the data from our local database to our cloud database. Open "phpMyAdmin" from your local computer. Go to the *online_store* database, go to the *Export* tab, and select "export method" as *Custom*. In tables, deselect all the *Structure* options, and in *Data*, select the next ones: (*items*, *orders*, *password_resets*, *personal_access_tokens*, *products*, and *users*). Finally, click *Go* (see Fig. 29-6).

Figure 29-6. Exporting the data from the local database.

You will get a SQL file. This file contains our application data. Then, go to the cloud database, go to "phpMyAdmin", go to your cloud database, click the *Import* tab, select the SQL file, deselect the *Enable foreign key checks* option, and click *Go* (see Fig. 29-7).

Figure 29-7. Importing the data to the cloud database.

Congratulations, we have our database and data in the cloud. You can check that your data was successfully uploaded by opening the *products* or *users* table (see Fig. 29-8).

Figure 29-8. Browsing the *products* table.

Chapter 30 – Deploying to the Cloud – Heroku – Laravel Application

Now, let's deploy our Laravel code over the cloud.

Creating a Heroku account

Let's create a Heroku account. First, access https://signup.heroku.com/, complete the information, and click "Create Free Account". Next, confirm your email address and set up your password.

Installing Git

Install git by following the instruction in this link: https://git-scm.com/.

Initializing a Git repository

Go to the **cloud project** directory, and in the Terminal, execute the following commands.

Execute in Terminal

```
git init
git add .
git commit -m "new laravel project"
```

The commands initialize a Git repository and commit the current state.

Installing Heroku CLI

The Heroku Command Line Interface (CLI) makes it easy to create and manage our Heroku apps directly from the Terminal. Follow the installation instructions from this link: https://devcenter.heroku.com/articles/heroku-cli.

Deploying to Heroku

To deploy our application to Heroku, you must create a *Procfile* file, which tells Heroku what setting to use to launch the web server. In the cloud project root folder, create a new file *Procfile* (without any extension), and fill it with the following code.

Add Entire Code

```
web: vendor/bin/heroku-php-apache2 public/
```

Heroku will launch an Apache web server together with PHP to serve applications from the cloud project's root directory. In this case, we specify that our root directory is *public/* .

Secure assets

Heroku requires that all assets be loaded from HTTPS. So, we must modify two code lines over our layouts.

In the cloud project directory, in *resources/views/layouts/app.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
<html lang="en">
<head>
   ...
   <link href="{{ secure_asset('/css/app.css') }}" rel="stylesheet" />
   …
```

In the cloud project directory, in *resources/views/layouts/admin.blade.php* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
<html lang="en">
<head>
   ...
   <link href="{{ secure_asset('/css/admin.css') }}" rel="stylesheet" />
   …
```

Committing changes

Since we added a new file and modified the other two, we need to include them in a new commit. So, go to the cloud project directory, and in the Terminal, execute the following commands.

Execute in Terminal

```
git add .
git commit -m "adding Procfile and secure assets"
```

Creating a new application on Heroku

To create a new Heroku application, we use the heroku create command. Go to the **cloud project** directory, and in the Terminal, execute the following command.

Execute in Terminal

```
heroku create
```

This command creates a Heroku application. Sometimes, the command will open a browser tab to prompt us to log in to the Heroku website (if that is your case, please complete the log in process). In the end, it provides a link with our new Heroku application deployed in the cloud (see Fig. 30-1).

```
PS C:\xampp\htdocs\onlineStore> heroku create
Creating app... done, ⬢ powerful-savannah-94864
https://powerful-savannah-94864.herokuapp.com/ | https://git.heroku.com/powerful-savannah-94864.git
```

Figure 30-1. Creating a Heroku application.

If you access the link (in our case, it was https://powerful-savannah-94864.herokuapp.com/), you will see a default application with a welcome message (see Fig. 30-2).

Figure 30-2. Accessing the Heroku application.

Setting a Laravel encryption key

The application's encryption key is used by Laravel to encrypt user sessions and other information. Its value will be read from the APP_KEY environment variable.

Go to the cloud project directory, and in the Terminal, execute the following command.

Execute in Terminal

```
php artisan key:generate --show
```

The previous command will print a key that you need to copy and paste at the end of the following command (execute it in the Terminal). Replace the *ARTISAN_KEY* with the value you got from the previous command.

Execute in Terminal with Your Value

heroku config:set APP_KEY=**ARTISAN_KEY**

You will execute something like the following command.

Analyze Code

```
heroku config:set APP_KEY=base64:pN+ekIge1udSJXQR6J+al8NKewCXqG85H93RMrTPn78=
```

Setting Clever Cloud credentials

Go to the cloud project directory, and in the Terminal, execute the following commands. Replace the *DATABASE_CREDENTIALS_HOST*, *DATABASE_CREDENTIALS_DATABASE_NAME*, *DATABASE_CREDENTIALS_USER*, and *DATABASE_CREDENTIALS_PASSWORD* with your own values (those collected from the "Clever Cloud addon dashboard").

Execute in Terminal with Your Values

heroku config:set DB_HOST=**DATABASE_CREDENTIALS_HOST**
heroku config:set DB_DATABASE=**DATABASE_CREDENTIALS_DATABASE_NAME**
heroku config:set DB_USERNAME=**DATABASE_CREDENTIALS_USER**
heroku config:set DB_PASSWORD=**DATABASE_CREDENTIALS_PASSWORD**

Pushing to Heroku

Next, it's time to deploy our application to Heroku. Go to the cloud project directory, and in the Terminal, execute the following command.

Execute in Terminal

```
git push heroku master
```

Now, open your application with the following command.

Execute in Terminal

```
heroku open
```

You will see the complete Online Store application running over the cloud (see Fig. 30-3). Congratulations!

Figure 30-3. Online Store running over the cloud.

Note 1: If you get a 500 error, it means you didn't configure your *APP_KEY* properly or some of your database credentials.

Note 2: Heroku uses an Ephemeral filesystem. It means that any files we store on the local disk get deleted at least once every 24 hours without doing anything. They're also deleted every time we push to Heroku. So, we won't configure our product images (they won't load). There is a solution that requires to use Amazon S3 or another cloud image storage system. However, it is out of the scope of this book. If you have a proper alternative, post it in the discussion zone of the book repository.

Chapter 31 – Continue your Laravel Journey

We have learned a lot since we started. We took a practical journey to master the design and implementation of MVC web applications with Laravel. We developed a real-world project in which we applied a lot of different Laravel functionalities. We purposely made code mistakes, but we fixed them and learned some practical tips about clean code (not only for Laravel projects but for any web MVC project).

There are missing elements we didn't address in this book. So, we will show you additional interesting links and concepts of Laravel that you can learn independently. We will also give you some book recommendations.

Other Laravel interesting features

- **Factories and seeders.** Do you remember the SQL command we used to insert our first products? You can automate this process with fakers and seeders https://laravel.com/docs/9.x/database-testing.
- **Gates and policies.** Do you remember the admin middleware that we created to restrict access to admin routes? You can have similar behavior with gates and policies https://laravel.com/docs/9.x/authorization.
- **Eloquent Resources.** Do you want to design a service-oriented architecture? Or do you have a frontend developed with React or Vue, and you need a backend with Laravel? Check Eloquent resources https://laravel.com/docs/9.x/eloquent-resources.
- **Query builder.** Do you want to implement complex SQL queries or better database performance? Check Query builder (an alternative to Laravel Eloquent) https://laravel.com/docs/9.x/queries.
- **Testing.** Do you want to test your Laravel application? Check Laravel testing https://laravel.com/docs/9.x/testing.
- **Pagination.** Do you want to paginate your products? Check database pagination https://laravel.com/docs/9.x/pagination.
- **Cache.** Do you want to cache some parts of your application? Check Laravel cache https://laravel.com/docs/9.x/cache.
- **Mail.** Do you want to send an email from your application? Check Laravel mail system https://laravel.com/docs/9.x/mail.
- **Localization.** Do you want to provide multilanguage support for your application? Check Laravel localization https://laravel.com/docs/9.x/localization.

Recommended books

Do you want to learn other frameworks and other types of software architectures? We have some recommendations.

- **Practical Nest.js: Develop clean MVC web applications (By Daniel Correa and Greg Lim)** https://www.amazon.com/dp/B09RKLFXD4 - This is a book where we design an MVC architecture with Nest.js (Node.js). It is a book like this one, we design the same Online Store application with a trending technology called Nest.js.
- **Beginning Django 4: Build Full Stack Python Web Applications (By Greg Lim and Daniel Correa)** https://www.amazon.com/gp/product/B09M2N778S - This is a book where we design an MVT architecture with Django (Python). We don't have some clean code strategies and diagrams as we did in this book, but it is a great book for beginners.
- **Beginning Vue Stack: Build and Deploy a Full Stack MongoDB, Express, Vue.js, Node.js App (By Greg Lim and Daniel Correa)** https://www.amazon.com/gp/product/B09G9V44MN - Do you want to split your application into two parts (frontend and backend)? You can use this book as a base. We design two applications: a frontend with Vue.js (which follows an MVVM architecture) and a backend with Express (Node.js). In the process, you will learn another database system (MongoDB).
- **Beginning Django API with React: Build Django 4 Web APIs with React Full Stack Applications (By Greg Lim and Daniel Correa)** https://www.amazon.com/dp/B09S5XZ6RK - We design two applications: a frontend with React and a backend with Django (Python).

Final remarks and future books

Across our books, we have developed an extensive range of web applications with different architectures (not only MVC), e.g., applications with separated frontend and backend, service-oriented architectures (SOA), and even micro-services. We are developing books like this on Express (Node.js), Django (Python), and Spring (Java), so keep checking Daniel's Twitter account (**@danielgarax**).

We would love to get your feedback. Contact us at practicalbooksco@gmail.com. Please let us know what you liked and what you didn't. That's the way all of us improve as web developers. We constantly write to book authors with questions, suggestions, critics. Both author and readers learn a lot in this process.

Finally, please try to leave an honest review on Amazon. This is vital so that other potential readers can see and use your unbiased opinion to buy the book and understand what people like and didn't like about the book. It will only take a few minutes of your time but is very valuable to us.

"Hecho en Medellín"