Practical Laravel

Develop clean MVC web applications

Daniel Correa - Paola Vallejo

Practical Laravel

Develop clean MVC web applications

Daniel Correa – Paola Vallejo

Practical Books

Practical Laravel

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

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.

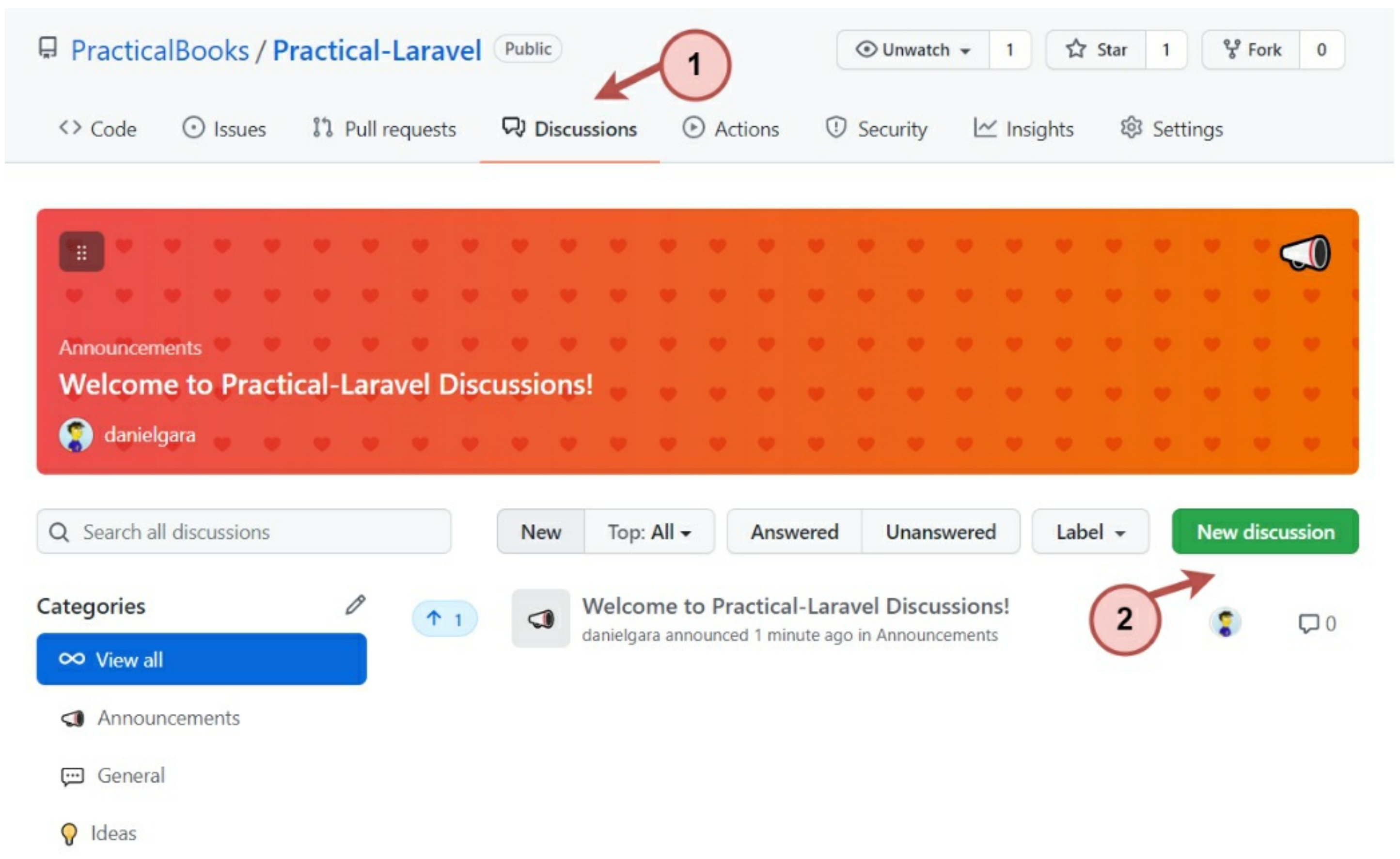


Figure P-1. Discussion zone of the GitHub repository.

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.

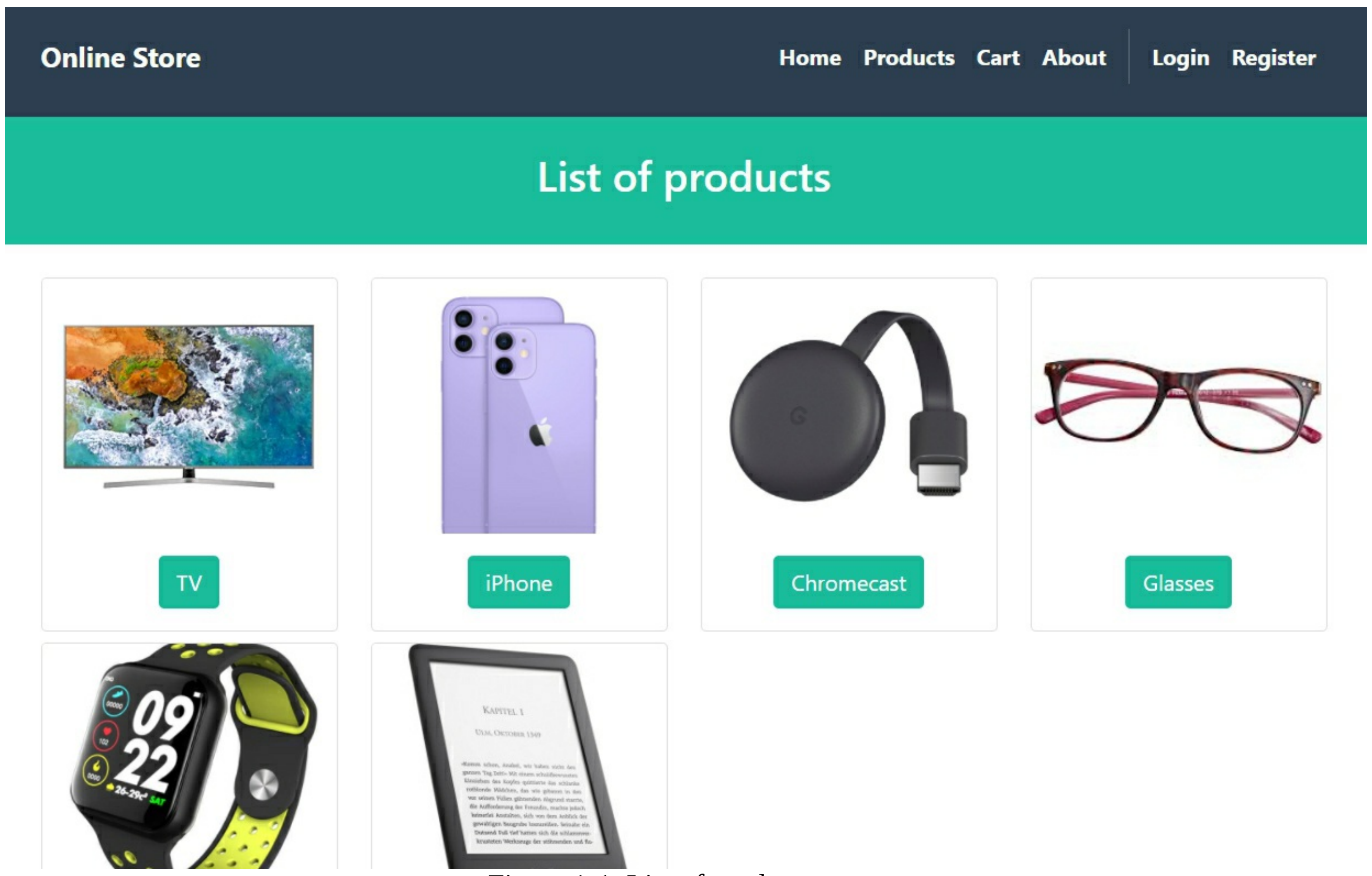
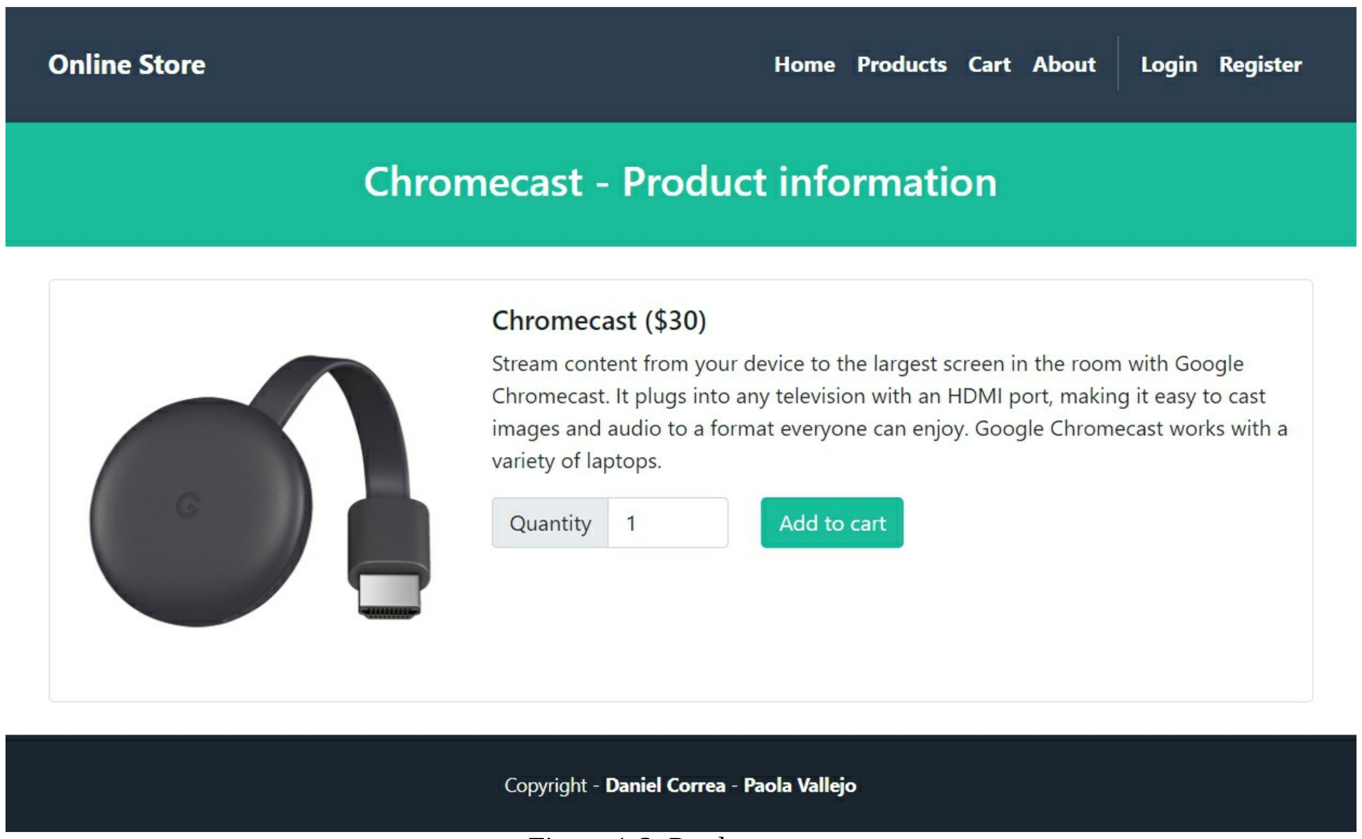


Figure 1-1. List of products page.

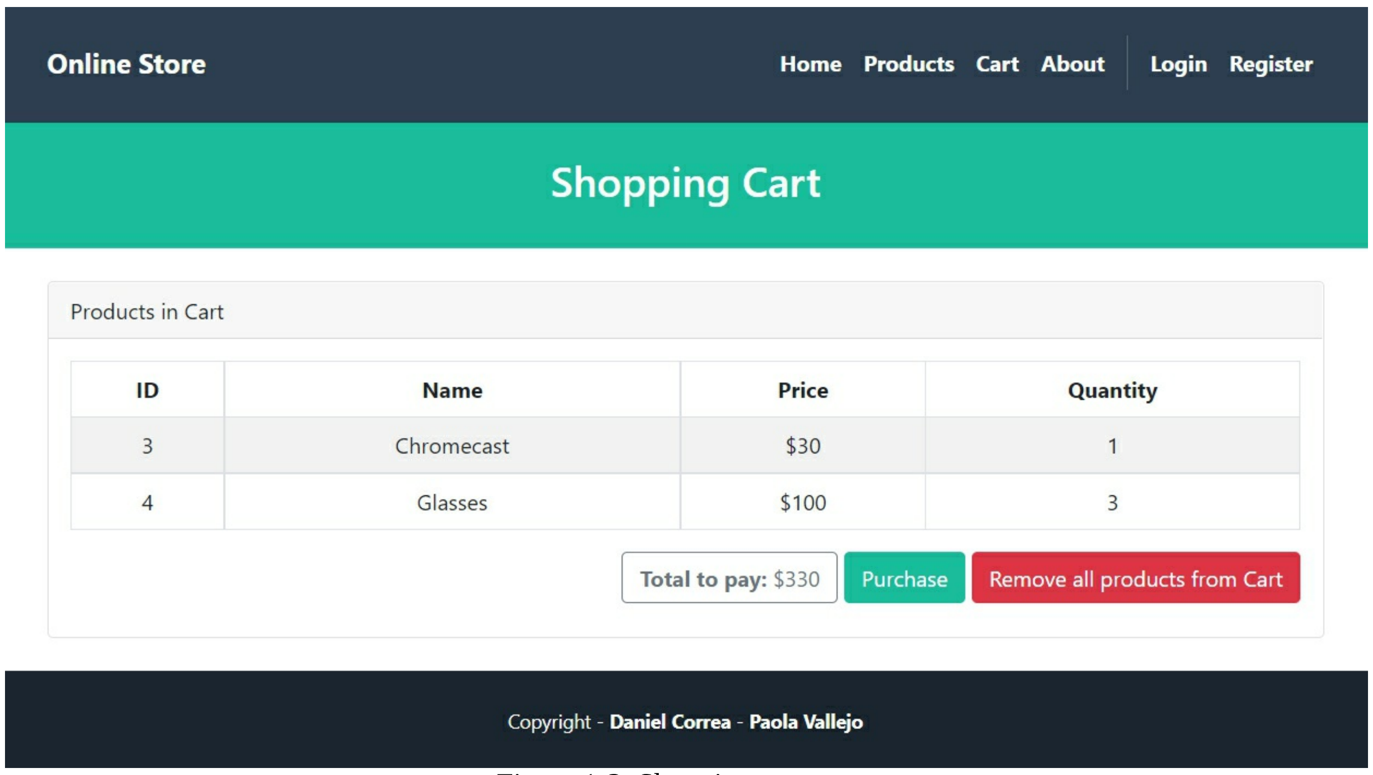


Chromecast ($30)

Stream content from your device to the largest screen in the room with Google Chromecast. It plugs into any television with an HDMI port, making it easy to cast images and audio to a format everyone can enjoy. Google Chromecast works with a variety of laptops.

Copyright - **Daniel Correa** - **Paola Vallejo**

Figure 1-2. Product page.



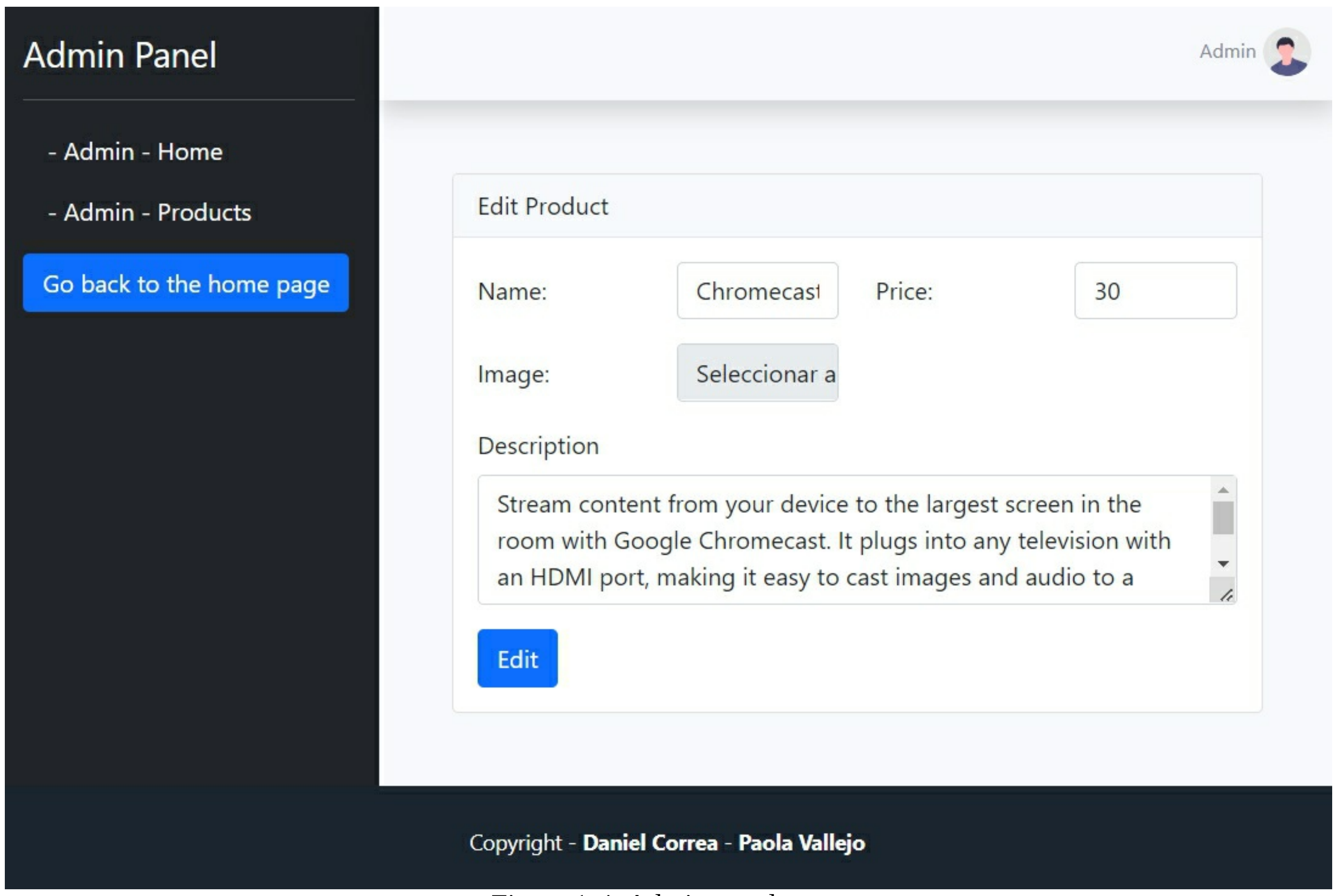
Products in Cart

ID	Name	Price	Quantity
3	Chromecast	$30	1
4	Glasses	$100	3

Total to pay: $330

Copyright - **Daniel Correa** - **Paola Vallejo**

Figure 1-3. Shopping cart page.



Stream content from your device to the largest screen in the room with Google Chromecast. It plugs into any television with an HDMI port, making it easy to cast images and audio to a

Copyright - **Daniel Correa** - **Paola Vallejo**

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.).

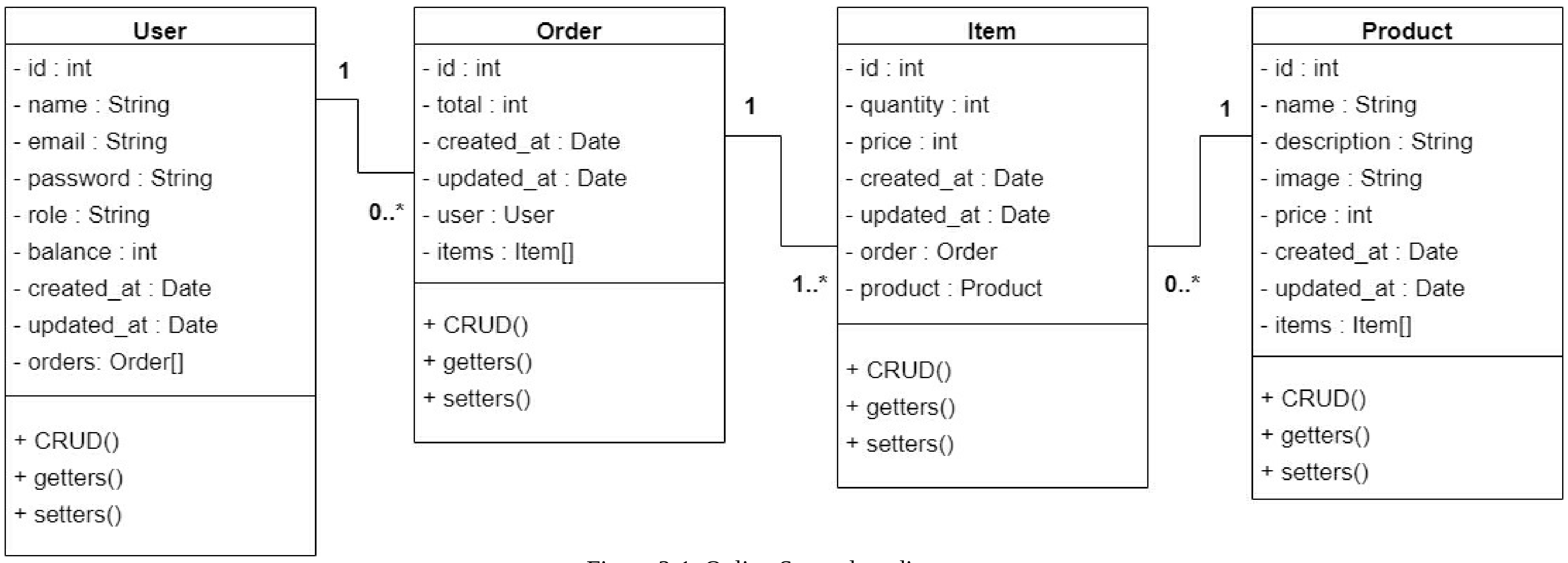


Figure 2-1. Online Store class diagram.

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

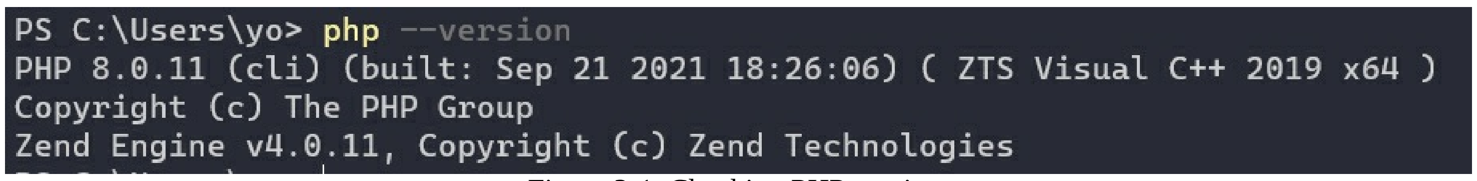
XAMPP is the most popular PHP development environment. XAMPP is a free, easy to install Apache distribution containing MySQL, PHP, and Perl. If you don't have XAMPP installed, go to https://www.apachefriends.org/download.html. Download and install it. **Be careful, you will need to download and install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

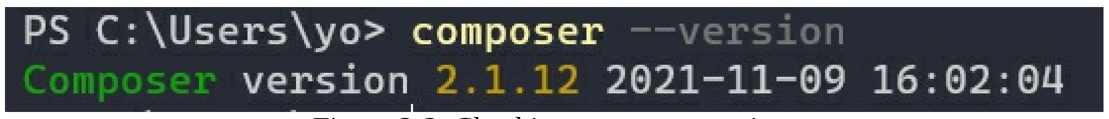
Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on, and it will manage (install/update) them for you. If you don't have Composer installed, go to https://getcomposer.org/download/. Download and install it.

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

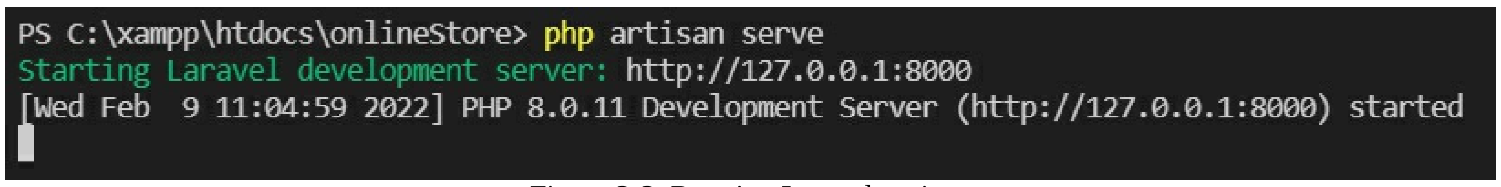
```
composer create-project laravel/laravel onlineStore "9.*" --prefer-dist
```

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
```

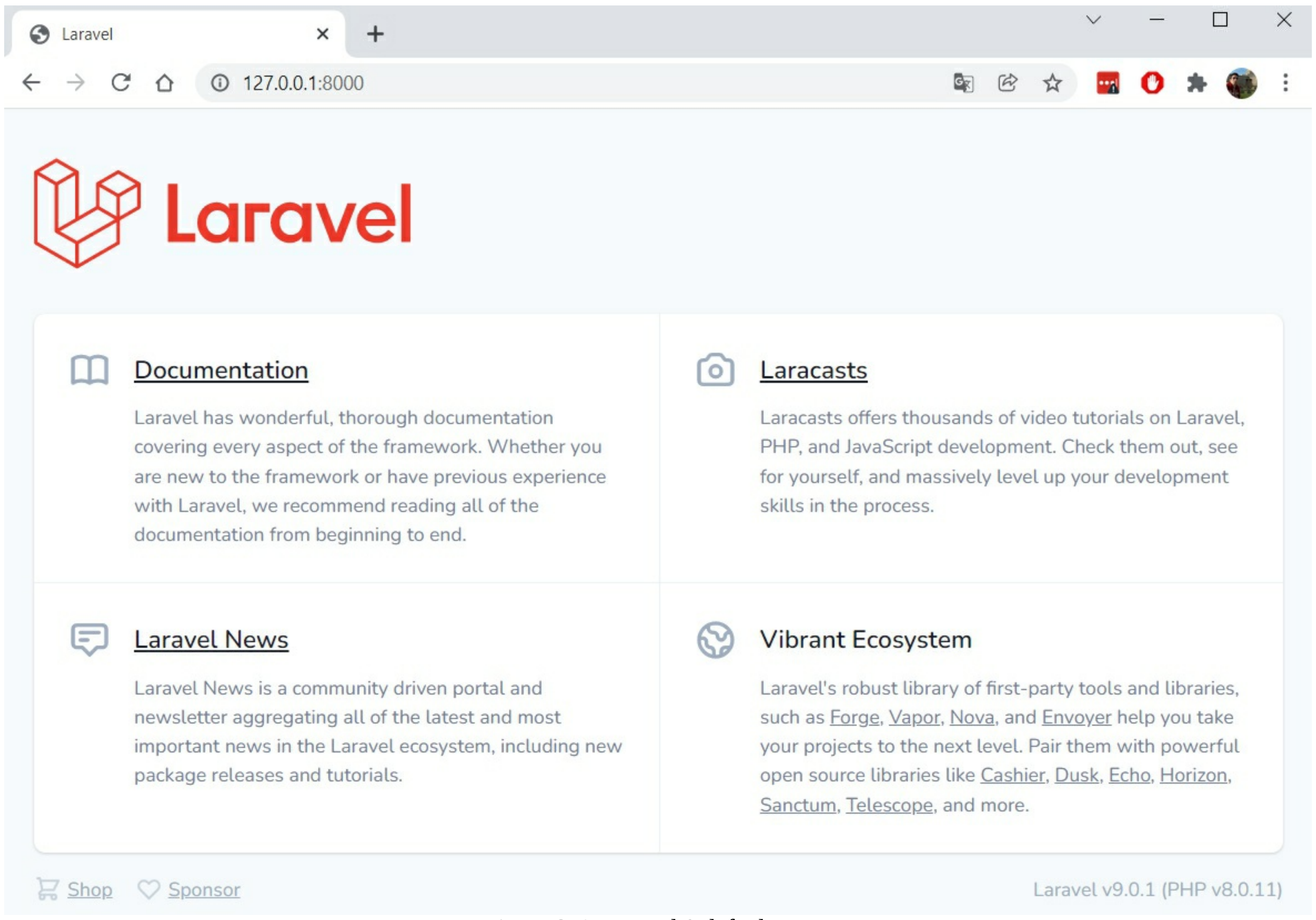
The *php artisan serve* command starts Laravel's server (see Fig. 3-3). **Artisan** is the command line interface included with Laravel. Artisan exists at the root of your application as the artisan script and provides helpful commands to assist you while you build your application. More information 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.



Documentation

Laravel has wonderful, thorough documentation covering every aspect of the framework. Whether you are new to the framework or have previous experience with Laravel, we recommend reading all of the documentation from beginning to end.

Laracasts

Laracasts offers thousands of video tutorials on Laravel, PHP, and JavaScript development. Check them out, see for yourself, and massively level up your development skills in the process.

Laravel News

Laravel News is a community driven portal and newsletter aggregating all of the latest and most important news in the Laravel ecosystem, including new package releases and tutorials.

Vibrant Ecosystem

Laravel's robust library of first-party tools and libraries, such as Forge, Vapor, Nova, and Envoyer help you take your projects to the next level. Pair them with powerful open source libraries like Cashier, Dusk, Echo, Horizon, Sanctum, Telescope, and more.

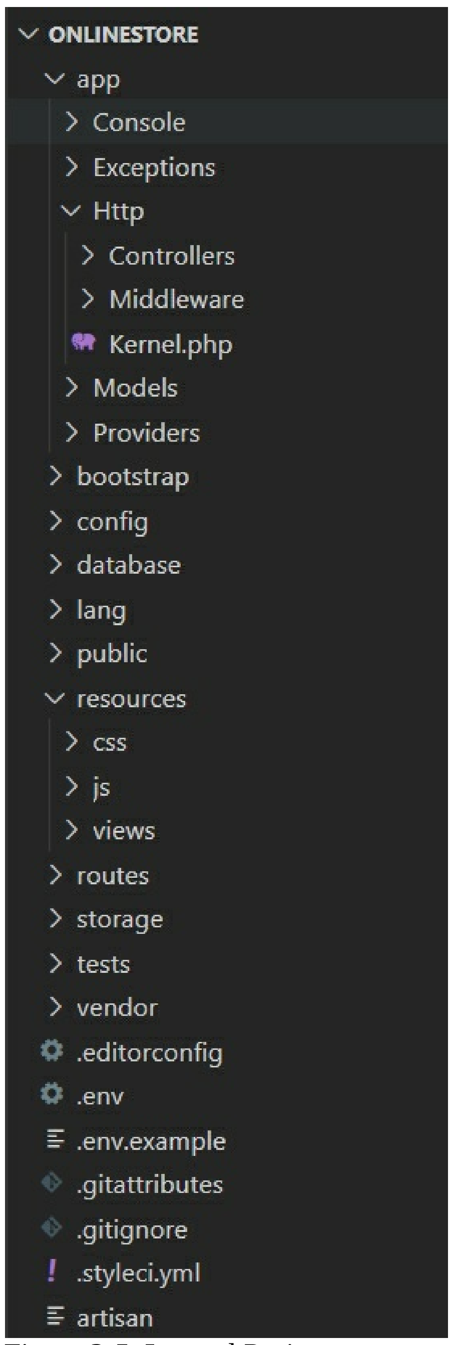
Shop Sponsor Laravel v9.0.1 (PHP v8.0.11)

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.



```
ONLINESTORE
  app
    Console
    Exceptions
    Http
      Controllers
      Middleware
      Kernel.php
    Models
    Providers
  bootstrap
  config
  database
  lang
  public
  resources
    css
    js
    views
  routes
  storage
  tests
  vendor
  .editorconfig
  .env
  .env.example
  .gitattributes
  .gitignore
  .styleci.yml
  artisan
```

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.

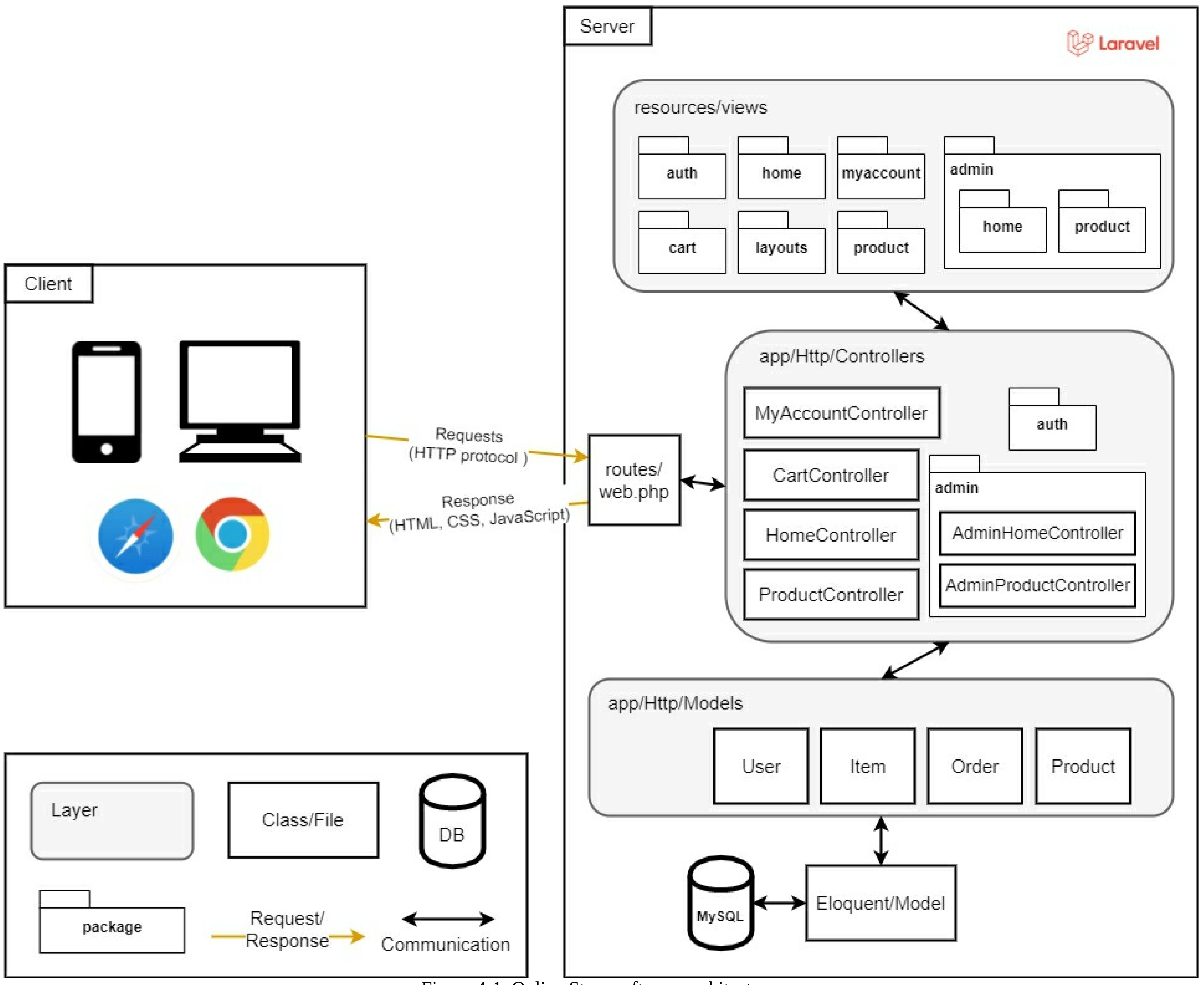


Figure 4-1. Online Store software architecture.

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

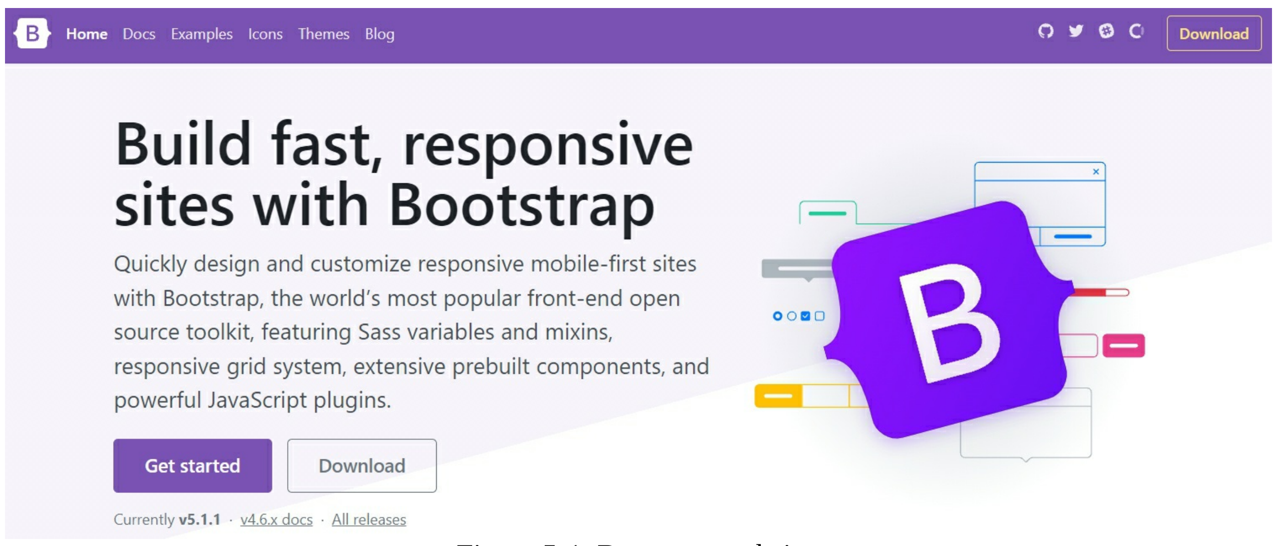
```
@if (count($records) > 0)
   I have records!
@else
   I don't have any records!
@endif
```

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/.



Build fast, responsive sites with Bootstrap

Quickly design and customize responsive mobile-first sites with Bootstrap, the world's most popular front-end open source toolkit, featuring Sass variables and mixins, responsive grid system, extensive prebuilt components, and powerful JavaScript plugins.

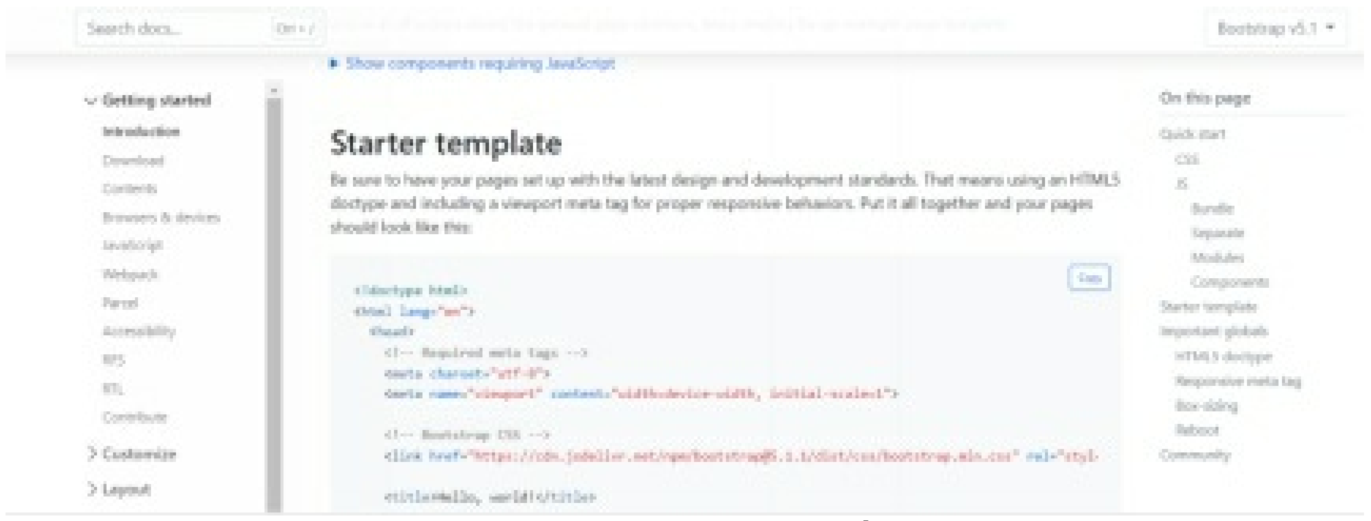
Figure 5-1. Bootstrap website.

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/.



Starter template

Be sure to have your pages set up with the latest design and development standards. That means using an HTML5 doctype and including a viewport meta tag for proper responsive behaviors. Put it all together and your pages should look like this:

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.min.css" rel="styl

    <title>Hello, world!</title>
```

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).

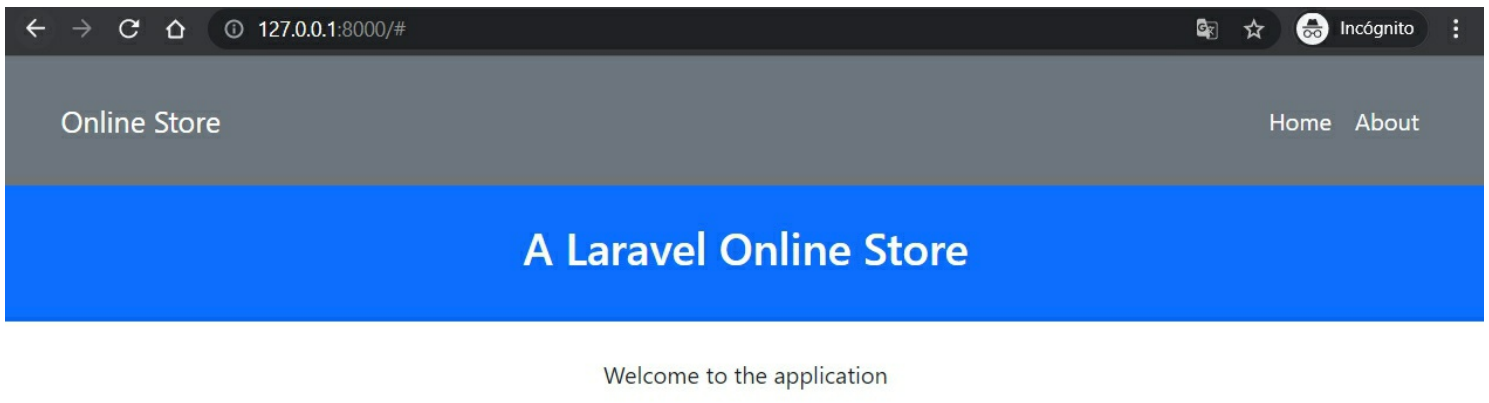


Figure 5-3. Application home page with the layout.

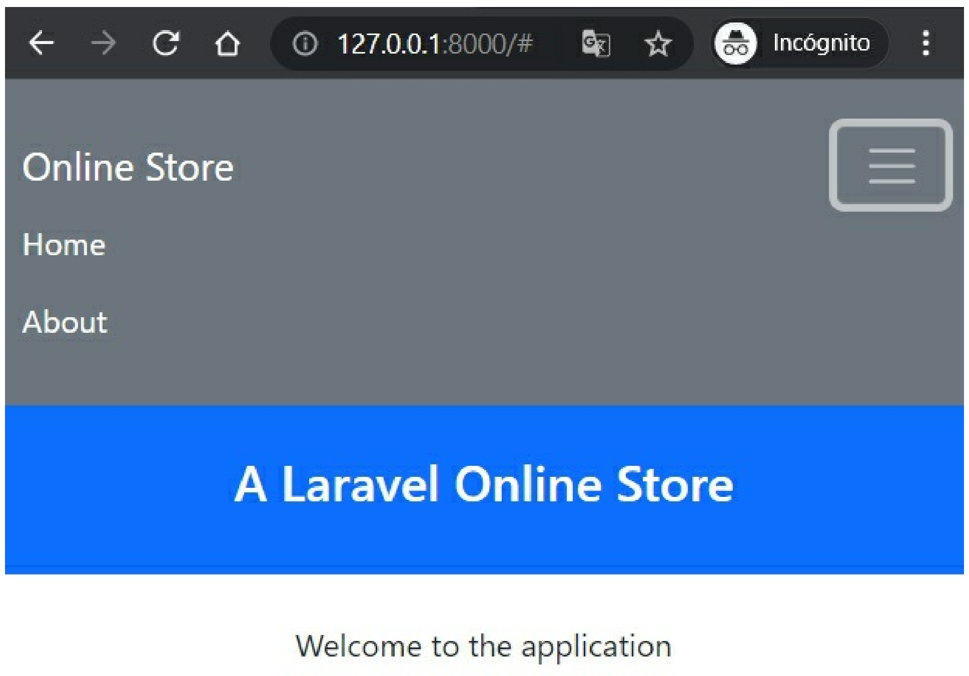


Figure 5-4. Application home page with reduced browser window width.

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