

LARAVEL 12

TRAINING KIT

A Practical Guide to Modern Web
Development



Agus Kurniawan

Laravel 12 Training Kit: A Practical Guide to Modern Web Development

Agus Kurniawan

Ilmu Data

1 June 2025

© 2025 Ilmu Data. All rights reserved.

OceanofPDF.com

Table of Contents

[Preface](#)

[Acknowledgments](#)

[1 Introduction to PHP Laravel 12](#)

[1.1 Overview of Laravel 12](#)

[1.2 Laravel Ecosystem and Features](#)

[1.3 Why Laravel for Modern Web Development?](#)

[1.3.1 Developer Experience](#)

[1.3.2 Rapid Development](#)

[1.3.3 Scalable Architecture](#)

[1.3.4 Security First](#)

[1.3.5 Rich Community and Ecosystem](#)

[1.3.6 Future-Ready](#)

[1.4 Installing Laravel 12](#)

[1.4.1 PHP Version](#)

[1.4.2 Using Composer](#)

[1.4.3 Using Laravel Installer](#)

[1.5 Installing Node.js and NPM](#)

[1.6 Setting Up the Development Environment](#)

[1.6.1 Composer, PHP, and MySQL Setup](#)

[1.6.2 Laravel Editor and IDE](#)

[1.7 Laravel Directory Structure and Conventions](#)

[1.8 Exercise 1: Creating Your First Laravel 12 Web Application](#)

[1.8.1 Description](#)

[1.8.2 Objectives](#)

[1.8.3 Prerequisites](#)

[1.8.4 Steps](#)

[1.8.5 Summary](#)

[1.9 Laravel Troubleshooting: SQLite Issues on PHP for Windows, Linux, and macOS](#)

[1.9.1 Problem](#)

[1.9.2 Solutions by Platform](#)

[1.9.3 Summary](#)

1.10 Conclusion

2 How to Use This Book

2.1 Introduction

2.2 Source Code

2.3 Feedback

3 Laravel Fundamentals

3.1 What is MVC?

3.2 Laravel Routing

3.3 Middleware in Laravel

3.4 Request and Response Handling

3.4.1 Accessing Request Data

3.4.2 Form Requests

3.4.3 Returning Responses

3.5 Controller and Views

3.6 Blade Templating Engine

3.7 Exercise 2: Creating a Route, Controller, and Blade View in Laravel 12

3.7.1 Description

3.7.2 Objectives

3.7.3 Prerequisites

3.7.4 Steps

3.7.5 Summary

3.8 Exercise 3: Creating a Laravel 12 Web Calculator

3.8.1 Description

3.8.2 Objectives

3.8.3 Prerequisites

3.8.4 Steps

3.8.5 Summary

3.9 Conclusion

4 Controller

4.1 What is a Controller?

4.2 Creating a Controller

4.3 Types of Controllers in Laravel

4.3.1 Basic Controllers

4.3.2 Resource Controllers

4.3.3 Invokable Controllers

4.4 Grouping Routes with Controllers

[4.5 Request Injection and Dependency Injection](#)

[4.6 Validating Requests in Controllers](#)

[4.7 Returning Responses from Controllers](#)

[4.8 Best Practices for Controllers](#)

[4.9 Exercise 4: Handling Request and Returning Views in Laravel 12](#)

[4.9.1 Description](#)

[4.9.2 Objectives](#)

[4.9.3 Prerequisites](#)

[4.9.4 Steps](#)

[4.9.5 Summary](#)

[4.10 Exercise 5: Use Route Grouping for Clean Definitions in Laravel 12](#)

[4.10.1 Description](#)

[4.10.2 Objectives](#)

[4.10.3 Prerequisites](#)

[4.10.4 Steps](#)

[4.10.5 Summary](#)

[4.11 Exercise 6: Prefix Grouping with Route Namespaces in Laravel 12](#)

[4.11.1 Description](#)

[4.11.2 Objectives](#)

[4.11.3 Prerequisites](#)

[4.11.4 Steps](#)

[4.11.5 Summary](#)

[4.12 Conclusion](#)

[5 Dependency Injection](#)

[5.1 What is Dependency Injection?](#)

[5.2 Laravel's Service Container](#)

[5.3 Constructor Injection](#)

[5.4 Method Injection](#)

[5.5 Binding Interfaces to Implementations](#)

[5.6 Singleton Bindings](#)

[5.7 Contextual Binding](#)

[5.8 Tagged Services and Service Discovery](#)

[5.9 Dependency Injection in Artisan Commands, Jobs, Events, and Listeners](#)

[5.10 Best Practices](#)

[5.11 Exercise 7: Exploring Dependency Injection in Laravel 12](#)

[5.11.1 Description](#)

[5.11.2 Objectives](#)

[5.11.3 Prerequisites](#)

[5.11.4 Steps](#)

[5.11.5 Summary](#)

[5.12 Exercise 8: Contextual Binding with Dependency Injection in Laravel 12](#)

[5.12.1 Description](#)

[5.12.2 Objectives](#)

[5.12.3 Prerequisites](#)

[5.12.4 Steps](#)

[5.12.5 Summary](#)

[5.13 Exercise 9: Dynamic Service Binding Based on Config or Environment](#)

[5.13.1 Description](#)

[5.13.2 Objectives](#)

[5.13.3 Prerequisites](#)

[5.13.4 Steps](#)

[5.13.5 Summary](#)

[5.14 Conclusion](#)

[6 View with Blade View](#)

[6.1 Introduction to Blade](#)

[6.2 Creating Your First Blade View](#)

[6.3 Passing Data to Views](#)

[6.4 Blade Syntax and Directives](#)

[6.5 Layouts and Sections](#)

[6.6 Blade Components](#)

[6.7 Blade Includes](#)

[6.8 Blade Loops and Conditionals](#)

[6.9 Displaying Validation Errors](#)

[6.10 Best Practices](#)

[6.11 Exercise 10: Passing Data from Controller to Blade View](#)

[6.11.1 Description](#)

[6.11.2 Objectives](#)

[6.11.3 Prerequisites](#)

[6.11.4 Steps](#)

[6.11.5 Summary](#)

[6.12 Exercise 11: Use Blade Control Structures](#)

[6.12.1 Description](#)

[6.12.2 Objectives](#)

[6.12.3 Prerequisites](#)

[6.12.4 Steps](#)

[6.12.5 Summary](#)

[6.13 Exercise 12: Layout and Personalization in Laravel 12 with Bootstrap](#)

[6.13.1 Description](#)

[6.13.2 Objectives](#)

[6.13.3 Prerequisites](#)

[6.13.4 Steps](#)

[6.13.5 Summary](#)

[6.14 Exercise 13: Using Partial Views in Laravel 12](#)

[6.14.1 Description](#)

[6.14.2 Objectives](#)

[6.14.3 Prerequisites](#)

[6.14.4 Steps](#)

[6.14.5 Summary](#)

[6.15 Exercise 14: Using Blade Components in Laravel 12](#)

[6.15.1 Description](#)

[6.15.2 Objectives](#)

[6.15.3 Prerequisites](#)

[6.15.4 Steps](#)

[6.15.5 Summary](#)

[6.16 Exercise 15: Implementing Light and Dark Themes in Laravel 12](#)

[6.16.1 Description](#)

[6.16.2 Objectives](#)

[6.16.3 Prerequisites](#)

[6.16.4 Steps](#)

[6.16.5 Summary](#)

[6.17 Conclusion](#)

[7 Form Submission and Data Validation](#)

- [7.1 Introduction](#)
- [7.2 Creating a Basic HTML Form](#)
- [7.3 Handling Form Submission in Controller](#)
- [7.4 Understanding CSRF Protection](#)
- [7.5 Displaying Validation Errors](#)
- [7.6 Exercise 16: Handle Form Submission with Validation in Laravel 12](#)
 - [7.6.1 Description](#)
 - [7.6.2 Objectives](#)
 - [7.6.3 Prerequisites](#)
 - [7.6.4 Steps](#)
 - [7.6.5 Summary](#)
- [7.7 Exercise 17: Custom Validation Rules and Messages in Laravel 12](#)
 - [7.7.1 Description](#)
 - [7.7.2 Objectives](#)
 - [7.7.3 Prerequisites](#)
 - [7.7.4 Steps](#)
 - [7.7.5 Summary](#)
- [7.8 Exercise 18: Use Regex and Conditional Validation in Laravel 12](#)
 - [7.8.1 Description](#)
 - [7.8.2 Objectives](#)
 - [7.8.3 Prerequisites](#)
 - [7.8.4 Steps](#)
 - [7.8.5 Summary](#)
- [7.9 Exercise 19: Multi-Step Form Submission with Session Data in Laravel 12](#)
 - [7.9.1 Description](#)
 - [7.9.2 Objectives](#)
 - [7.9.3 Prerequisites](#)
 - [7.9.4 Steps](#)
 - [7.9.5 Summary](#)
- [7.10 Exercise 20: Form Submission with AJAX and Validation Response in Laravel 12](#)
 - [7.10.1 Description](#)
 - [7.10.2 Objectives](#)

[7.10.3 Prerequisites](#)

[7.10.4 Steps](#)

[7.10.5 Summary](#)

[7.11 Conclusion](#)

[8 Model](#)

[8.1 Introduction](#)

[8.2 Understanding Models in Laravel](#)

[8.3 Entities and POCO \(Plain Old Class Object\)](#)

[8.4 Data Transfer Object \(DTO\)](#)

[8.5 Repository Pattern](#)

[8.6 Exercise 21: Using Model for Form Binding and Display](#)

[8.6.1 Description](#)

[8.6.2 Objectives](#)

[8.6.3 Prerequisites](#)

[8.6.4 Steps](#)

[8.6.5 Summary](#)

[8.7 Exercise 22: Build and Use a Data Transfer Object \(DTO\)](#)

[8.7.1 Description](#)

[8.7.2 Objectives](#)

[8.7.3 Prerequisites](#)

[8.7.4 Steps](#)

[8.7.5 Summary](#)

[8.8 Exercise 23: Use Model Accessors and Mutators](#)

[8.8.1 Description](#)

[8.8.2 Objectives](#)

[8.8.3 Prerequisites](#)

[8.8.4 Steps](#)

[8.8.5 Summary](#)

[8.9 Conclusion](#)

[9 File and Storage Management](#)

[9.1 Introduction](#)

[9.2 Configuration](#)

[9.3 Storing Files](#)

[9.3.1 Uploading via Form](#)

[9.3.2 Retrieving Files](#)

[9.3.3 Downloading Files](#)

[9.3.4 Deleting Files](#)

9.3.5 File Existence and Metadata

9.4 Public vs Private Files

9.5 Working with Remote Disks

9.6 Exercise 24: Upload and Display Image

9.6.1 Description

9.6.2 Objectives

9.6.3 Prerequisites

9.6.4 Steps

9.6.5 Summary

9.7 Exercise 25: Upload and Display Image to MinIO (S3-Compatible)

9.7.1 Description

9.7.2 Objectives

9.7.3 Prerequisites

9.7.4 Steps

9.7.5 Summary

9.8 Exercise 26: Upload and Display Image from Private Storage

9.8.1 Description

9.8.2 Objectives

9.8.3 Prerequisites

9.8.4 Steps

9.8.5 Summary

9.9 Exercise 27: Upload and Display Image to MinIO using Signed URLs

9.9.1 Description

9.9.2 Objectives

9.9.3 Prerequisites

9.9.4 Steps

9.9.5 Summary

9.10 Conclusion

10 Database and Eloquent ORM

10.1 Laravel Database Configuration

10.2 Migrations and Seeders

10.2.1 Migrations

10.2.2 Seeders

10.3 Working with Eloquent ORM

[10.3.1 Creating an Eloquent Model](#)

[10.3.2 Basic Eloquent Operations](#)

[10.4 Query Builder vs. Eloquent ORM](#)

[10.4.1 Query Builder](#)

[10.4.2 Eloquent ORM](#)

[10.4.3 Comparison](#)

[10.5 Exercise 28: Building a Simple Todo Web App with Eloquent ORM and SQLite](#)

[10.5.1 Description](#)

[10.5.2 Objectives](#)

[10.5.3 Prerequisites](#)

[10.5.4 SQLite Driver Notes](#)

[10.5.5 Steps](#)

[10.5.6 Test Features](#)

[10.5.7 Summary](#)

[10.6 Exercise 29: Building a Simple Todo Web App with Laravel 12, Eloquent ORM, and MySQL](#)

[10.6.1 Description](#)

[10.6.2 Objectives](#)

[10.6.3 Prerequisites](#)

[10.6.4 MySQL Driver Notes](#)

[10.6.5 Steps](#)

[10.6.6 Summary](#)

[10.7 Exercise 30: Eloquent ORM Relationships: One-to-One, One-to-Many, Many-to-Many](#)

[10.7.1 Description](#)

[10.7.2 Objectives](#)

[10.7.3 Prerequisites](#)

[10.7.4 Steps](#)

[10.7.5 Summary](#)

[10.8 Exercise 31: Pagination with Eloquent ORM \(SQLite\)](#)

[10.8.1 Description](#)

[10.8.2 Objectives](#)

[10.8.3 Prerequisites](#)

[10.8.4 Steps](#)

[10.8.5 Summary](#)

[10.9 Conclusion](#)

[11 Query Builder and Repository Pattern](#)

[11.1 Introduction to Query Builder](#)

[11.2 Basic Syntax and Usage](#)

[11.2.1 Selecting Records](#)

[11.2.2 Filtering with Where Clauses](#)

[11.2.3 Ordering, Limiting, and Offsetting](#)

[11.2.4 Aggregates](#)

[11.2.5 Joins](#)

[11.3 Insert, Update, and Delete](#)

[11.3.1 Inserting Records](#)

[11.3.2 Updating Records](#)

[11.3.3 Deleting Records](#)

[11.4 Query Builder and Raw Expressions](#)

[11.4.1 Selecting with Raw Expressions](#)

[11.4.2 Using Raw in Where Clauses](#)

[11.4.3 Order By with Raw](#)

[11.5 Introduction to the Repository Pattern](#)

[11.6 Exercise 32: Performing CRUD with Query Builder in Laravel 12](#)

[11.6.1 Description](#)

[11.6.2 Objectives](#)

[11.6.3 Prerequisites](#)

[11.6.4 Steps](#)

[11.6.5 Summary](#)

[11.7 Exercise 33: Query Builder for Joins, Aggregate, and Filtering](#)

[11.7.1 Description](#)

[11.7.2 Objectives](#)

[11.7.3 Prerequisites](#)

[11.7.4 Steps](#)

[11.7.5 Summary](#)

[11.8 Exercise 34: Query Builder with Raw Expressions and Secure Input Parameters](#)

[11.8.1 Description](#)

[11.8.2 Objectives](#)

[11.8.3 Prerequisites](#)

[11.8.4 Steps](#)

11.8.5 Summary

11.9 Exercise 35: Implementing Repository Pattern using Query Builder

11.9.1 Description

11.9.2 Objectives

11.9.3 Prerequisites

11.9.4 Steps

11.9.5 Summary

11.10 Conclusion

12 NoSQL Databases and Laravel 12

12.1 Introduction to NoSQL Databases

12.2 When to Use NoSQL vs SQL

12.3 Installing MongoDB for Laravel

12.4 Configuring MongoDB in Laravel

12.5 Creating a MongoDB Model

12.6 CRUD Operations with MongoDB

12.7 Using MongoDB Aggregation and Queries

12.8 Real-World Use Case: Product Inventory

12.9 MongoDB and Eloquent Relationships (Limitations)

12.10 Exercise 36: CRUD Web App with MongoDB in Laravel 12

12.10.1 Description

12.10.2 Objectives

12.10.3 Prerequisites

12.10.4 Steps

12.10.5 Summary

12.11 Exercise 37: MongoDB Embedded Relationships with Laravel 12

12.11.1 Description

12.11.2 Objectives

12.11.3 Prerequisites

12.11.4 Steps

12.11.5 Summary

12.12 Exercise 38: Data Pagination with MongoDB in Laravel 12

12.12.1 Description

12.12.2 Objectives

[12.12.3 Prerequisites](#)

[12.12.4 Steps](#)

[12.12.5 Summary](#)

[12.13 Conclusion](#)

[13 Authentication and Authorization](#)

[13.1 Introduction](#)

[13.2 Setting Up Authentication with Laravel Breeze](#)

[13.3 Understanding the Authentication Flow](#)

[13.4 Managing Users and Profiles](#)

[13.5 Securing Routes with Middleware](#)

[13.6 Implementing Authorization with Gates and Policies](#)

[13.7 Role and Permission Management](#)

[13.8 Exercise 39: Authentication and Authorization with Laravel 12 Breeze](#)

[13.8.1 Description](#)

[13.8.2 Objectives](#)

[13.8.3 Prerequisites](#)

[13.8.4 Steps](#)

[13.8.5 Summary](#)

[13.9 Exercise 40: Restrict Access Based on Role in Laravel 12](#)

[13.9.1 Description](#)

[13.9.2 Objectives](#)

[13.9.3 Prerequisites](#)

[13.9.4 Steps](#)

[13.9.5 Summary](#)

[13.10 Exercise 41: Authentication with Google using Laravel Socialite in Laravel 12](#)

[13.10.1 Description](#)

[13.10.2 Objectives](#)

[13.10.3 Prerequisites](#)

[13.10.4 Steps](#)

[13.10.5 Summary](#)

[13.11 Conclusion](#)

[14 REST API Development with Laravel](#)

[14.1 Introduction to REST APIs](#)

[14.2 Setting Up the Laravel API Project](#)

[14.3 Defining API Routes](#)

- [14.4 Building the Product API](#)
- [14.5 Using JSON Resources for API Responses](#)
- [14.6 Input Validation and Error Handling](#)
- [14.7 Securing APIs with Laravel Sanctum](#)
- [14.8 Pagination and Filtering](#)
- [14.9 Best Practices in API Development](#)
- [14.10 Exercise 42: Building a “Hello World” REST API in Laravel 12](#)
 - [14.10.1 Description](#)
 - [14.10.2 Objectives](#)
 - [14.10.3 Prerequisites](#)
 - [14.10.4 Steps](#)
 - [14.10.5 Summary](#)
- [14.11 Exercise 43: Building a Calculator REST API in Laravel 12](#)
 - [14.11.1 Description](#)
 - [14.11.2 Objectives](#)
 - [14.11.3 Prerequisites](#)
 - [14.11.4 Steps](#)
 - [14.11.5 Summary](#)
- [14.12 Exercise 44: Creating a CRUD Resource REST API](#)
 - [14.12.1 Description](#)
 - [14.12.2 Objectives](#)
 - [14.12.3 Prerequisites](#)
 - [14.12.4 Steps](#)
 - [14.12.5 Summary](#)
- [14.13 Exercise 45: Upload File via REST API with Form Data in Laravel 12](#)
 - [14.13.1 Description](#)
 - [14.13.2 Objectives](#)
 - [14.13.3 Prerequisites](#)
 - [14.13.4 Steps](#)
 - [14.13.5 Summary](#)
- [14.14 Exercise 46: Authentication & Authorization REST API](#)
 - [14.14.1 Description](#)
 - [14.14.2 Objectives](#)
 - [14.14.3 Prerequisites](#)

[14.14.4 Steps](#)

[14.14.5 Summary](#)

[14.15 Conclusion](#)

[15 Frontend Development with Starter Kits](#)

[15.1 Introduction to Laravel Frontend Scaffolding](#)

[15.2 React Starter Kit](#)

[15.3 Vue.js Starter Kit](#)

[15.4 Livewire Starter Kit](#)

[15.5 Comparison: React vs Vue.js vs Livewire](#)

[15.6 Exercise 47: Full Stack Product CRUD with Laravel 12 API and React UI](#)

[15.6.1 Description](#)

[15.6.2 Objectives](#)

[15.6.3 Prerequisites](#)

[15.6.4 Steps](#)

[15.6.5 Summary](#)

[15.7 Exercise 48: Full Stack Product CRUD with Laravel 12 API and Vue.js UI](#)

[15.7.1 Description](#)

[15.7.2 Objectives](#)

[15.7.3 Prerequisites](#)

[15.7.4 Steps](#)

[15.7.5 Summary](#)

[15.8 Conclusion](#)

[16 Laravel 12 Security – How to Harden the Web App](#)

[16.1 Introduction](#)

[16.2 Understanding Common Threats](#)

[16.3 Cross-Site Scripting \(XSS\) Protection](#)

[16.4 SQL Injection Prevention](#)

[16.5 CSRF Protection](#)

[16.6 Authentication and Authorization Best Practices](#)

[16.7 Securing Sessions and Cookies](#)

[16.8 Force HTTPS and TLS](#)

[16.9 Secure File Uploads](#)

[16.10 Set Security Headers](#)

[16.11 Rate Limiting and Throttling](#)

[16.12 Logging and Monitoring](#)

[16.13 Security Testing and Audits](#)

[16.14 Conclusion](#)

[17 Monitoring and Deployment](#)

[17.1 Introduction](#)

[17.2 Monitoring Laravel Applications](#)

[17.2.1 Laravel Telescope \(Local Development Debugging\)](#)

[17.2.2 Laravel Log Monitoring](#)

[17.3 Laravel 12 Deployment Best Practices](#)

[17.4 Exercise 49: Exploring Laravel Telescope in Laravel 12](#)

[17.4.1 Description](#)

[17.4.2 Objectives](#)

[17.4.3 Prerequisites](#)

[17.4.4 Steps](#)

[17.4.5 Summary](#)

[17.5 Laravel Deployment](#)

[17.6 Containerizing Laravel 12 with Docker](#)

[17.6.1 Why Use Docker?](#)

[17.6.2 Installing Docker](#)

[17.7 Exercise 50: Deploying Laravel 12 to Production on Ubuntu Server with PHP 8.4](#)

[17.7.1 Description](#)

[17.7.2 Objectives](#)

[17.7.3 Prerequisites](#)

[17.7.4 Steps](#)

[17.7.5 Summary](#)

[17.8 Exercise 51: Deploying Laravel 12 to Production in Docker on Ubuntu Server](#)

[17.8.1 Description](#)

[17.8.2 Objectives](#)

[17.8.3 Prerequisites](#)

[17.8.4 Steps](#)

[17.8.5 Summary](#)

[17.9 Conclusion](#)

[Appendix A: PHP Cheat Sheet](#)

[Appendix B: Resources](#)

[Enhance Your Learning with Our Udemy Course](#)

[Build Secure PHP APIs Like a Pro with Laravel 12, OAuth2, and JWT](#)

[Master Real-World Logging & Visualization with the Full ELK Stack](#)

[Appendix C: Source Code](#)

[About](#)

[OceanofPDF.com](#)

Preface

In the dynamic world of web development, keeping up with the latest tools and frameworks is essential for building modern, efficient, and scalable applications. Laravel 12 continues to set the standard for PHP frameworks, offering developers a powerful, elegant, and expressive toolkit for crafting robust web solutions.

“Laravel 12 Training Kit: A Practical Guide to Modern Web Development” is designed to help you master the essentials and advanced features of Laravel 12. This book provides a hands-on approach, guiding you through real-world scenarios and practical examples to ensure you gain both the knowledge and confidence needed to build professional web applications.

Throughout this book, you will explore:

- The core concepts and new features introduced in Laravel 12.
- Step-by-step instructions for setting up and configuring Laravel projects.
- Best practices for organizing your codebase for maintainability and scalability.
- Advanced topics such as authentication, authorization, API development, and testing.
- Practical examples that demonstrate Laravel’s capabilities in real-world projects.
- Tips for deploying, optimizing, and maintaining your Laravel applications.

Whether you are new to Laravel or an experienced developer seeking to deepen your understanding of the latest version, this book serves as a comprehensive resource. It is also valuable for students, IT professionals, and anyone interested in modern PHP web development.

Agus Kurniawan

Depok, June 2025

OceanofPDF.com

Acknowledgments

A heartfelt thank you to the Laravel community, contributors, and all technology enthusiasts whose feedback and support have been invaluable in shaping this book.

As you explore these chapters, I hope you find *Laravel 12 Training Kit: A Practical Guide to Modern Web Development* both informative and inspiring, helping you unlock new possibilities and creativity in your journey as a web developer.

OceanofPDF.com

1 Introduction to PHP Laravel 12

1.1 Overview of Laravel 12

Laravel is a powerful and elegant PHP framework designed for modern web application development. Since its first release in 2011, Laravel has evolved rapidly, offering developers a rich set of tools and expressive syntax that significantly reduces the effort required to build robust web applications.

Laravel 12 continues this tradition with enhanced performance, improved developer experience, and modern features aligned with the latest PHP standards. With built-in support for routing, middleware, ORM, authentication, testing, and more, Laravel is not just a framework—it's a complete development ecosystem.

Some highlights of Laravel 12:

- Full support for **PHP 8.3**
- Enhanced **Eloquent ORM** capabilities
- Improved **route model binding** and **route caching**
- **Configuration caching** for faster performance
- Seamless **frontend integration** with Laravel Vite
- Better support for **dependency injection** and **service container** patterns

1.2 Laravel Ecosystem and Features

The Laravel ecosystem offers a comprehensive suite of official tools and packages that enhance productivity and accelerate development:

Tool	Description
Laravel Breeze / Jetstream / Fortify	Authentication scaffolding options with different stacks
Laravel Sanctum / Passport	API authentication with token-based access
Eloquent ORM	Fluent and powerful ORM for working with databases
Blade Templating Engine	Simple and efficient view templating engine

Tool	Description
Artisan CLI	Command-line interface to automate tasks and generate boilerplate code
Laravel Mix / Vite	Frontend asset bundling and hot module replacement
Laravel Horizon	Dashboard and monitoring for queue workers
Laravel Nova	Premium administration panel for managing data
Laravel Scout	Full-text search integration
Laravel Telescope	Debugging and profiling tool for applications
Laravel Valet / Sail	Lightweight local development environments

In addition, Laravel integrates smoothly with frontend frameworks like **React**, **Vue.js**, and **Alpine.js**, and provides first-class support for **REST APIs**, **GraphQL**, and **WebSockets**.

1.3 Why Laravel for Modern Web Development?

Laravel is the framework of choice for many developers due to its expressive syntax, developer-friendly design, and commitment to modern practices. Here's why Laravel excels in modern web development:

1.3.1 Developer Experience

Laravel offers a clean and elegant syntax that encourages readable and maintainable code. Its intuitive conventions reduce boilerplate and let developers focus on what matters most—building features.

1.3.2 Rapid Development

With Artisan CLI, scaffolding tools, and built-in features, Laravel significantly speeds up the development process. Tasks like database migrations, seeding, and authentication setup can be done in minutes.

1.3.3 Scalable Architecture

Laravel supports a wide range of architectural patterns—from monolithic applications to microservices and serverless deployments. Features like queues, broadcasting, events, and caching allow apps to scale easily.

1.3.4 Security First

Laravel includes out-of-the-box security features like CSRF protection, input validation, password hashing, and rate limiting, ensuring secure application development.

1.3.5 Rich Community and Ecosystem

Laravel has one of the most active and supportive communities in the PHP ecosystem. It also boasts an extensive package ecosystem that covers everything from payment integration to admin panels.

1.3.6 Future-Ready

Laravel aligns closely with modern PHP and web development standards, embracing tools like Composer, PHPUnit, Vite, Docker (via Laravel Sail), and more. It also supports modern API development practices and can be deployed to cloud environments like Laravel Forge or Vapor.

1.4 Installing Laravel 12

Laravel can be installed globally via Composer or by using Laravel Installer. Before you begin, ensure you have **PHP ≥ 8.2**, **Composer**, and a database like **MySQL** installed on your system.

1.4.1 PHP Version

In this book, we will use the latest version of PHP, which is **PHP 8.4** or higher. You can check your PHP version by running:

```
| php -v
```

If you have an older version, consider upgrading to take advantage of the latest features and performance improvements.

1.4.2 Using Composer

Composer is the recommended way to install Laravel. It manages dependencies and allows you to create new projects easily.

To install Composer, follow the instructions on the [Composer website](#).

We can install Laravel 12 using Composer with the following command:

```
| composer create-project laravel/laravel:^12.0 myapp
```

This command will create a new Laravel 12 project in a folder named `myapp`.

If you don't specify a version, Composer will install the latest stable version of Laravel.

```
| composer create-project laravel/laravel myapp
```

This will create a new Laravel project with the latest version available.

1.4.3 Using Laravel Installer

Laravel Installer is a global Composer package that provides a command-line tool for creating new Laravel projects.

To install the Laravel Installer, first ensure you have Composer installed. Then, run the following command:

```
| composer global require laravel/installer
```

After installation, we can check if the installer is available by running:

```
| laravel --version
```

Then create a new Laravel 12 project:

```
| laravel new myapp
```

Tip: Add Composer's global bin to your system path: `~/.composer/vendor/bin` (macOS/Linux) or `%USERPROFILE%\AppData\Roaming\Composer\vendor\bin` (Windows).

1.5 Installing Node.js and NPM

Laravel uses **Node.js** and **NPM** for managing frontend assets. You can install Node.js from the Node.js website, <https://nodejs.org/>. Select the LTS version for stability based on your system architecture (Windows, macOS, or Linux).

After installation, verify by running:

```
| node -v  
| npm -v
```

If you have Node.js installed, NPM is included by default.

You can install the latest version of Node.js using a package manager like Homebrew (macOS) or Chocolatey (Windows).

1.6 Setting Up the Development Environment

Laravel is flexible with local development options. You can use:

- **Traditional setup** with PHP, MySQL, and Composer installed directly on your machine
- **Docker-based setup** using Laravel Sail for a containerized and portable environment

1.6.1 Composer, PHP, and MySQL Setup

For a traditional setup, make sure the following are installed:

Tool	Minimum Version
PHP	8.2 or higher
Composer	Latest version
MySQL	5.7+ or MariaDB 10.3+

You can verify with:

```
| php -v  
| composer -V  
| mysql --version
```

Once installed:

```
| composer create-project laravel/laravel:^12.0 myapp
| cd myapp
| php artisan serve
```

Visit <http://localhost:8000> to see the Laravel welcome page.

1.6.2 Laravel Editor and IDE

Laravel works well with various code editors and IDEs. Some popular choices include:

- **Visual Studio Code:** Lightweight, extensible, and has great support for PHP and Laravel.
- **PhpStorm:** A powerful IDE with built-in support for Laravel, including code completion, debugging, and testing.
- **Sublime Text:** Fast and customizable, with many plugins available for PHP and Laravel development.
- **Atom:** A hackable text editor with a vibrant community and many packages for PHP development.

In this book, we will use **Visual Studio Code** as our primary editor. You can install it from Visual Studio Code <https://code.visualstudio.com/>.

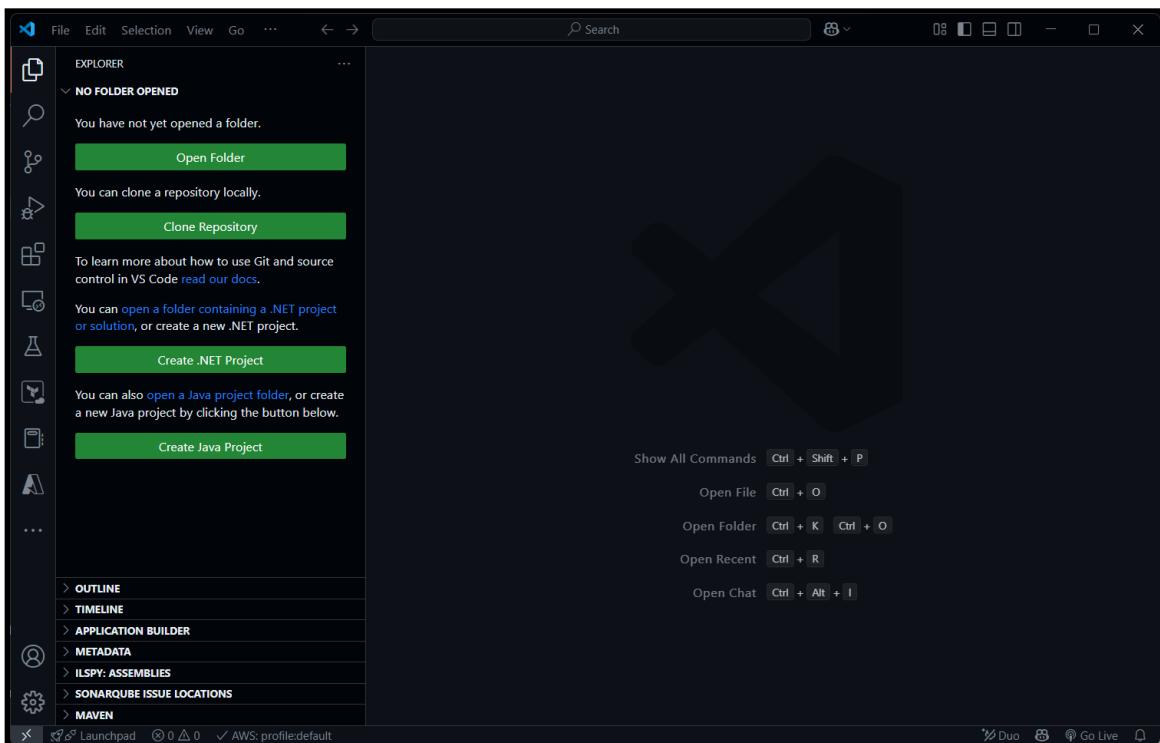


Figure 1.1: Visual Studio Code.

You may need extensions for PHP and Laravel support: **PHP Intelephense**, provides intelligent code completion and linting for PHP.

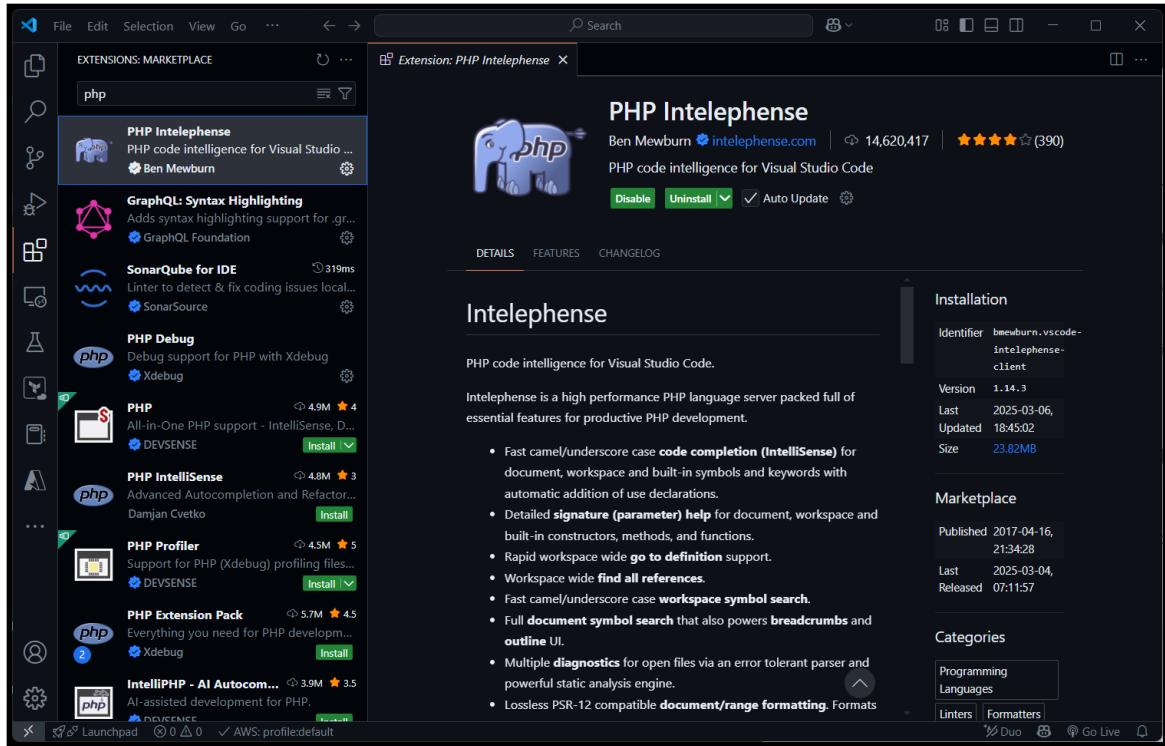


Figure 1.2: PHP Intelephense.

1.7 Laravel Directory Structure and Conventions

Once installed, your Laravel project follows a clean and modular directory structure:

```
myapp/
  app/                      # Core application code (MVC)
    ├── Console/
    ├── Exceptions/
    ├── Http/
    └── Models/                # Controllers, Middleware, Requests
  bootstrap/                 # App bootstrapping (autoloaders)
  config/                    # Configuration files
  database/                 # Migrations, seeders, factories
  lang/                      # Language localization
  public/                    # Entry point for web server (index.php)
  resources/                # Views (Blade), assets (CSS/JS)
  routes/                   # Web/API/Console routes
  storage/                  # Logs, compiled views, file uploads
  tests/                     # Unit and feature tests
  vendor/                   # Composer dependencies
```

Key Conventions:

- **MVC Architecture:** Models in `app/Models`, Views in `resources/views`, Controllers in `app/Http/Controllers`
- **Routing:** Defined in `routes/web.php` (browser) and `routes/api.php` (API)
- **Configuration:** Environment variables in `.env` and config files in `config/`
- **Database:** Migrations and seeders in `database/` directory
- **Artisan CLI:** Use `php artisan` for generating code, migrations, and more

*Laravel's structure encourages **clean separation of concerns** and supports **test-driven development**, making it ideal for modern applications.*

1.8 Exercise 1: Creating Your First Laravel 12 Web Application

1.8.1 Description

In this lab, we will create a new Laravel 12 web application using Composer, set it up in Visual Studio Code, and run it using Laravel's built-in development server.

1.8.2 Objectives

By the end of this lab, you will be able to:

- Install Laravel 12 using Composer
- Open and configure the project in Visual Studio Code
- Run the Laravel development server
- View your first Laravel web app in the browser

1.8.3 Prerequisites

Make sure the following tools are installed:

- PHP ≥ 8.2, <https://www.php.net/downloads.php>
- Composer, <https://getcomposer.org/>
- Visual Studio Code, <https://code.visualstudio.com/>
- Basic terminal or command prompt knowledge

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

1.8.4 Steps

Here are the steps to create your first Laravel 12 web application:

1.8.4.1 Step 1: Open Terminal or Command Prompt

Open your terminal (macOS/Linux) or Command Prompt / PowerShell (Windows).

1.8.4.2 Step 2: Create a New Laravel Project

Navigate to the directory where you want to create your Laravel project and run:

```
| composer create-project laravel/laravel:^12.0 myfirstapp
```

This will create a new Laravel 12 project in a folder named `myfirstapp`.

1.8.4.3 Step 3: Open the Project in Visual Studio Code

```
| cd myfirstapp  
| code .
```

`code .` will open the current directory in Visual Studio Code. Make sure `code` command is available in your terminal. You can install it from within VS Code:

`Cmd+Shift+P > Shell Command: Install 'code' command in PATH.`

1.8.4.4 Step 4: Explore the Project Structure

In VS Code, review the following directories: - `app/Http/Controllers`: where you will define controllers - `routes/web.php`: defines the web routes - `resources/views`: Blade templates for HTML views - `.env`: environment configuration file

1.8.4.5 Step 5: Run Laravel's Built-in Development Server

In the terminal (inside VS Code or system terminal), run:

```
| php artisan serve
```

You will see output similar to:

```
| Starting Laravel development server: http://127.0.0.1:8000
```

1.8.4.6 Step 6: Open the App in Browser

Go to your browser and visit:

```
| http://127.0.0.1:8000
```

You should see the Laravel welcome page!

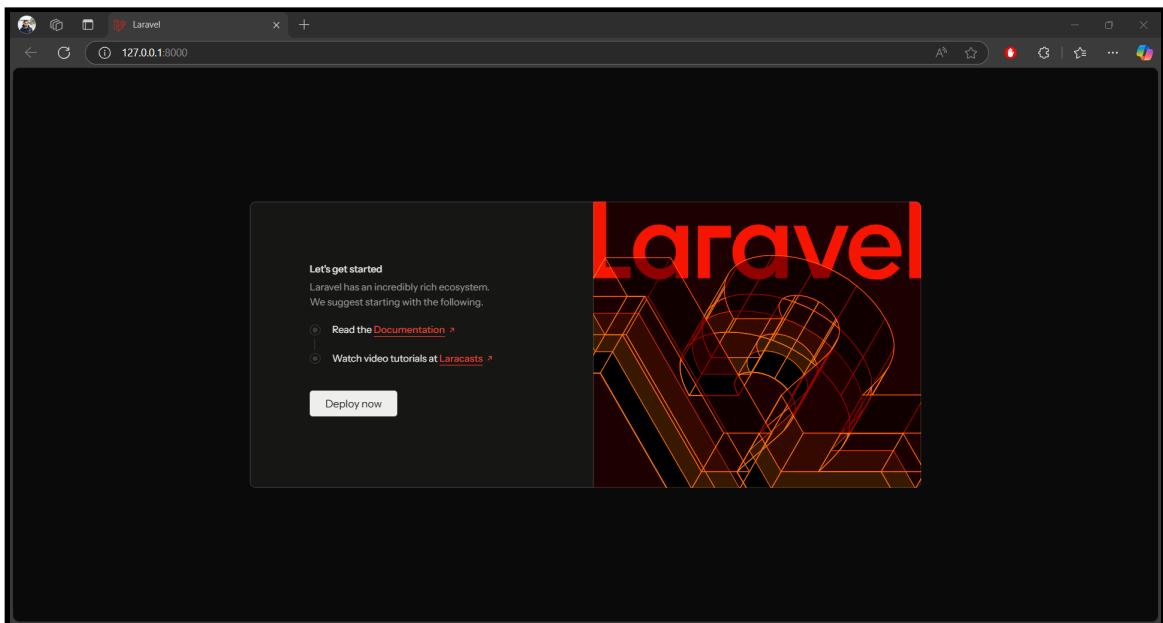


Figure 1.3: Laravel 12 web app.

1.8.5 Summary

In this lab, you:

- Created a new Laravel 12 project using Composer
- Opened the project in Visual Studio Code
- Learned the basic folder structure of a Laravel app
- Ran the development server using `php artisan serve`
- Visited your first Laravel app in the browser

1.9 Laravel Troubleshooting: SQLite Issues on PHP for Windows, Linux, and macOS

Laravel 12 uses **SQLite** as the default database in the `.env` file to simplify setup. However, many students may encounter issues when running migrations if the

SQLite environment isn't properly configured. This guide helps troubleshoot SQLite setup across **Windows**, **Linux**, and **macOS**.

1.9.1 Problem

When running:

```
| php artisan migrate
```

You might see an error like:

```
| Database (/database/database.sqlite) does not exist.
```

or

```
| could not find driver (SQLITE driver missing)
```

This indicates that the SQLite database file is missing or the SQLite driver is not enabled in your PHP configuration.

1.9.2 Solutions by Platform

Here are the solutions for each platform:

1.9.2.1 1. All Platforms — Ensure SQLite Database File Exists

By default, Laravel uses this in `.env`:

```
| DB_CONNECTION=sqlite
```

Or (simplified):

```
| DB_CONNECTION=sqlite
```

By default, Laravel 12 creates a SQLite database file at `database/database.sqlite`. If it doesn't exist, you need to create it.

Create the SQLite file manually:

```
| touch database/database.sqlite
```

Or on Windows (CMD or PowerShell):

```
| type nul > database\database.sqlite
```

1.9.2.2 2. Windows — Enable SQLite in php.ini

If you see:

```
| could not find driver
```

This means the SQLite driver is not enabled in your PHP configuration.

To enable it, you need to edit your `php.ini` file.

1. Open your `php.ini` (run `php --ini` to find it).

2. Make sure this line is **uncommented**:

```
| extension=pdo_sqlite  
| extension=sqlite3
```

3. Make `extension_dir` point to the correct directory where your SQLite extensions are located. For example:

```
| extension_dir = "ext"
```

This is the default for PHP installations.

Or if you installed PHP manually, it might look like this:

```
| extension_dir = "C:\php\ext"
```

4. Save and restart your terminal or development server.

You may need `sqlite3.dll` in your PHP directory. If not, download it from the SQLite website and place it in your PHP directory. Download it on <https://www.sqlite.org/download.html>. There are two versions of SQLite, 32-bit and 64-bit. Make sure to download the correct version for your system.

In my case, I downloaded the 64-bit version. The file name is `sqlite-dll-win64-x86-3350500.zip`. After unzipping, I copied the `sqlite3.dll` file to my PHP directory.

1.9.2.3 3. macOS — Ensure PHP has SQLite Support

macOS often includes SQLite support with PHP. But if you're using PHP via **Homebrew**:

Check if SQLite is installed:

```
| php -m | grep sqlite
```

Expected output:

```
| pdo_sqlite  
| sqlite3
```

If missing:

```
| brew install php
```

Or reinstall with:

```
| brew reinstall php
```

1.9.2.4 4. Linux (Ubuntu/Debian) — Install SQLite Extensions

We can install SQLite extensions using the package manager. For Ubuntu/Debian, run:

```
| sudo apt update  
| sudo apt install php-sqlite3
```

Then restart your PHP server (Apache/Nginx) or CLI:

```
| sudo service apache2 restart
```

Or for Laravel built-in server:

```
| php artisan serve
```

1.9.2.5 5. Optional — Switch to MySQL if SQLite Is Not Preferred

Edit `.env`:

```
| DB_CONNECTION=mysql  
| DB_HOST=127.0.0.1  
| DB_PORT=3306  
| DB_DATABASE=your_db_name  
| DB_USERNAME=your_username  
| DB_PASSWORD=your_password
```

Then configure your local MySQL/MariaDB instance accordingly.

1.9.3 Summary

Here's a summary of the solutions for SQLite issues in Laravel 12:

Platform	Fix
All	Manually create <code>database.sqlite</code> file
Windows	Enable <code>pdo_sqlite</code> and <code>sqlite3</code> in <code>php.ini</code>
macOS	Use <code>brew</code> to install PHP with SQLite support
Linux	Install <code>php-sqlite3</code> via <code>apt</code> or package manager

1.10 Conclusion

Laravel 12 is a powerful framework that simplifies modern web development. With its elegant syntax, rich ecosystem, and robust features, it's an excellent choice for building scalable and maintainable applications.

OceanofPDF.com

2 How to Use This Book

2.1 Introduction

Here are some general steps you can follow to use this book:

1. Read the introduction: Start by reading the introduction and any other sections that provide an overview of the lab or technology you will be working with. This will give you a better understanding of the goals and objectives of the lab.
2. Gather materials: Depending on the lab, you may need to gather specific materials or software tools. Make sure you have everything you need before you start.
3. Follow the instructions: The lab will provide a set of instructions that will guide you through the tasks you need to complete. Read each step carefully and make sure you understand what you need to do before moving on to the next step.
4. Perform the exercises: Perform the exercises or tasks as instructed. Take your time and make sure you understand what you're doing at each step.
5. Troubleshoot: If you encounter any problems or errors, try to troubleshoot the issue by referring back to the instructions or doing some research online. If you're still stuck, don't hesitate to ask for help from a mentor or online community.
6. Review and reflect: After completing each exercise or task, take some time to review what you've learned and reflect on your experience. This will help reinforce your knowledge and identify areas where you may need to spend more time.
7. Move on to the next exercise: Once you've completed all the exercises or tasks in the lab, move on to the next one. Repeat the steps above for each exercise until you've completed the entire lab.

This book can be a great way to learn new skills and technologies. Just make sure you take your time, follow the instructions carefully, and don't

hesitate to ask for help if you need it.

2.2 Source Code

You can download all sources codes on this book on
<https://www.github.com/aguskilmudata-book-laravel12>.

If you want to use and run the projects in this book, make to install the following tools:

```
| composer install  
| npm install
```

Then, you can run the project using `php artisan serve` command.

2.3 Feedback

If you have any specific questions or feedback about a particular hands-on lab book related to Laravel, feel free to ask and I'll do my best to assist you.

You can contact me on this email, agusk2007@gmail.com

OceanofPDF.com

3 Laravel Fundamentals

This chapter introduces you to the foundational building blocks of a Laravel application. Understanding routing, middleware, request/response handling, controllers, views, and Blade templating is essential to start developing with Laravel effectively.

Here's a **short technical review about MVC** that you can place in section **2.1 of Chapter 2** in your book "*Laravel 12 Training Kit: A Practical Guide to Modern Web Development*":

3.1 What is MVC?

MVC stands for **Model–View–Controller**, a software architectural pattern used to separate concerns in web applications:

- **Model:** Represents the data and business logic. In Laravel, this is typically handled by **Eloquent models**, which interact with the database.
- **View:** Handles the presentation layer—the HTML shown to the user. Laravel uses **Blade** as its templating engine for views.
- **Controller:** Manages the flow between the model and the view. It receives input from the user (via routes), processes it (often involving models), and returns the appropriate view or response.

This separation allows for: - **Cleaner code structure** - **Easier testing and maintenance** - **Better scalability in large applications**

Laravel fully embraces the MVC pattern, making it a reliable choice for structured and modern web development.

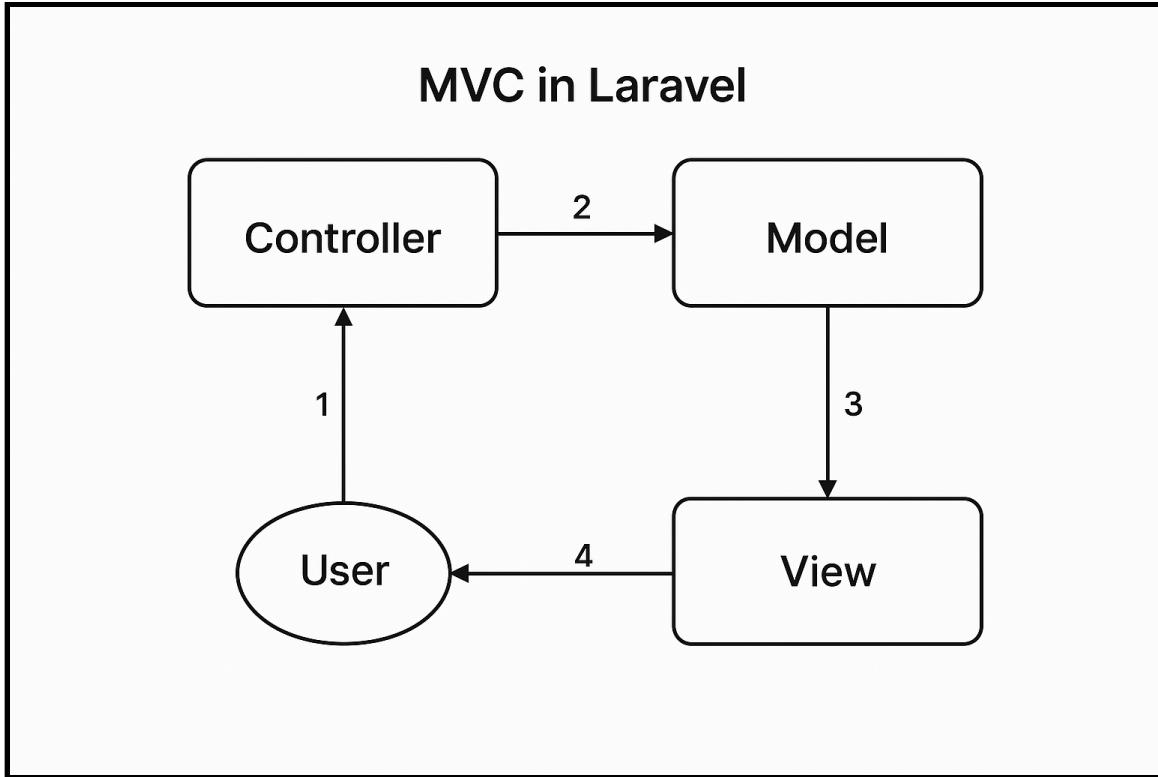


Figure 3.1: MVC diagram.

You can see the MVC diagram in [Figure 2.1](#). Here's an explanation of the **MVC in Laravel** diagram shown above:

The diagram illustrates how Laravel handles a user request using the **Model–View–Controller** (MVC) pattern:

① User → Controller

- A user makes a request, such as visiting a URL (e.g., `/posts`).
- Laravel's **Route** maps this URL to a specific **controller method**.

② Controller → Model

- The controller handles the request logic.
- It may interact with the **Model** to retrieve, update, or delete data from the database using **Eloquent ORM**.

③ Model → View

- The model returns the data to the controller.
- The controller then passes that data to a **View** for rendering.

④ View → User

- The **Blade view** generates the final HTML output.
- Laravel returns this output as the **HTTP response** to the user.

Here's a quick summary of the MVC components in Laravel:

Component	Description
Model	Represents the application data and business logic (Eloquent ORM)
View	Responsible for presenting data to the user (Blade templates)
Controller	Handles user input, coordinates between Model and View

This separation of concerns allows Laravel applications to be more modular, scalable, and easier to maintain.

3.2 Laravel Routing

Routing in Laravel defines how the application responds to incoming requests. It maps URIs to controller actions or closures.

Laravel routes are defined in the `routes/web.php` file for web requests and in `routes/api.php` for API routes.

Here is a simple route that returns a string when accessed:

```
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    return 'Hello, Laravel!';
});
```

Here's how to define a route with a parameter:

```
Route::get('/user/{id}', function ($id) {
    return "User ID: " . $id;
});
```

Having named routes allows you to generate URLs or redirects easily:

```
| Route::get('/dashboard', function () {
|     return view('dashboard');
| })->name('dashboard');
```

You can also route directly to a controller method:

```
| Route::get('/about', [AboutController::class, 'index']);
```

Laravel's routing system is powerful, supporting route groups, middleware, resource controllers, and more.

3.3 Middleware in Laravel

Middleware provides a mechanism for filtering HTTP requests entering your application. It acts like a layer that sits between the request and response cycle.

Here are some common uses of middleware in Laravel:

- Authentication (`auth`)
- Logging
- CORS
- Input sanitization

Here's how to assign middleware to a route:

```
| Route::get('/dashboard', function () {
|     return view('dashboard');
| })->middleware('auth');
```

You can create custom middleware using the Artisan command:

```
| php artisan make:middleware CheckAge
```

Then edit `app/Http/Middleware/EnsureUserIsAdmin.php` and register it in `app/Http/Kernel.php`.

We will explore middleware in more detail in next chapters.

3.4 Request and Response Handling

Laravel simplifies handling HTTP requests and generating responses.

3.4.1 Accessing Request Data

We can access request data using the `Request` object:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;

Route::get('/user', function (Request $request) {
    $name = $request->input('name');
    return "Hello, " . $name;
});
```

You can also access query parameters, form data, and JSON payloads easily.

In a controller or route closure:

```
$request->input('name');
$request->query('page'); // for query string ?page=1
```

Or use helper:

```
$name = request('name');
```

3.4.2 Form Requests

Laravel allows you to validate and authorize incoming data via form request classes:

```
php artisan make:request StoreUserRequest
```

3.4.3 Returning Responses

Laravel provides a simple way to return responses. You can return strings, views, JSON, or even file downloads.

```
return response('Hello World', 200)
    ->header('Content-Type', 'text/plain');
```

Laravel also supports JSON responses, redirects, file downloads, and more.

3.5 Controller and Views

Controllers organize your application logic. They live in `app/Http/controllers`.

We can create a controller using the Artisan command:

```
| php artisan make:controller PageController
```

We can define a method in the controller to handle a specific route:

```
| public function home() {
|     return view('home');
| }
```

In `routes/web.php`, you can register a route to the controller method:

```
| Route::get('/home', [PageController::class, 'home']);
```

Views are stored in `resources/views`. A simple view file might look like this, `resources/views/home.blade.php`.

```
| <!DOCTYPE html>
| <html>
| <head>
|   <title>Home</title>
| </head>
| <body>
|   <h1>Welcome to Laravel 12!</h1>
| </body>
| </html>
```

3.6 Blade Templating Engine

Blade is Laravel's powerful, lightweight templating engine. It allows you to use plain PHP in your views with a cleaner syntax.

Here are some common Blade directives:

```
| <h1>Hello, {{ $name }}</h1>
```

Blade encourages clean, maintainable templates and integrates seamlessly with Laravel's routing and controllers.

We will explore Blade in more detail in the next chapters.

3.7 Exercise 2: Creating a Route, Controller, and Blade View in Laravel 12

3.7.1 Description

In this lab, we'll build a simple web page that displays a welcome message with dynamic data using Laravel's MVC architecture. You'll define a route, create a controller to handle the request logic, and display the output using a Blade view template.

3.7.2 Objectives

By the end of this lab, you will be able to:

- Define a route in Laravel
- Create a controller and pass data to a view
- Create and render a Blade template
- Understand the flow of MVC in Laravel

3.7.3 Prerequisites

Make sure you have:

- Laravel 12 installed
- PHP \geq 8.2 and Composer installed
- Visual Studio Code installed
- Basic terminal/command prompt experience

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

3.7.4 Steps

Here's a step-by-step guide to creating a simple Laravel web page using MVC:

3.7.4.1 Step 1: Open the Laravel Project in VS Code

If you haven't already created a project:

```
composer create-project laravel/laravel:^v12.0.3 mvc-demo
cd mvc-demo
code .
```

We created a new Laravel project named `mvc-demo` with Laravel project template v12.0.3 and opened it in Visual Studio Code.

This opens the project in Visual Studio Code.

3.7.4.2 Step 2: Define a Route

Open `routes/web.php` and add the following route:

```
| use App\Http\Controllers\WelcomeController;  
|  
| Route::get('/welcome', [WelcomeController::class, 'show']);
```

This route tells Laravel to call the `show()` method of `WelcomeController` when a user visits `/welcome`.

3.7.4.3 Step 3: Create the Controller

Run this command in your terminal:

```
| php artisan make:controller WelcomeController
```

Open the newly created file at `app/Http/Controllers/WelcomeController.php` and modify it:

```
| namespace App\Http\Controllers;  
|  
| use Illuminate\Http\Request;  
|  
| class WelcomeController extends Controller  
{  
|     public function show()  
|     {  
|         $name = "Laravel Learner";  
|         return view('mywelcome', ['name' => $name]);  
|     }  
| }
```

3.7.4.4 Step 4: Create the Blade View

Create a new file: `resources/views/mywelcome.blade.php`

Paste this HTML + Blade template code:

```
| <!DOCTYPE html>  
| <html>  
| <head>  
|     <title>Welcome</title>  
| </head>  
| <body>  
|     <h1>Welcome to Laravel 12!</h1>  
|     <p>Hello, {{ $name }} </p>  
| </body>  
| </html>
```

This view displays a dynamic message using the `$name` variable passed from the controller.

3.7.4.5 Step 5: Run the Laravel Development Server

In the terminal, run:

```
| php artisan serve
```

Visit this URL in your browser:

```
| http://127.0.0.1:8000/welcome
```

You should see the message:

```
| Welcome to Laravel 12!
| Hello, Laravel Learner
```

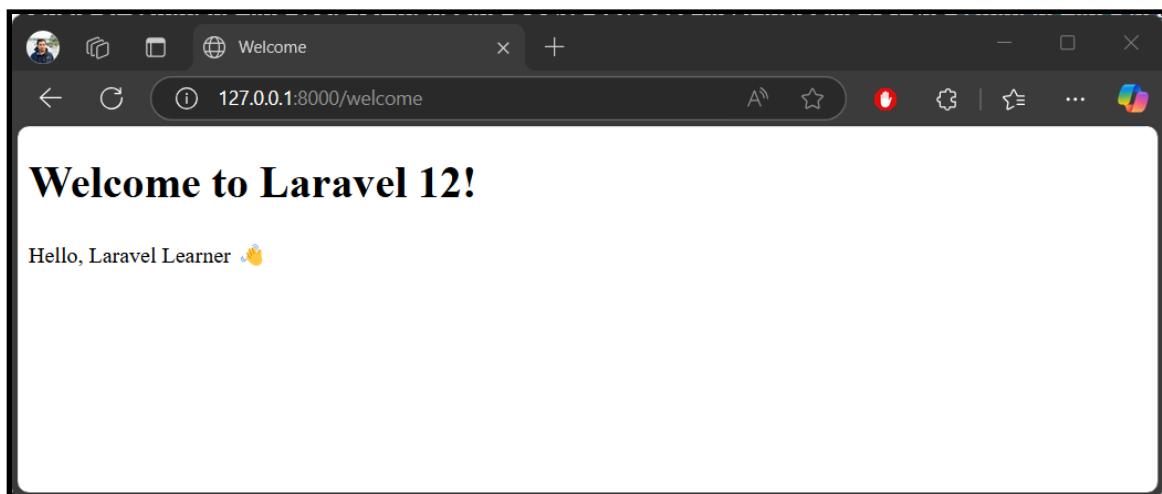


Figure 3.2: mvc-demo application.

3.7.5 Summary

In this lab, you:

- Created a route that connects a URL to a controller method
- Created a controller that processes logic and passes data to a view
- Created a Blade view that displays dynamic content
- Ran the app to visualize the complete MVC flow

This hands-on exercise demonstrates the heart of Laravel's MVC architecture. You're now ready to build more complex pages with routing, logic, and templating.

3.8 Exercise 3: Creating a Laravel 12 Web Calculator

3.8.1 Description

In this lab, we will create a simple calculator web app using Laravel 12. The calculator will take two numbers and an operator (add, subtract, multiply, divide), then display the result. This project demonstrates Laravel's MVC architecture with form input, routing, controllers, and Blade views.

3.8.2 Objectives

By the end of this lab, you will be able to:

- Handle form submissions in Laravel
- Use routes and controllers to process user input
- Pass and display calculated results in Blade views
- Understand MVC interaction through a practical scenario

3.8.3 Prerequisites

Ensure the following are installed: - PHP \geq 8.2 - Composer - Laravel 12 - Visual Studio Code - Basic terminal usage

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

3.8.4 Steps

Here's a step-by-step guide to creating a simple calculator web app using Laravel 12:

3.8.4.1 Step 1: Create and Open Laravel Project

We use Laravel 12.0.3 as the project template. In your terminal:

```
composer create-project laravel/laravel:^12.0.3 calculator
cd calculator
code .
```

You should see the Laravel project structure in Visual Studio Code.

3.8.4.2 Step 2: Define the Route

We will create two routes: one for displaying the calculator form and another for processing the calculation.

Open routes/web.php and add two routes:

```
use App\Http\Controllers\CalculatorController;

Route::get('/calculator', [CalculatorController::class, 'index']);
Route::post('/calculator', [CalculatorController::class, 'calculate'])->name('calculator.ca
```

The first route displays the calculator form, while the second route processes the form submission.

3.8.4.3 Step 3: Create the Controller

Run the following command:

```
| php artisan make:controller CalculatorController
```

Open app/Http/Controllers/CalculatorController.php **and modify it:**

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class CalculatorController extends Controller
{
    public function index()
    {
        return view('calculator');
    }

    public function calculate(Request $request)
    {
        $validated = $request->validate([
            'number1' => 'required|numeric',
            'number2' => 'required|numeric',
            'operator' => 'required|in:add,sub,mul,div'
        ]);
    }
}
```

```

    ]);

    $result = match ($validated['operator']) {
        'add' => $validated['number1'] + $validated['number2'],
        'sub' => $validated['number1'] - $validated['number2'],
        'mul' => $validated['number1'] * $validated['number2'],
        'div' => $validated['number2'] != 0 ? $validated['number1'] / $validated['number2']
    };

    return view('calculator', [
        'result' => $result,
        'number1' => $validated['number1'],
        'number2' => $validated['number2'],
        'operator' => $validated['operator']
    ]);
}
}

```

3.8.4.4 Step 4: Create the Blade View

Create a file at `resources/views/calculator.blade.php` with the following content:

```

<!DOCTYPE html>
<html>
<head>
    <title>Laravel Calculator</title>
</head>
<body>
    <h1>Simple Calculator</h1>

    @if ($errors->any())
        <div style="color: red;">
            <ul>
                @foreach ($errors->all() as $error)
                    <li>{{ $error }}</li>
                @endforeach
            </ul>
        </div>
    @endif

    <form method="POST" action="{{ route('calculator.calculate') }}">
        @csrf
        <input type="number" name="number1" value="{{ old('number1', $number1 ?? '') }}">
        placeholder="First Number" required>
        <select name="operator" required>
            <option value="add" {{ ($operator ?? '') == 'add' ? 'selected' : '' }}>+
        </option>
            <option value="sub" {{ ($operator ?? '') == 'sub' ? 'selected' : '' }}>-
        </option>
            <option value="mul" {{ ($operator ?? '') == 'mul' ? 'selected' : '' }}>*
        </option>
            <option value="div" {{ ($operator ?? '') == 'div' ? 'selected' : '' }}>/</option>
        </select>
        <input type="number" name="number2" value="{{ old('number2', $number2 ?? '') }}">
        placeholder="Second Number" required>
        <button type="submit">Calculate</button>
    </form>

```

```
| @isset($result)
|     <h3>Result: {{ $result }}</h3>
| @endisset
| </body>
| </html>
```

This view contains a form for user input and displays the result after calculation. It also handles validation errors.

3.8.4.5 Step 5: Run the Application

Run Laravel's built-in server:

```
| php artisan serve
```

Open your browser and go to:

```
| http://127.0.0.1:8000/calculator
```

Try different values and operators to see the result calculated and displayed.

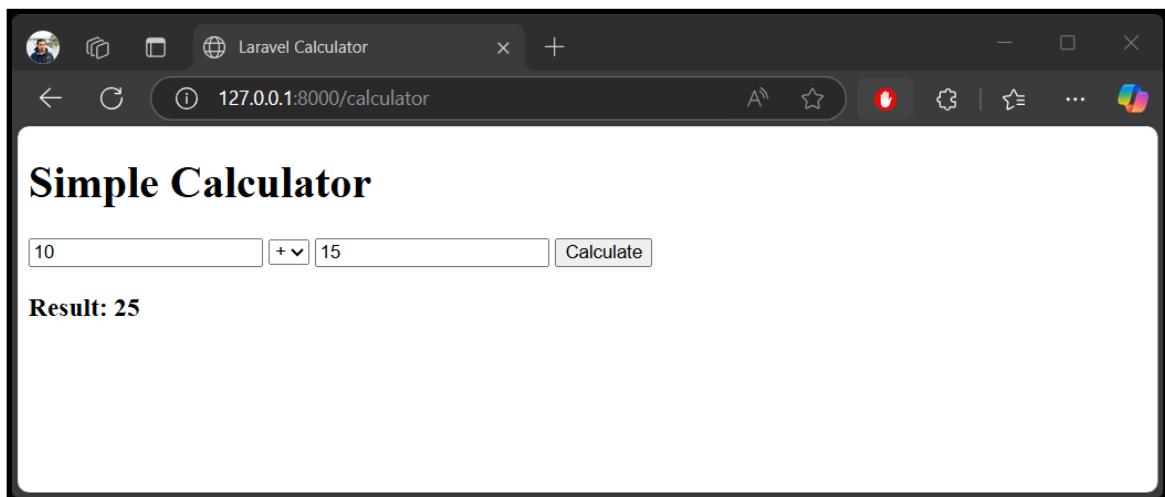


Figure 3.3: calculator application.

3.8.5 Summary

In this hands-on lab, you:

- Built a simple calculator web app using Laravel 12
- Learned to handle form input with POST requests
- Created a controller to handle logic and return results to a view
- Practiced using Blade templating to display data

This project showcases Laravel's MVC pattern in action and sets the foundation for more advanced applications with user interaction and business logic.

3.9 Conclusion

In this chapter, we covered the foundational concepts of Laravel 12, including routing, middleware, request/response handling, controllers, views, and Blade templating. We also created a simple web application to demonstrate the MVC architecture in action.

OceanofPDF.com

4 Controller

Controllers in Laravel are essential for handling application logic. They receive requests from routes, process the input, and return responses. This chapter introduces the different types of controllers, how to create them, and best practices for organizing your controller logic in Laravel 12.

4.1 What is a Controller?

In the MVC (Model-View-Controller) pattern, a **controller** acts as the bridge between the **model** and the **view**. It handles user input, interacts with models for data, and returns the correct response—often rendering a view.

4.2 Creating a Controller

To generate a controller, use the Artisan command:

```
| php artisan make:controller PageController
```

This creates a controller at:

```
| app/Http/Controllers/PageController.php
```

Here's a simple controller that returns a view:

```
| namespace App\Http\Controllers;  
  
| use Illuminate\Http\Request;  
  
| class PageController extends Controller  
{  
|     public function home()  
|     {  
|         return view('home');  
|     }  
| }
```

Register it in the route file:

```
| use App\Http\Controllers\PageController;  
  
| Route::get('/home', [PageController::class, 'home']);
```

4.3 Types of Controllers in Laravel

Laravel supports several controller types to support various development patterns:

4.3.1 Basic Controllers

Standard classes with multiple methods for different endpoints.

4.3.2 Resource Controllers

Ideal for CRUD operations using RESTful conventions.

```
| php artisan make:controller ProductController --resource
```

This creates predefined methods like: - `index()` - `create()` - `store()` - `show()` - `edit()` - `update()` - `destroy()`

Register using:

```
| Route::resource('products', ProductController::class);
```

4.3.3 Invokable Controllers

Useful for single-action endpoints.

```
| php artisan make:controller ContactController --invokable
```

Define a route:

```
| Route::get('/contact', ContactController::class);
```

4.4 Grouping Routes with Controllers

You can group routes for better organization:

```
| Route::controller(UserController::class)->group(function () {
|     Route::get('/users', 'index');
|     Route::get('/users/{id}', 'show');
| });
| 
```

4.5 Request Injection and Dependency Injection

Laravel allows you to inject the `Request` object or any service into a controller method.

```
| use Illuminate\Http\Request;  
  
| public function store(Request $request)  
{  
|     $data = $request->all();  
|     // process the data  
| }
```

You can also inject services:

```
| public function index(UserService $service)  
{  
|     return $service->getAllUsers();  
| }
```

4.6 Validating Requests in Controllers

Inline validation:

```
| public function store(Request $request)  
{  
|     $request->validate([  
|         'name' => 'required|string|max:255',  
|         'email' => 'required|email|unique:users'  
|     ]);  
| }
```

Using **Form Request Classes** (recommended for cleaner code):

```
| php artisan make:request StoreUserRequest
```

Then:

```
| public function store(StoreUserRequest $request)  
{  
|     $validated = $request->validated();  
| }
```

4.7 Returning Responses from Controllers

Controllers can return various response types:

- **View**

```
| return view('welcome');
```

- **JSON**

```
| return response()->json(['status' => 'success']);
```

- **Redirect**

```
| return redirect()->route('home');
```

- **Custom Response**

```
| return response('Hello', 200)->header('Content-Type', 'text/plain');
```

4.8 Best Practices for Controllers

- Keep controller methods **short and focused**.
- Offload heavy logic to **services or model methods**.
- Use **Form Request classes** to manage validation.
- Follow **RESTful naming conventions** for consistency.
- Group related routes and controllers to keep your app organized.

4.9 Exercise 4: Handling Request and Returning Views in Laravel 12

4.9.1 Description

In this lab, you will learn how to pass data from a controller to a view, handle route parameters, and use query strings. We'll use simple examples and minimal Blade templates to help you focus on the request–response lifecycle.

4.9.2 Objectives

By the end of this lab, you will be able to:

- Pass variables from a controller to a Blade view
- Use parameterized routes in Laravel
- Retrieve query string parameters from the URL

4.9.3 Prerequisites

- Laravel 12 installed

- PHP ≥ 8.2, Composer, and Visual Studio Code
- Basic knowledge of Laravel routing and terminal commands

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

4.9.4 Steps

Here's a step-by-step guide to create a simple web application that demonstrates request handling and data passing in Laravel 12.

4.9.4.1 Step 1: Create and Open Laravel Project

We will create a new Laravel project named `lab-view` using Composer. Open your terminal and run:

```
| composer create-project laravel/laravel:^12.0.3 lab-view
| cd lab-view
| code .
```

You should see the Laravel project structure in Visual Studio Code.

4.9.4.2 Step 2: Create a Controller

```
| php artisan make:controller DemoController
```

Open `app/Http/Controllers/DemoController.php` and add these methods:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class DemoController extends Controller
{
    // Simple data passing
    public function hello()
    {
        $name = 'Laravel Learner';
        return view('hello', ['name' => $name]);
    }

    // Parameterized route
    public function greet($name)
    {
        return view('greet', ['name' => ucfirst($name)]);
    }
}
```

```
// Query string
public function search(Request $request)
{
    $keyword = $request->query('q', 'none');
    return view('search', ['keyword' => $keyword]);
}
```

4.9.4.3 Step 3: Define Routes

Edit `routes/web.php`:

```
use App\Http\Controllers\DemoController;

Route::get('/hello', [DemoController::class, 'hello']);
Route::get('/greet/{name}', [DemoController::class, 'greet']);
Route::get('/search', [DemoController::class, 'search']);
```

4.9.4.4 Step 4: Create Simple Views

Create these Blade files under `resources/views/`:

- * `hello.blade.php`
- * `greet.blade.php`
- * `search.blade.php`

View `hello.blade.php` is a simple HTML file that displays a greeting message.

```
<!DOCTYPE html>
<html>
<head><title>Hello</title></head>
<body>
    <h1>Hello, {{ $name }}!</h1>
</body>
</html>
```

View `greet.blade.php` is a simple HTML file that displays a personalized greeting message.

```
<!DOCTYPE html>
<html>
<head><title>Greet</title></head>
<body>
    <h1>Nice to meet you, {{ $name }}!</h1>
</body>
</html>
```

View `search.blade.php` is a simple HTML file that displays the search keyword.

```
<!DOCTYPE html>
<html>
<head><title>Search</title></head>
<body>
    <h1>You searched for: <strong>{{ $keyword }}</strong></h1>
</body>
</html>
```

Save all files.

4.9.4.5 Step 5: Test the Routes

Start the development server:

```
| php artisan serve
```

Visit the following URLs:

1. Passing data to view

`http://127.0.0.1:8000/hello`

→ Output: *Hello, Laravel Learner!*

2. Parameterized route

`http://127.0.0.1:8000/greet/thariq`

→ Output: *Nice to meet you, Thariq!*

3. Query string

`http://127.0.0.1:8000/search?q=laravel`

→ Output: *You searched for: laravel*

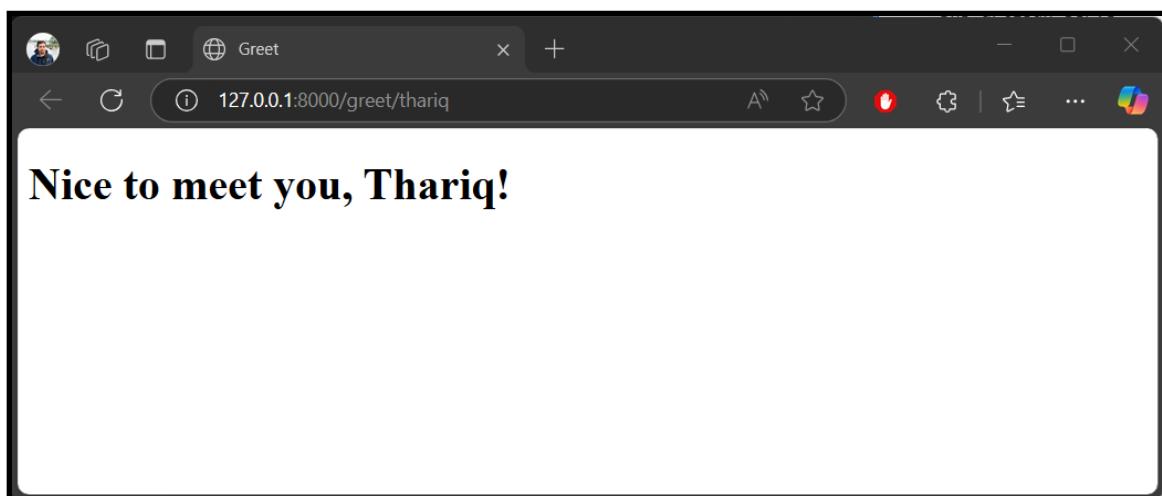


Figure 4.1: Output program from lab-view.

4.9.5 Summary

In this hands-on lab, you:

- Created a controller and defined route logic

- Passed data from controller to Blade views
- Used parameterized routes to extract dynamic URL values
- Retrieved query string parameters using `Request`

You now understand the basics of request handling and data passing in Laravel — skills you'll use in almost every Laravel project!

4.10 Exercise 5: Use Route Grouping for Clean Definitions in Laravel 12

4.10.1 Description

In this lab, you'll learn how to define multiple routes for a single controller using **route grouping**. This helps reduce repetition and keeps your routing clean and easy to maintain as your application grows.

4.10.2 Objectives

By the end of this lab, you will:

- Create a controller with multiple related actions
- Group routes using `Route::controller()` syntax
- Pass data from controller methods to simple Blade views

4.10.3 Prerequisites

- Laravel 12 installed
- PHP ≥ 8.2 and Composer
- Visual Studio Code
- Basic knowledge of Laravel routing and controllers

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

4.10.4 Steps

Here's a step-by-step guide to create a simple web application that demonstrates route grouping in Laravel 12.

4.10.4.1 Step 1: Create and Open Laravel Project

We will create a new Laravel project named `lab-group` using Composer. Open your terminal and run:

```
| composer create-project laravel/laravel:^12.0.3 lab-group
| cd lab-group
| code .
```

You should see the Laravel project structure in Visual Studio Code.

4.10.4.2 Step 2: Create a Controller

```
| php artisan make:controller PageController
```

Update the controller `app/Http/Controllers/PageController.php`:

```
| namespace App\Http\Controllers;
|
| class PageController extends Controller
| {
|     public function home()
|     {
|         $message = "Welcome to the homepage.";
|         return view('pages.home', compact('message'));
|     }
|
|     public function about()
|     {
|         $message = "This is the about page.";
|         return view('pages.about', compact('message'));
|     }
|
|     public function contact()
|     {
|         $message = "Reach us through the contact page.";
|         return view('pages.contact', compact('message'));
|     }
| }
```

Explain the code:

- The `PageController` is created in the `app/Http/Controllers` directory.
- The `PageController` has three methods: `home()`, `about()`, and `contact()`.
- Each method returns a view with a message passed to it.

- The `compact()` function creates an array from the variable name, making it available in the view.

This controller will handle all the routes for our pages.

4.10.4.3 Step 3: Define Grouped Routes

Open `routes/web.php` and define the route group:

```
use App\Http\Controllers\PageController;

Route::controller(PageController::class)->group(function () {
    Route::get('/', 'home')->name('home');
    Route::get('/about', 'about')->name('about');
    Route::get('/contact', 'contact')->name('contact');
});
```

This groups all routes handled by `PageController`, keeping the definition clean and DRY.

4.10.4.4 Step 4: Create Simple Views

Create the folder: `resources/views/pages/`

Then create the following files inside `pages/`:

- `home.blade.php`
- `about.blade.php`
- `contact.blade.php`

View `home.blade.php` is a simple HTML file that displays a welcome message.

```
<!DOCTYPE html>
<html>
<head><title>Home</title></head>
<body>
    <h1>{{ $message }}</h1>
</body>
</html>
```

View `about.blade.php` is a simple HTML file that displays an about message.

```
<!DOCTYPE html>
<html>
<head><title>About</title></head>
<body>
    <h1>{{ $message }}</h1>
</body>
</html>
```

View `contact.blade.php` is a simple HTML file that displays a contact message.

```
<!DOCTYPE html>
<html>
<head><title>Contact</title></head>
<body>
    <h1>{{ $message }}</h1>
</body>
</html>
```

4.10.4.5 Step 5: Run the Application

Start the development server:

```
| php artisan serve
```

Test in your browser:

- `http://127.0.0.1:8000/` → Home
- `http://127.0.0.1:8000/about` → About
- `http://127.0.0.1:8000/contact` → Contact

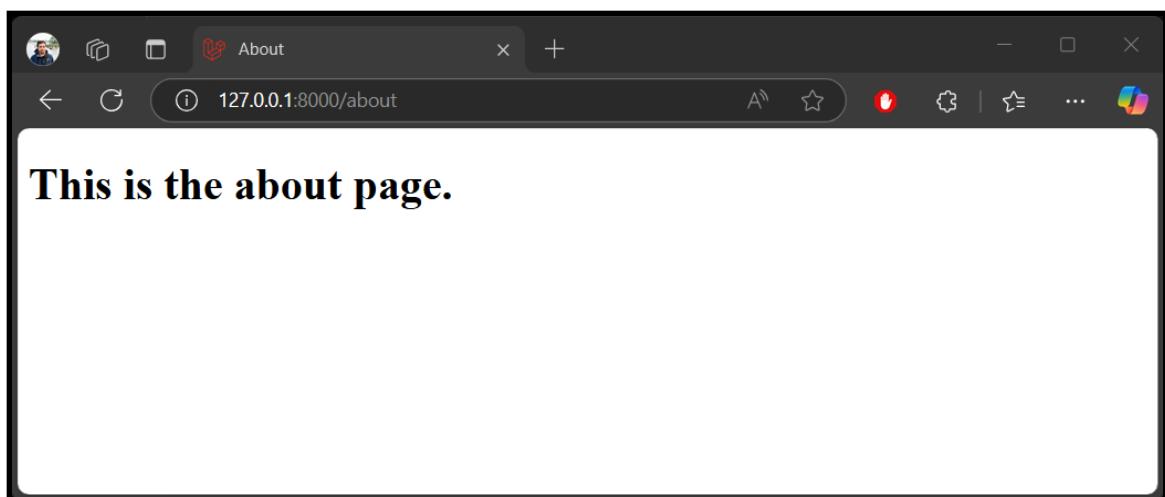


Figure 4.2: Output program from lab-group.

4.10.5 Summary

In this lab, you:

- Created a controller with multiple related actions
- Used `Route::controller(...)->group(...)` to group routes cleanly
- Returned simple Blade views with passed data

Route grouping is a great practice that keeps your `web.php` clean and improves maintainability as you scale your Laravel project.

4.11 Exercise 6: Prefix Grouping with Route Namespaces in Laravel 12

4.11.1 Description

In this lab, you'll learn how to group routes using a **common URL prefix** and organize controllers into **namespaced folders**. This improves route structure, especially for admin/user dashboards or API-like routing.

4.11.2 Objectives

By the end of this lab, you will be able to:

- Create grouped routes with a common URL prefix
- Organize controllers using folders (namespaces)
- Map controller actions to routes with grouping
- Return data to simple Blade views

4.11.3 Prerequisites

- Laravel 12
- PHP ≥ 8.2, Composer, Visual Studio Code
- Basic Laravel routing and controller knowledge

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

4.11.4 Steps

Here's a step-by-step guide to create a simple web application that demonstrates prefix grouping with route namespaces in Laravel 12.

4.11.4.1 Step 1: Create and Open Laravel Project

We will create a new Laravel project named `lab-prefix` using Composer. Open your terminal and run:

```
| composer create-project laravel/laravel:^12.0.3 lab-prefix
| cd lab-prefix
| code .
```

You should see the Laravel project structure in Visual Studio Code.

4.11.4.2 Step 2: Create Namespaced Controllers

Create a folder and two controllers using Artisan:

```
| php artisan make:controller Admin/DashboardController
| php artisan make:controller Admin/UserController
```

This will create two controllers in the `app/Http/Controllers/Admin/` directory:

```
| app/Http/Controllers/Admin/DashboardController.php
| app/Http/Controllers/Admin/UserController.php
```

4.11.4.3 Step 3: Define Route Groups with Prefix and Controller Namespace

We'll define a route group with a prefix of `admin` and map it to the `Admin` namespace. This allows us to keep our routes organized and easy to manage.

Edit `routes/web.php`:

```
| use App\Http\Controllers\Admin\DashboardController;
| use App\Http\Controllers\Admin\UserController;

Route::prefix('admin')->group(function () {
    Route::get('/dashboard', [DashboardController::class, 'index'])->name('admin.dashboard');
    Route::get('/users', [UserController::class, 'index'])->name('admin.users');
    Route::get('/users/{id}', [UserController::class, 'show'])->name('admin.users.show');
});
```

Save the file.

All routes in this group will be prefixed with `/admin`.

4.11.4.4 Step 4: Add Actions to Controllers

We'll create a simple action that returns a view with a welcome message. Here's the code for `DashboardController.php`:

```
namespace App\Http\Controllers\Admin;

use App\Http\Controllers\Controller;

class DashboardController extends Controller
{
    public function index()
    {
        return view('admin.dashboard', ['message' => 'Welcome to Admin Dashboard']);
    }
}
```

We'll create a simple action that returns a list of users. Here's the code for `UserController.php`:

```
namespace App\Http\Controllers\Admin;

use App\Http\Controllers\Controller;

class UserController extends Controller
{
    public function index()
    {
        $users = ['Ria', 'Lie', 'Jon'];
        return view('admin.users.index', compact('users'));
    }

    public function show($id)
    {
        $user = "User #" . $id;
        return view('admin.users.show', compact('user'));
    }
}
```

4.11.4.5 Step 5: Create Simple Views

Create folders and files under `resources/views/admin/`. Then, create the following files:

- `dashboard.blade.php`
- `users/index.blade.php`

View `dashboard.blade.php` is a simple HTML file that displays a welcome message.

```
| <!DOCTYPE html>
| <html>
| <head><title>Admin Dashboard</title></head>
| <body>
|   <h1>{{ $message }}</h1>
| </body>
| </html>
```

You can create a folder `users` under `resources/views/admin/` and create the file `index.blade.php` inside it. **View** `users/index.blade.php` is a simple HTML file that displays a list of users.

```
| <!DOCTYPE html>
| <html>
| <head><title>Users</title></head>
| <body>
|   <h1>User List</h1>
|   <ul>
|     @foreach ($users as $user)
|       <li>{{ $user }}</li>
|     @endforeach
|   </ul>
| </body>
| </html>
```

View `users/show.blade.php` is a simple HTML file that displays user details.

```
| <!DOCTYPE html>
| <html>
| <head><title>User Details</title></head>
| <body>
|   <h1>Details for: {{ $user }}</h1>
| </body>
| </html>
```

4.11.4.6 Step 6: Run the Application

Start the server:

```
| php artisan serve
```

Test these routes:

- `http://127.0.0.1:8000/admin/dashboard`
- `http://127.0.0.1:8000/admin/users`
- `http://127.0.0.1:8000/admin/users/2`

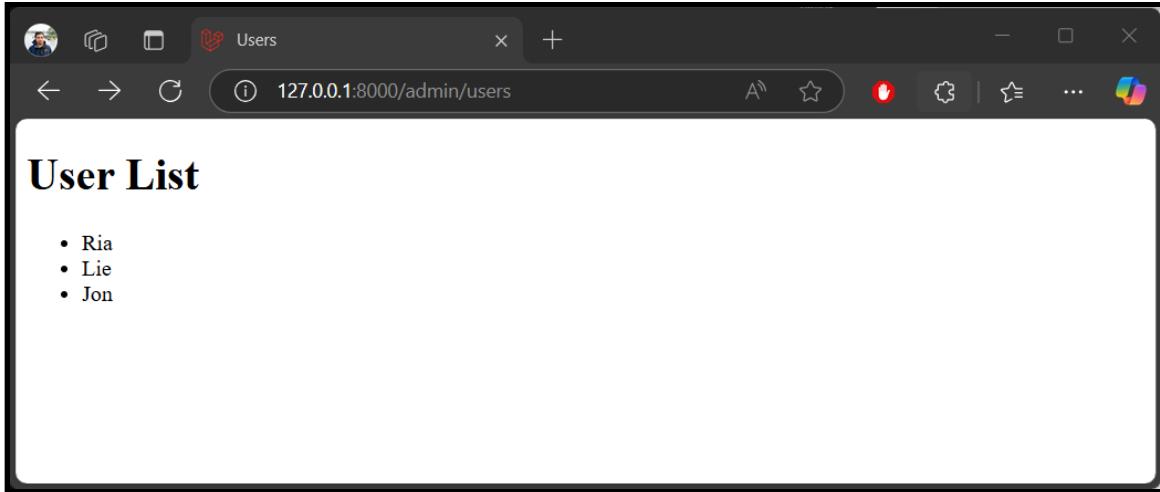


Figure 4.3: Output program from lab-prefix.

4.11.5 Summary

In this lab, you:

- Created controllers within a sub-namespace (`Admin`)
- Grouped routes under a URL prefix (`admin`)
- Routed multiple controllers using grouped structure
- Passed data from controller to view using simple templates

Prefix and namespace-based route grouping helps you manage feature areas like admin panels or modules in a scalable way.

4.12 Conclusion

In this chapter, we covered the basics of controllers in Laravel 12, including how to create them, define routes, and pass data to views. We also explored different types of controllers, such as resource and invokable controllers, and learned how to group routes for better organization.

We also discussed how to handle requests, validate input, and return various response types. Finally, we completed hands-on labs to reinforce these concepts by creating simple web applications that demonstrate request handling and route grouping.

This chapter serves as a foundation for understanding how controllers work in Laravel, and you will build upon this knowledge in the following chapters as we dive deeper into more advanced topics.

OceanofPDF.com

5 Dependency Injection

Dependency Injection (DI) is a design pattern used to reduce tight coupling between components in software. Laravel fully embraces DI and uses it extensively through its **Service Container**—a powerful tool for managing class dependencies and performing dependency resolution automatically.

5.1 What is Dependency Injection?

Dependency Injection means that instead of a class creating its dependencies, those dependencies are “injected” from the outside—usually via the constructor or method parameters.

This allows for more flexible, testable, and maintainable code.

For demo, let’s say we have a `ReportService` that uses a `FileLogger`:

```
class ReportService {  
    public function __construct() {  
        $this->logger = new FileLogger();  
    }  
}
```

This creates a tight coupling between `ReportService` and `FileLogger`. If we want to change the logger to `DatabaseLogger`, we have to modify the `ReportService` class.

```
class ReportService {  
    public function __construct() {  
        $this->logger = new DatabaseLogger();  
    }  
}
```

This is not ideal for testing or maintaining the code.

Now, let’s see how we can use **Dependency Injection** to improve this.

```
class ReportService {  
    public function __construct(FileLogger $logger) {  
        $this->logger = $logger;  
    }  
}
```

This approach makes code more **modular**, **testable**, and **maintainable**.

5.2 Laravel's Service Container

Laravel's **Service Container** is responsible for resolving and injecting dependencies automatically when a controller, route, or class is instantiated.

5.3 Constructor Injection

Most common and preferred method in Laravel.

Here's how you can inject a service into a controller using **constructor injection**:

```
namespace App\Http\Controllers;

use App\Services\ReportService;

class ReportController extends Controller
{
    protected $reportService;

    public function __construct(ReportService $reportService)
    {
        $this->reportService = $reportService;
    }

    public function index()
    {
        return $this->reportService->generate();
    }
}
```

Laravel automatically resolves the `ReportService` instance using the container.

5.4 Method Injection

You can inject dependencies directly into controller methods.

```
public function show(Request $request, ReportService $reportService)
{
    return $reportService->generate();
}
```

This works in **controller methods, routes, jobs, and more**.

5.5 Binding Interfaces to Implementations

Laravel allows you to bind an interface to a concrete class, useful for creating flexible, testable code.

First, create an interface in `app/Contracts`:

```
namespace App\Contracts;

interface LoggerInterface {
    public function log(string $message);
}
```

Then, create a class that implements the interface:

```
namespace App\Services;

use App\Contracts\LoggerInterface;

class FileLogger implements LoggerInterface {
    public function log(string $message) {
        // write to file
    }
}
```

Next, bind the interface to the implementation in a service provider, typically `AppServiceProvider`:

```
public function register()
{
    $this->app->bind(LoggerInterface::class, FileLogger::class);
}
```

Now, Laravel can resolve `LoggerInterface` automatically with `FileLogger`.

5.6 Singleton Bindings

You can bind a class as a singleton so that only one instance is used throughout the app.

```
$this->app->singleton(AnalyticsService::class, function ($app) {
    return new AnalyticsService(config('analytics.api_key'));
});
```

5.7 Contextual Binding

When you need different implementations of an interface in different contexts.

```
use App\Contracts\LoggerInterface;
use App\Services\DatabaseLogger;
```

```
use App\Http\Controllers\PaymentController;

$this->app->when(PaymentController::class)
    ->needs(LoggerInterface::class)
    ->give(DatabaseLogger::class);
```

5.8 Tagged Services and Service Discovery

Useful for resolving multiple implementations with a common interface.

```
$this->app->tag([ServiceA::class, ServiceB::class], 'report.generators');

$generators = $this->app->tagged('report.generators');
```

5.9 Dependency Injection in Artisan Commands, Jobs, Events, and Listeners

Laravel's container can also inject dependencies into:

- Artisan commands
- Job classes
- Event listeners
- Middleware

Example in a job:

```
public function __construct(PaymentService $service)
{
    $this->service = $service;
}
```

5.10 Best Practices

- Prefer constructor injection for required dependencies
- Use method injection for contextual or optional services
- Use interfaces for abstraction and testability
- Bind interfaces in `AppServiceProvider` or custom providers
- Avoid service location (`app()->make(...)`) in favor of automatic DI

5.11 Exercise 7: Exploring Dependency Injection in Laravel 12

5.11.1 Description

In this lab, you'll create a Laravel web app that demonstrates different types of **Dependency Injection (DI)** and **service lifetimes** using Laravel's **Service Container**. You'll inject a service into a controller, bind interfaces to concrete implementations, and explore the difference between **transient (default)** and **singleton** bindings.

*⚠ Note: Laravel singleton bindings persist **within the same application lifecycle**. When using `php artisan serve`, each request restarts the Laravel app, so singleton instances are **re-created on every refresh**. To observe real singleton behavior, resolve the service **multiple times within the same request** or use Laravel Octane for persistent lifecycles.*

5.11.2 Objectives

By the end of this lab, you will:

- Understand and apply constructor and method injection
- Bind interfaces to implementations using Laravel's service container
- Use singleton and transient (default) service lifetimes
- View DI output in a simple Blade view

5.11.3 Prerequisites

- Laravel 12 installed via Composer
- PHP ≥ 8.2
- Composer and Visual Studio Code
- Basic understanding of Laravel routes and controllers

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

5.11.4 Steps

Here's a step-by-step guide to creating a simple web app that demonstrates **Dependency Injection** in Laravel 12.

5.11.4.1 Step 1: Create a New Laravel Project

Open a terminal and run:

```
| composer create-project laravel/laravel:^12.0.3 di-demo  
| cd di-demo  
| code .
```

You should see the Laravel project structure in your editor.

5.11.4.2 Step 2: Create a Service Interface and Implementations

In the `app` directory, create a new directory called `Services`. Inside this directory, create three files:

- `MessageServiceInterface.php`
- `StandardMessageService.php`
- `SingletonMessageService.php`

`MessageServiceInterface.php` file contains the interface definition. We will use this interface to bind different implementations.

```
| <?php  
  
| namespace App\Services;  
  
| interface MessageServiceInterface  
{  
|     public function getMessage(): string;  
| }
```

`StandardMessageService.php` file contains the implementation of the `MessageServiceInterface`. This service will be bound as a transient service.

```
| <?php  
  
| namespace App\Services;  
  
| use App\Services\MessageServiceInterface;  
  
| class StandardMessageService implements MessageServiceInterface  
{  
|     public function getMessage(): string  
|     {  
|         return "Hello from StandardMessageService at " . now();  
|     }  
| }
```

`SingletonMessageService.php` file contains another implementation of the `MessageServiceInterface`. This service will be bound as a singleton service.

```
<?php

namespace App\Services;

use App\Services\MessageServiceInterface;

class SingletonMessageService implements MessageServiceInterface
{
    public string $timestamp;

    public function __construct()
    {
        $this->timestamp = now();
    }

    public function getMessage(): string
    {
        return "Hello from SingletonMessageService at {$this->timestamp}";
    }
}
```

Save all three files.

5.11.4.3 Step 3: Register Bindings in AppServiceProvider

Open `app/Providers/AppServiceProvider.php` and add this to the `register()` method:

```
use App\Services\MessageServiceInterface;
use App\Services\StandardMessageService;
use App\Services\SingletonMessageService;

public function register(): void
{
    // Transient (default) binding
    $this->app->bind('standard.message', StandardMessageService::class);

    // Singleton binding
    $this->app->singleton('singleton.message', SingletonMessageService::class);

    // Interface binding (default uses Standard)
    $this->app->bind(MessageServiceInterface::class, StandardMessageService::class);
}
```

5.11.4.4 Step 4: Create a Controller with Different Injection Types

Now we will create a controller that uses both constructor and method injection.

```
| php artisan make:controller MessageController
```

Update `app/Http/Controllers/MessageController.php`:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Services\MessageServiceInterface;

class MessageController extends Controller
{
    protected MessageServiceInterface $service;

    // Constructor Injection
    public function __construct(MessageServiceInterface $service)
    {
        $this->service = $service;
    }

    // Method Injection
    public function show(Request $request)
    {
        $standard = app('standard.message');
        $singleton = app('singleton.message');

        return view('message', [
            'interfaceMsg' => $this->service->getMessage(),
            'standardMsg' => $standard->getMessage(),
            'singletonMsg' => $singleton->getMessage(),
        ]);
    }
}
```

5.11.4.5 Step 5: Define the Route

We will define a route for our controller method. Edit `routes/web.php` and add the following line:

```
use App\Http\Controllers\MessageController;

Route::get('/message', [MessageController::class, 'show']);
```

5.11.4.6 Step 6: Create a Simple View

We will create a simple Blade view to display the messages.

Create the file: `resources/views/message.blade.php`

```
<!DOCTYPE html>
<html>
<head>
    <title>Dependency Injection Demo</title>
</head>
<body>
    <h1>Laravel 12 DI Demo</h1>
```

```
<ul>
    <li><strong>Interface Binding:</strong> {{ $interfaceMsg }}</li>
    <li><strong>Standard Binding (Transient):</strong> {{ $standardMsg }}</li>
    <li><strong>Singleton Binding:</strong> {{ $singletonMsg }}</li>
</ul>
</body>
</html>
```

5.11.4.7 Step 7: Run the App

After completing the above steps, run the app using the built-in server:

```
| php artisan serve
```

Visit in your browser:

- <http://127.0.0.1:8000/message>

Expected Output:

- **StandardMessageService** (transient) should show a different timestamp on every request
- **SingletonMessageService** will also show a new timestamp on every request (since `php artisan serve` restarts the container every time)

5.11.4.8 To simulate actual singleton behavior:

Modify the controller to resolve the singleton **twice in the same request**:

```
public function show(Request $request)
{
    $standard = app('standard.message');
    $singleton = app('singleton.message');

    $first = app('singleton.message')->getMessage();
    $second = app('singleton.message')->getMessage();

    return view('message', [
        'interfaceMsg' => $this->service->getMessage(),
        'standardMsg' => $standard->getMessage(),
        'singletonMsg' => $singleton->getMessage(),
        'first' => $first,
        'second' => $second,
    ]);
}
```

In your Blade view (`resources/views/message.blade.php`):

```
<body>
    <h1>Laravel 12 DI Demo</h1>
```

```

<ul>
    <li><strong>Interface Binding:</strong> {{ $interfaceMsg }}</li>
    <li><strong>Standard Binding (Transient):</strong> {{ $standardMsg }}</li>
    <li><strong>Singleton Binding:</strong> {{ $singletonMsg }}</li>
</ul>

<h2>Singleton Test</h2>
<ul>
    <li>First: {{ $first }}</li>
    <li>Second: {{ $second }}</li>
</ul>
</body>

```

Now you can refresh the page and see that the **first** and **second** values are the same, proving that the singleton service is indeed a singleton within the same request lifecycle.

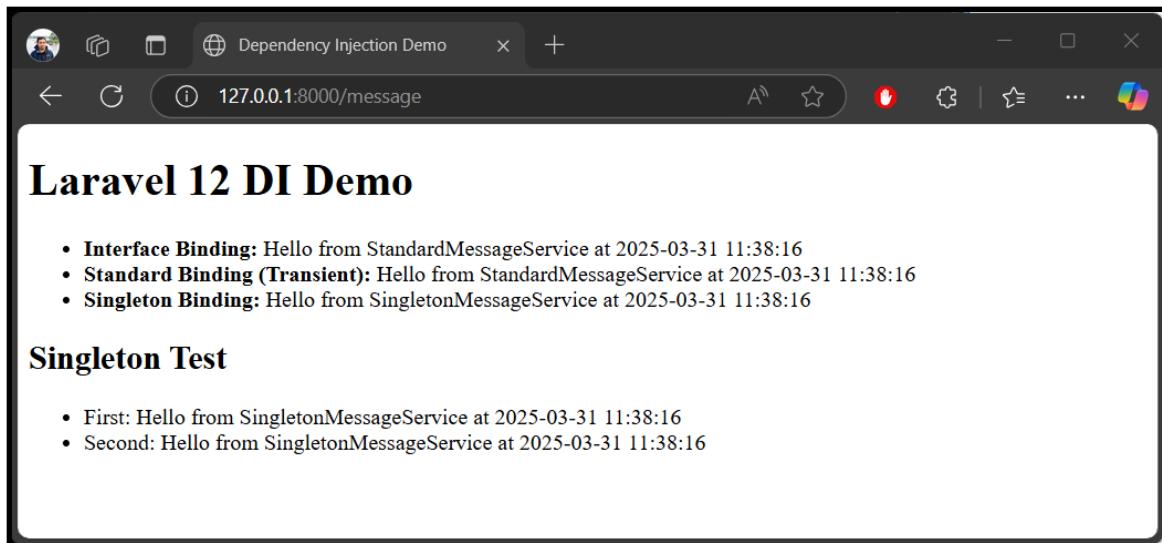


Figure 5.1: Service demo.

You will now see the **same value** for both — proving singleton behavior **within the same request**.

5.11.5 Summary

In this hands-on lab, you:

- Created a service interface and multiple implementations
- Injected dependencies using constructor and method injection
- Registered bindings in the service container
- Demonstrated the concept of **transient vs singleton lifetimes**

*Laravel singletons persist **only for the duration of a single request lifecycle**. In development mode (via `php artisan serve`), the container is reloaded on each request, so singleton behavior is not preserved across refreshes. To observe persistence across requests, consider using **Laravel Octane** or test by resolving the singleton multiple times within the same request.*

5.12 Exercise 8: Contextual Binding with Dependency Injection in Laravel 12

5.12.1 Description

In this lab, we'll create a scenario where different controllers require different implementations of the same interface. We'll use **contextual binding** to tell Laravel's **Service Container** which class to inject based on the controller.

5.12.2 Objectives

By the end of this lab, you will:

- Understand and implement **contextual dependency injection**
- Create an interface with multiple implementations
- Use `when() ->needs() ->give()` in a service provider
- Render injected results in a simple Blade view

5.12.3 Prerequisites

Ensure the following are installed:

- Laravel 12
- PHP \geq 8.2
- Visual Studio Code
- Basic Laravel knowledge (routes, controllers, Blade)

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

5.12.4 Steps

Here are the steps to create a simple web app that demonstrates **contextual binding** in Laravel 12.

5.12.4.1 Step 1: Create and Open Laravel Project

We will create a new Laravel project. Open a terminal and run:

```
| composer create-project laravel/laravel:^12.0.3 di-context  
| cd di-context  
| code .
```

You should see the Laravel project structure in your editor.

5.12.4.2 Step 2: Create an Interface and Implementations

Create these files under `app/Services/`:

- `GreetingServiceInterface.php`
- `FormalGreetingService.php`
- `CasualGreetingService.php`

`GreetingServiceInterface.php` defines the contract for our greeting services.

```
| <?php  
  
| namespace App\Services;  
  
| interface GreetingServiceInterface  
{  
|     public function greet(): string;  
| }
```

`FormalGreetingService.php` implements the `GreetingServiceInterface` and provides a formal greeting.

```
| <?php  
  
| namespace App\Services;  
  
| use App\Services\GreetingServiceInterface;  
  
| class FormalGreetingService implements GreetingServiceInterface  
{  
|     public function greet(): string  
|     {
```

```
    return "Good day, dear guest.";
}
}
```

CasualGreetingService.php implements the GreetingServiceInterface and provides a casual greeting.

```
<?php

namespace App\Services;

use App\Services\GreetingServiceInterface;

class CasualGreetingService implements GreetingServiceInterface
{
    public function greet(): string
    {
        return "Hey there! What's up?";
    }
}
```

5.12.4.3 Step 3: Create Two Controllers

We will create two controllers: GuestController and FriendController.

```
php artisan make:controller GuestController
php artisan make:controller FriendController
```

This will create two files in app/Http/Controllers/:

- GuestController.php
- FriendController.php

Update GuestController.php:

```
namespace App\Http\Controllers;

use App\Services\GreetingServiceInterface;

class GuestController extends Controller
{
    protected GreetingServiceInterface $greetingService;

    public function __construct(GreetingServiceInterface $greetingService)
    {
        $this->greetingService = $greetingService;
    }

    public function greet()
    {
        $message = $this->greetingService->greet();
        return view('greeting', ['message' => $message]);
    }
}
```

```
| } }
```

Update FriendController.php:

```
namespace App\Http\Controllers;

use App\Services\GreetingServiceInterface;

class FriendController extends Controller
{
    protected GreetingServiceInterface $greetingService;

    public function __construct(GreetingServiceInterface $greetingService)
    {
        $this->greetingService = $greetingService;
    }

    public function greet()
    {
        $message = $this->greetingService->greet();
        return view('greeting', ['message' => $message]);
    }
}
```

Save both files.

5.12.4.4 Step 4: Register Contextual Bindings

We need to tell Laravel which implementation to use for each controller.

Open app/Providers/AppServiceProvider.php and edit the register() method:

```
use App\Services\GreetingServiceInterface;
use App\Services\FormalGreetingService;
use App\Services\CasualGreetingService;
use App\Http\Controllers\GuestController;
use App\Http\Controllers\FriendController;

public function register(): void
{
    $this->app->when(GuestController::class)
        ->needs(GreetingServiceInterface::class)
        ->give(FormalGreetingService::class);

    $this->app->when(FriendController::class)
        ->needs(GreetingServiceInterface::class)
        ->give(CasualGreetingService::class);
}
```

5.12.4.5 Step 5: Define Routes

We will define two routes for our controllers. Edit `routes/web.php`:

```
| use App\Http\Controllers\GuestController;
| use App\Http\Controllers\FriendController;

Route::get('/greet/guest', [GuestController::class, 'greet']);
Route::get('/greet/friend', [FriendController::class, 'greet']);
```

5.12.4.6 Step 6: Create a Simple View

Last, create a simple view to display the greeting message.

Create `resources/views/greeting.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Greeting</title>
</head>
<body>
    <h1>{{ $message }}</h1>
</body>
</html>
```

5.12.4.7 Step 7: Run the Laravel App

Now run the app using the built-in server:

```
| php artisan serve
```

Visit in your browser:

- `http://127.0.0.1:8000/greet/guest` → shows formal greeting
- `http://127.0.0.1:8000/greet/friend` → shows casual greeting

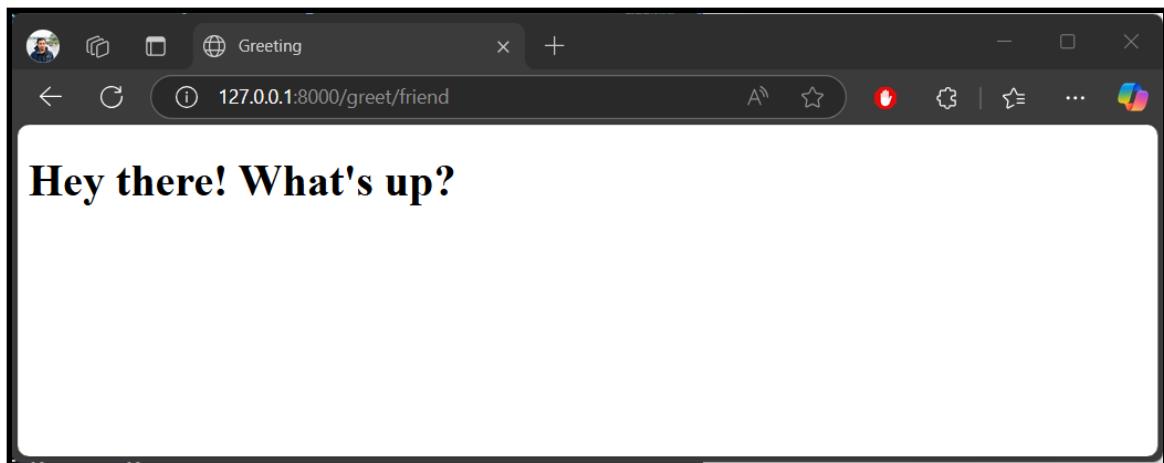


Figure 5.2: Contextual binding demo.

5.12.5 Summary

In this lab, you:

- Defined an interface and two different implementations
- Used **contextual binding** to assign implementations to different controllers
- Injected the service using Laravel’s DI container
- Rendered the result in a simple view

This pattern is especially useful when different parts of the application require different behaviors while adhering to the same contract (interface).

5.13 Exercise 9: Dynamic Service Binding Based on Config or Environment

5.13.1 Description

In this lab, you’ll create a Laravel application that dynamically binds a service implementation based on configuration or environment (e.g., `local` vs `production`). This is useful when your app behaves differently in different environments—without changing code.

5.13.2 Objectives

By the end of this lab, you will:

- Understand how to bind services conditionally using Laravel’s Service Container
- Use Laravel configuration or environment detection (`env()`, `app() ->environment()`)
- Display the selected service behavior using a simple Blade view

5.13.3 Prerequisites

- Laravel 12 installed via Composer

- PHP \geq 8.2
- Composer and Visual Studio Code
- Basic knowledge of controllers, views, and service providers

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

5.13.4 Steps

Here are the steps to create a simple web app that demonstrates **dynamic service binding** in Laravel 12.

5.13.4.1 Step 1: Create and Open a Laravel Project

Firstly, we will create a new Laravel project. Open a terminal and run:

```
| composer create-project laravel/laravel:^12.0.3 di-env-binding  
| cd di-env-binding  
| code .
```

You should see the Laravel project structure in your editor.

5.13.4.2 Step 2: Create an Interface and Two Implementations

Create these under `app/Services/`:

- `NotifierInterface.php`
- `EmailNotifier.php`
- `SlackNotifier.php`

`NotifierInterface.php` defines the contract for our notification services.

```
<?php  
  
namespace App\Services;  
  
interface NotifierInterface  
{  
    public function notify(): string;  
}
```

`EmailNotifier.php` implements the `NotifierInterface` and provides an email notification.

```
<?php

namespace App\Services;

use App\Services\NotifierInterface;

class EmailNotifier implements NotifierInterface
{
    public function notify(): string
    {
        return "Notification sent via Email.";
    }
}
```

`SlackNotifier.php` implements the `NotifierInterface` and provides a Slack notification.

```
<?php

namespace App\Services;

use App\Services\NotifierInterface;

class SlackNotifier implements NotifierInterface
{
    public function notify(): string
    {
        return "Notification sent via Slack.";
    }
}
```

Save all three files.

5.13.4.3 Step 3: Add Configuration Setting

Create a new config file: `config/notifier.php`

```
<?php

return [
    'driver' => env('NOTIFIER_DRIVER', 'email'), // 'email' or 'slack'
];
```

Then add this line to your `.env`:

```
| NOTIFIER_DRIVER=email
```

You can change this to `slack` later for testing.

5.13.4.4 Step 4: Bind the Interface Dynamically in `AppServiceProvider`

Open `app/Providers/AppServiceProvider.php` and update the `register()` method:

```
use App\Services\NotifierInterface;
use App\Services\EmailNotifier;
use App\Services\SlackNotifier;

public function register(): void
{
    $this->app->bind(NotifierInterface::class, function ($app) {
        return match (config('notifier.driver')) {
            'slack' => new SlackNotifier(),
            default => new EmailNotifier(),
        };
    });
}
```

This uses the `notifier.driver` config value to determine which class to bind.

5.13.4.5 Step 5: Create a Controller

Create a controller to handle the notification logic:

```
| php artisan make:controller NotificationController
```

Update the controller:

```
namespace App\Http\Controllers;

use App\Services\NotifierInterface;

class NotificationController extends Controller
{
    public function notify(NotifierInterface $notifier)
    {
        $message = $notifier->notify();
        return view('notify', ['message' => $message]);
    }
}
```

5.13.4.6 Step 6: Define the Route

We will define a route for our controller method. Edit `routes/web.php`:

```
| use App\Http\Controllers\NotificationController;

| Route::get('/notify', [NotificationController::class, 'notify']);
```

5.13.4.7 Step 7: Create a Simple View

Create `resources/views/notify.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Notifier</title>
</head>
<body>
    <h1>{{ $message }}</h1>
</body>
</html>
```

5.13.4.8 Step 8: Run the App and Test

Now run the app using the built-in server:

```
| php artisan serve
```

Visit:

```
| http://127.0.0.1:8000/notify
```

It should show:

```
| Notification sent via Email.
```

Now, change .env:

```
| NOTIFIER_DRIVER=slack
```

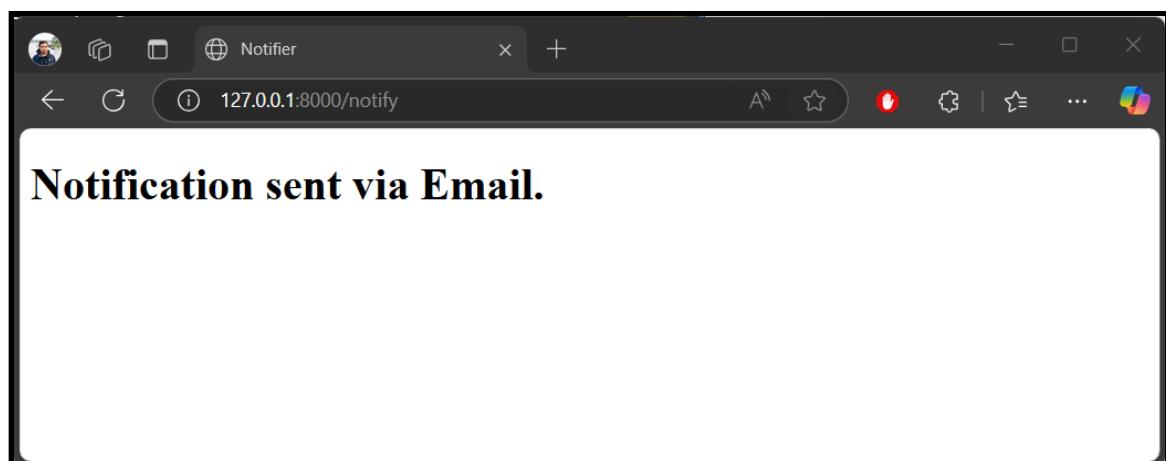


Figure 5.3: Dynamic binding demo.

Then clear config cache:

```
| php artisan config:clear
```

Refresh the browser, and it should show:

| Notification sent via Slack.

5.13.5 Summary

In this lab, you:

- Created an interface and two service implementations
- Used `.env` and `config/` to determine which service to bind
- Used Laravel's service container to bind services dynamically
- Confirmed behavior by switching the environment value

Dynamic binding is a powerful technique that allows Laravel apps to behave differently in staging, production, or development environments—without code changes.

5.14 Conclusion

In this chapter, we explored the concept of **Dependency Injection** in Laravel 12. We learned how to use the **Service Container** to manage class dependencies and perform dependency resolution automatically. We also covered various types of dependency injection, including constructor and method injection, as well as contextual and dynamic binding.

We created hands-on labs to demonstrate these concepts, including creating a simple web app that uses dependency injection to manage service implementations based on configuration or environment. By the end of this chapter, you should have a solid understanding of how to use dependency injection in Laravel 12 and how it can help you write cleaner, more maintainable code.

6 View with Blade View

6.1 Introduction to Blade

Laravel's Blade is a powerful, simple templating engine that allows you to write clean, readable templates using plain PHP with helpful shortcuts.

Key Features:

- Lightweight and fast.
- Includes inheritance and components.
- Includes control structures like `@if`, `@foreach`, `@include`, etc.

6.2 Creating Your First Blade View

We will create a simple Blade view to display a welcome message.

Example:

```
| resources/views/welcome.blade.php
```

Codes for `welcome.blade.php`

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome</title>
</head>
<body>
    <h1>Hello, Laravel!</h1>
</body>
</html>
```

In the controller, we can return this view:

```
| Route::get('/welcome', function () {
    return view('welcome');
});
```

6.3 Passing Data to Views

After creating a view, we can pass data to it from the controller.

We can pass data using the `view()` function:

```
| Route::get('/greeting', function () {
|     return view('greeting', ['name' => 'Agus']);
|});
```

In `resources/views/greeting.blade.php`, we can access the passed data using Blade syntax:

```
| <h1>Hello, {{ $name }}!</h1>
```

6.4 Blade Syntax and Directives

Blade provides a clean syntax for writing PHP code in your views. Here are some common directives:

Here are some examples of Blade syntax:

```
| {{ $name }}
| {!! $html !!}
```

We can use Blade's control structures to handle logic in our views.

```
| @if($user)
|     Hello, {{ $user->name }}
| @else
|     Welcome, Guest!
| @endif
```

We can also use loops:

```
| @foreach($posts as $post)
|     <p>{{ $post->title }}</p>
| @endforeach
```

6.5 Layouts and Sections

Blade allows you to create layouts and sections for better organization of your views. This is useful for creating a consistent look and feel across your application.

For example, we can create a layout file:

```
| resources/views/layouts/app.blade.php
```

Here's a simple layout example:

```
<html>
<head>
    <title>My App - @yield('title')</title>
</head>
<body>
    @yield('content')
</body>
</html>
```

We can create a child view that extends this layout:

```
| resources/views/home.blade.php
```

Here's how to extend the layout:

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

@section('title', 'Home Page')

@section('content')
    <h1>Welcome to the Home Page</h1>
@endsection
```

6.6 Blade Components

Blade components allow you to create reusable pieces of UI. You can create a component using the `make:component` command.

```
| php artisan make:component Alert
```

After creating the component, you can use it in your views, `resources/views/components/alert.blade.php`, like this:

```
<div class="alert alert-danger">
    {{ $slot }}
</div>
```

To use the component in a view, you can do:

```
| <x-alert>
    Something went wrong!
</x-alert>
```

6.7 Blade Includes

Blade allows you to include other Blade views within a view. This is useful for reusing common UI elements like headers, footers, or navigation bars. You can include a partial view using the `@include` directive.

```
| @include('partials.navbar')
```

This will include the `resources/views/partials/navbar.blade.php` file in your view.

```
<nav>
    <a href="/">Home</a>
    <a href="/about">About</a>
</nav>
```

6.8 Blade Loops and Conditionals

Blade provides several directives for loops and conditionals:

- `@isset`
- `@empty`
- `@switch`, `@case`

Example:

```
@switch($user->role)
    @case('admin')
        <p>Welcome admin</p>
        @break

    @default
        <p>Welcome user</p>
@endswitch
```

6.9 Displaying Validation Errors

When using forms, you can display validation errors in your Blade views. Laravel automatically binds the error messages to the `$errors` variable.

You can check for errors using the `any()` method and display them in a list.

```
@if ($errors->any())
    <div>
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

6.10 Best Practices

- Use layouts and components for DRY code.
- Avoid putting business logic in Blade views.
- Use helper functions when necessary.

- Organize views into folders by feature/domain.

6.11 Exercise 10: Passing Data from Controller to Blade View

6.11.1 Description

In this lab, we will create a Laravel 12 web application that demonstrates how to pass various types of data—strings, arrays, associative arrays, and objects—from a controller to a Blade view. We will use **Visual Studio Code** as our main development environment.

6.11.2 Objectives

By the end of this lab, you will be able to:

- Create a route and controller method to pass data.
- Understand how to pass different types of data (string, array, associative array, object) to Blade views.
- Use `{{ }}` Blade syntax to display the data.

6.11.3 Prerequisites

- Laravel 12 installed
- Visual Studio Code installed
- Basic knowledge of PHP and Laravel routing
- A working local Laravel development server

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

6.11.4 Steps

Here are the steps to create a simple Laravel application that passes data from a controller to a Blade view:

6.11.4.1 Step 1: Create a New Laravel Project

If you haven't created one yet:

```
| laravel new blade-data-demo  
| cd blade-data-demo  
| code .
```

Use default Laravel installer to create a new Laravel project. Don't use a starter kit or boilerplate.

You also use `composer create-project` command to create a new Laravel project.

```
| composer create-project laravel/laravel:^12.0.3 blade-data-demo  
| cd blade-data-demo  
| code .
```

laravel new command will create a new Laravel project with the latest version of Laravel. If you want to create a new Laravel project with a specific version, you can use the `composer create-project` command instead.

6.11.4.2 Step 2: Create a Controller

We will create a controller to handle our routes and logic. Run this command to generate a controller:

```
| php artisan make:controller DemoController
```

This will create `app/Http/Controllers/DemoController.php`.

6.11.4.3 Step 3: Define Routes

Open `routes/web.php` and add:

```
| use App\Http\Controllers\DemoController;  
  
| Route::get('/demo', [DemoController::class, 'showData']);
```

6.11.4.4 Step 4: Create the Controller Method

After creating the controller, open `app/Http/Controllers/DemoController.php` and add the following method:

```
| public function showData()  
{
```

```

$name = 'Devi';
$fruits = ['Apple', 'Banana', 'Cherry'];
$user = [
    'name' => 'Eva',
    'email' => 'eva@ilmudata.id',
    'is_active' => true,
];
$product = (object)[
    'id' => 1,
    'name' => 'Laptop',
    'price' => 12000000
];
return view('demo', compact('name', 'fruits', 'user', 'product'));
}

```

6.11.4.5 Step 5: Create a Blade View

Create a new file at `resources/views/demo.blade.php`:

```

<!DOCTYPE html>
<html>
<head>
    <title>Data Passing Demo</title>
</head>
<body>
    <h1>Passing Data to Blade View</h1>

    <h2>String</h2>
    <p>Name: {{ $name }}</p>

    <h2>Array</h2>
    <ul>
        @foreach($fruits as $fruit)
            <li>{{ $fruit }}</li>
        @endforeach
    </ul>

    <h2>Associative Array</h2>
    <p>Name: {{ $user['name'] }}</p>
    <p>Email: {{ $user['email'] }}</p>
    <p>Status: {{ $user['is_active'] ? 'Active' : 'Inactive' }}</p>

    <h2>Object</h2>
    <p>ID: {{ $product->id }}</p>
    <p>Product: {{ $product->name }}</p>
    <p>Price: Rp{{ number_format($product->price, 0, ',', '.') }}</p>
</body>
</html>

```

6.11.4.6 Step 6: Run the App

Start the Laravel development server:

```
| php artisan serve
```

Open your browser and go to:

```
| http://127.0.0.1:8000/demo
```

You should see all the data rendered in the view.

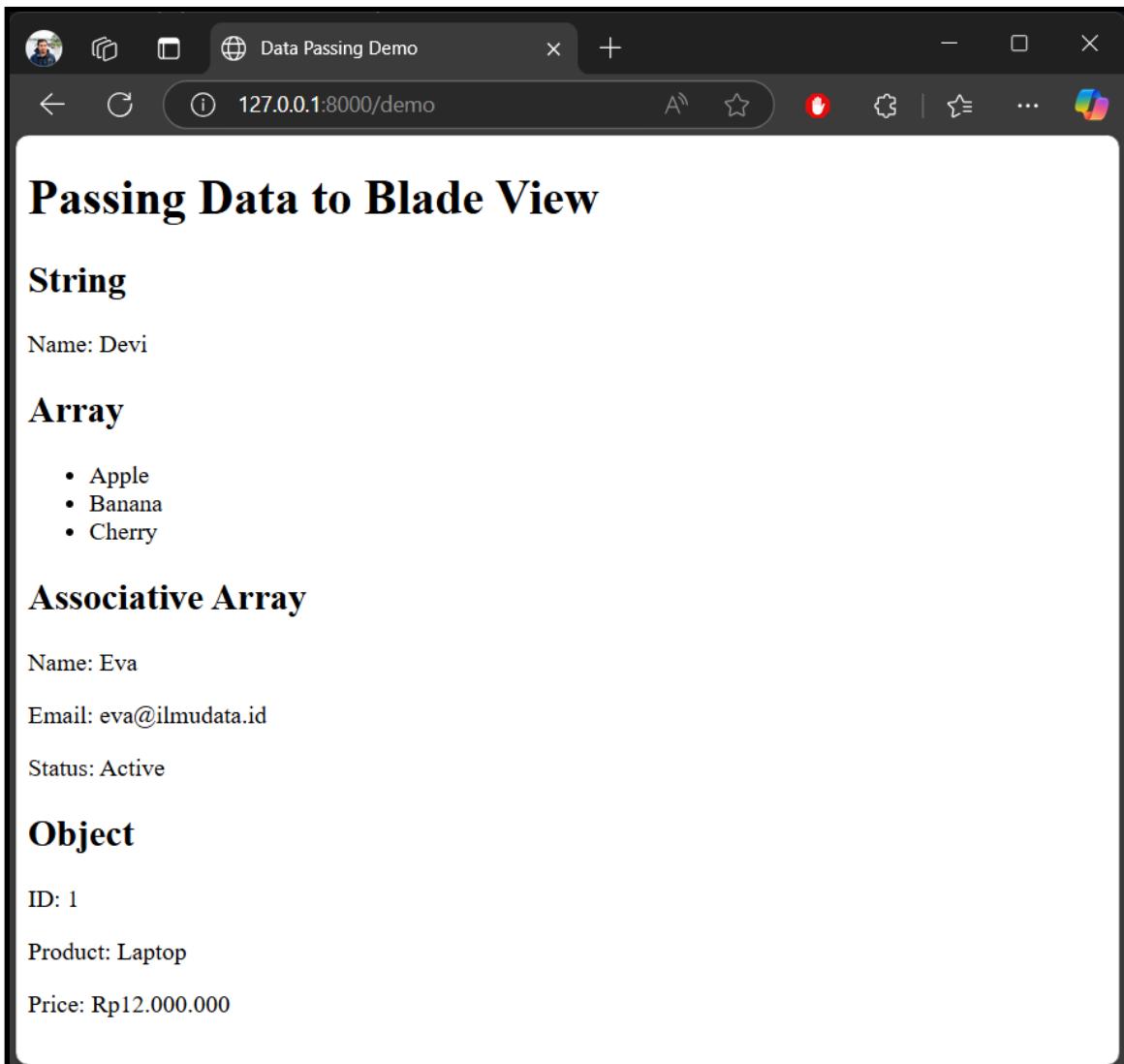


Figure 6.1: Data Passing Demo

6.11.5 Summary

In this lab, you:

- Created a route and controller method.
- Passed various data types—string, array, associative array, and object—to a Blade view.
- Used Blade syntax to dynamically display data in a clean and readable format.

This lab gives you a strong foundation for rendering dynamic data in Laravel applications using Blade.

6.12 Exercise 11: Use Blade Control Structures

6.12.1 Description

In this lab, we will create a Laravel 12 web application that demonstrates how to use Blade control structures such as `@if`, `@elseif`, `@else`, `@foreach`, `@forelse`, `@isset`, `@empty`, and `@switch`. We'll pass various data types—booleans, arrays, and associative arrays—from a controller to a view to drive logic and conditional rendering.

6.12.2 Objectives

By the end of this lab, you will be able to:

- Pass various data types from a controller to a Blade view.
- Use Blade control structures to dynamically render HTML based on data.
- Apply conditional rendering and loops using Blade syntax.

6.12.3 Prerequisites

- Laravel 12 installed
- Visual Studio Code installed
- Basic understanding of Laravel routes and controllers
- A running Laravel development server

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

6.12.4 Steps

Here are the steps to create a simple Laravel application that demonstrates Blade control structures:

6.12.4.1 Step 1: Create a New Laravel Project

If you haven't created the project:

```
| laravel new blade-logic-demo  
| cd blade-logic-demo  
| code .
```

Use default Laravel installer to create a new Laravel project. Don't use a starter kit or boilerplate.

Use `composer create-project` command to create a new Laravel project with specific version.

6.12.4.2 Step 2: Create a Controller

We generate a controller to handle our routes and logic. Run this command to create a controller:

```
| php artisan make:controller LogicController
```

This creates `app/Http/Controllers/LogicController.php`.

6.12.4.3 Step 3: Define a Route

Open `routes/web.php` and add:

```
| use App\Http\Controllers\LogicController;  
|  
| Route::get('/logic', [LogicController::class, 'show']);
```

6.12.4.4 Step 4: Add Logic in the Controller

We already created the controller. Now we will add the logic to the `show` method. This will include passing various data types to the view.

Edit `app/Http/Controllers/LogicController.php`:

```
| public function show()  
{  
    $isLoggedIn = true;  
  
    $users = [  
        ['name' => 'Marcel', 'role' => 'admin'],  
        ['name' => 'Thariq', 'role' => 'editor'],  
        ['name' => 'Ellian', 'role' => 'subscriber'],  
    ];
```

```

$products = []; // Simulating an empty array for @forelse

$profile = [
    'name' => 'Thariq',
    'email' => 'thariq@ilmudata.id'
];

$status = 'active';

return view('logic', compact('isLoggedIn', 'users', 'products', 'profile', 'status'));
}

```

6.12.4.5 Step 5: Create the Blade View

Create the view file at `resources/views/logic.blade.php`:

```

<!DOCTYPE html>
<html>
<head>
    <title>Blade Logic Demo</title>
</head>
<body>
    <h1>Blade Control Structures Demo</h1>

    <h2>1. @@if / @@else</h2>
    @if($isLoggedIn)
        <p>Welcome back, user!</p>
    @else
        <p>Please log in.</p>
    @endif

    <h2>2. @@foreach</h2>
    <ul>
        @foreach($users as $user)
            <li>{{ $user['name'] }} - Role: {{ $user['role'] }}</li>
        @endforeach
    </ul>

    <h2>3. @@forelse</h2>
    @forelse($products as $product)
        <p>{{ $product }}</p>
    @empty
        <p>No products found.</p>
    @endforelse

    <h2>4. @@isset</h2>
    @isset($profile['email'])
        <p>User Email: {{ $profile['email'] }}</p>
    @endisset

    <h2>5. @@empty</h2>
    @empty($profile['phone'])
        <p>No phone number available.</p>
    @endempty

    <h2>6. @@switch</h2>
    @switch($status)
        @case('active')
            <p>Status: Active</p>
        @break

```

```
    @case('inactive')
        <p>Status: Inactive</p>
        @break

    @default
        <p>Status: Unknown</p>
    @endswitch
</body>
</html>
```

6.12.4.6 Step 6: Run the Laravel App

Start the development server:

```
| php artisan serve
```

Access the app at:

```
| http://127.0.0.1:8000/logic
```

You should see a page demonstrating different Blade control structures in action.

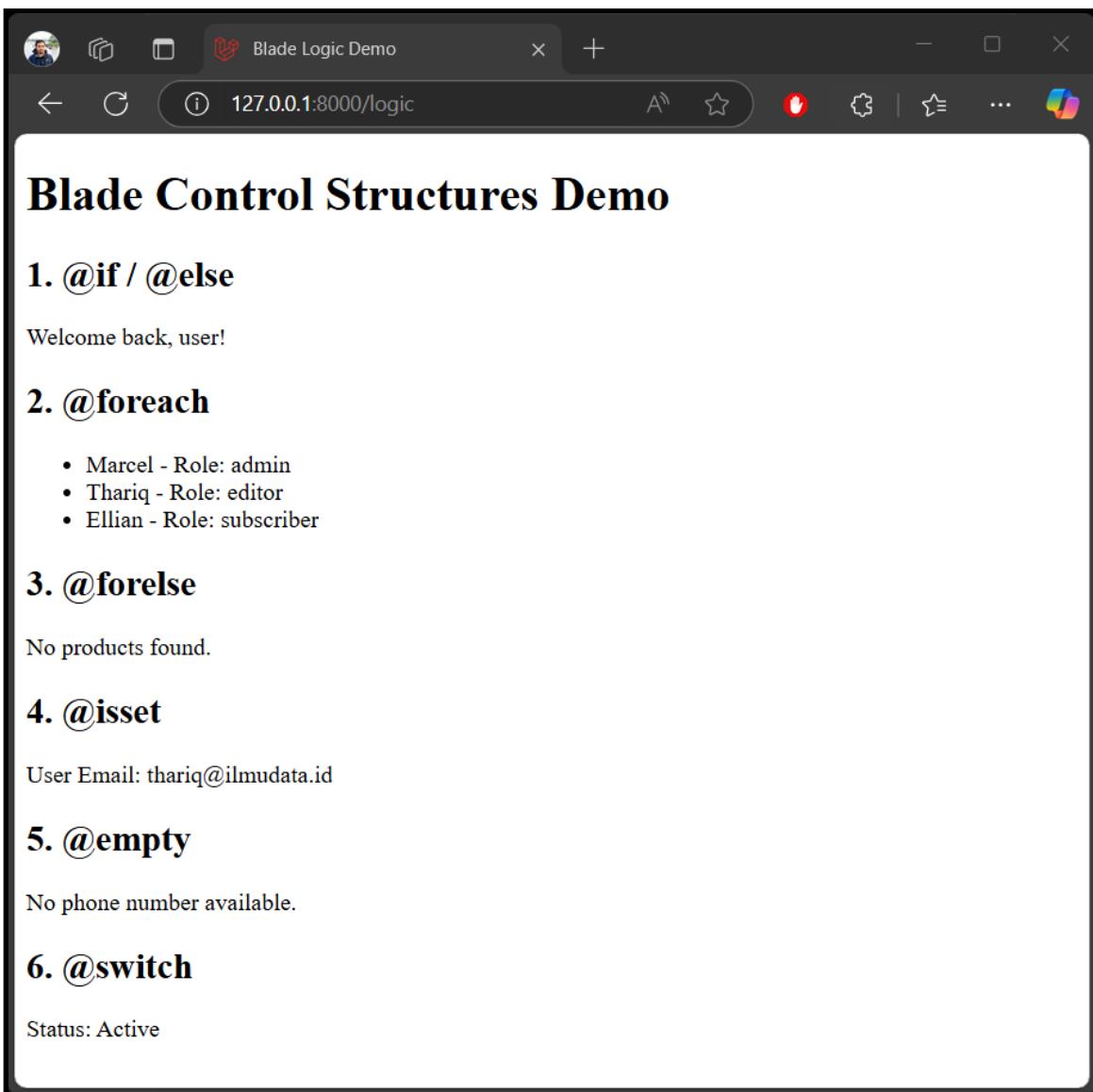


Figure 6.2: Blade Logic Demo

6.12.5 Summary

In this lab, you:

- Created a controller that passed various data types to a view.
- Applied Blade control structures (`@if`, `@foreach`, `@forelse`, `@isset`, `@empty`, and `@switch`) in your Blade file.
- Rendered dynamic content based on conditions and loops.

This exercise gives you a strong foundation for writing conditional logic and loops in Laravel views using Blade.

6.13 Exercise 12: Layout and Personalization in Laravel 12 with Bootstrap

6.13.1 Description

In this lab, you will build a Laravel 12 web application that uses **Blade layouts** and **Bootstrap 5** to create professional-looking pages. You will implement layout reuse and personalize the UI for **admin** and **user** roles using conditionally rendered content.

6.13.2 Objectives

By the end of this lab, you will be able to:

- Create a Bootstrap-based Blade layout.
- Reuse the layout across different views.
- Render personalized admin and user views using passed variables.
- Apply conditional formatting and menus based on user roles.

6.13.3 Prerequisites

- Laravel 12 installed
- Visual Studio Code installed
- Basic Laravel knowledge (routes, controllers, views)
- Internet connection to load Bootstrap via CDN
- Laravel dev server running

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

6.13.4 Steps

Here are the steps to create a simple Laravel application that demonstrates layout and personalization:

6.13.4.1 Step 1: Create Laravel Project

Starting a new Laravel project:

```
| laravel new blade-layout-bootstrap  
| cd blade-layout-bootstrap  
| code .
```

You should see the default Laravel directory structure in Visual Studio Code.

6.13.4.2 Step 2: Create a Controller

We will create a controller to handle our routes and logic:

```
| php artisan make:controller PageController
```

You will find the new controller in `app/Http/Controllers/PageController.php`.

6.13.4.3 Step 3: Add Routes

In `routes/web.php`:

```
| use App\Http\Controllers\PageController;  
  
| Route::get('/admin', [PageController::class, 'admin']);  
| Route::get('/user', [PageController::class, 'user']);
```

This will create two routes: `/admin` and `/user`, which will be handled by the `admin` and `user` methods in the `PageController`, respectively.

6.13.4.4 Step 4: Update Controller

Now we will add the logic to the controller methods. This will include passing user role and username to the views.

In `app/Http/Controllers/PageController.php`:

```
| public function admin()  
{  
|     $role = 'admin';  
|     $username = 'Yamato Admin';  
  
|     return view('admin.dashboard', compact('role', 'username'));  
}  
  
public function user()  
{  
    $role = 'user';  
    $username = 'Liu User';  
  
    return view('user.dashboard', compact('role', 'username'));  
}
```

6.13.4.5 Step 5: Create Base Layout with Bootstrap

Next, we will create a base layout file that uses Bootstrap for styling. This layout will be reused in both admin and user views.

Create `layouts` directory in `resources/views` if it doesn't exist. Then, create `resources/views/layouts/app.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>@yield('title') | Layout and Personalization</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark mb-4">
        <div class="container">
            <a class="navbar-brand" href="#">Layout and Personalization</a>
            <div class="collapse navbar-collapse">
                <ul class="navbar-nav ms-auto">
                    <li class="nav-item">
                        <span class="nav-link active">Welcome, {{ $username }}</span>
                    </li>
                </ul>
            </div>
        </div>
    </nav>

    <div class="container">
        @if($role === 'admin')
            <div class="alert alert-info">Admin Access Granted</div>
        @elseif($role === 'user')
            <div class="alert alert-success">User Area</div>
        @endif

        @yield('content')
    </div>

    <footer class="bg-light text-center mt-5 p-3 border-top">
        <p class="mb-0">&copy; 2025 Layout and Personalization. All rights reserved.</p>
    </footer>

    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js">
</script>
</body>
</html>
```

6.13.4.6 Step 6: Create Admin View

We also need to create a view for the admin dashboard. This will extend the base layout we just created.

Create `admin` directory in `resources/views` if it doesn't exist. Then, create `resources/views/admin/dashboard.blade.php`:

```
| @extends('layouts.app')  
|  
| @section('title', 'Admin Dashboard')  
|  
| @section('content')  
|     <h2 class="mb-4">Admin Dashboard</h2>  
|     <div class="list-group">  
|         <a href="#" class="list-group-item list-group-item-action">Manage Users</a>  
|         <a href="#" class="list-group-item list-group-item-action">Site Settings</a>  
|         <a href="#" class="list-group-item list-group-item-action">System Logs</a>  
|     </div>  
| @endsection
```

Save the file and make sure it is in the correct directory.

6.13.4.7 Step 7: Create User View

Lastly, we will create a view for the user dashboard. This will also extend the base layout.

Create `user` directory in `resources/views` if it doesn't exist. Then, create `resources/views/user/dashboard.blade.php`:

```
| @extends('layouts.app')  
|  
| @section('title', 'User Dashboard')  
|  
| @section('content')  
|     <h2 class="mb-4">User Dashboard</h2>  
|     <p>Welcome to your dashboard, {{ $username }}!</p>  
|     <div class="list-group">  
|         <a href="#" class="list-group-item list-group-item-action">View Profile</a>  
|         <a href="#" class="list-group-item list-group-item-action">Edit Settings</a>  
|         <a href="#" class="list-group-item list-group-item-action">Logout</a>  
|     </div>  
| @endsection
```

Save the file and make sure it is in the correct directory.

6.13.4.8 Step 8: Run the Application

Make sure all steps are completed, then start the Laravel development server:

```
| php artisan serve
```

Open your browser and visit:

- Admin: <http://127.0.0.1:8000/admin>
- User: <http://127.0.0.1:8000/user>

You should see the admin and user dashboards with personalized greetings and role-specific alerts.

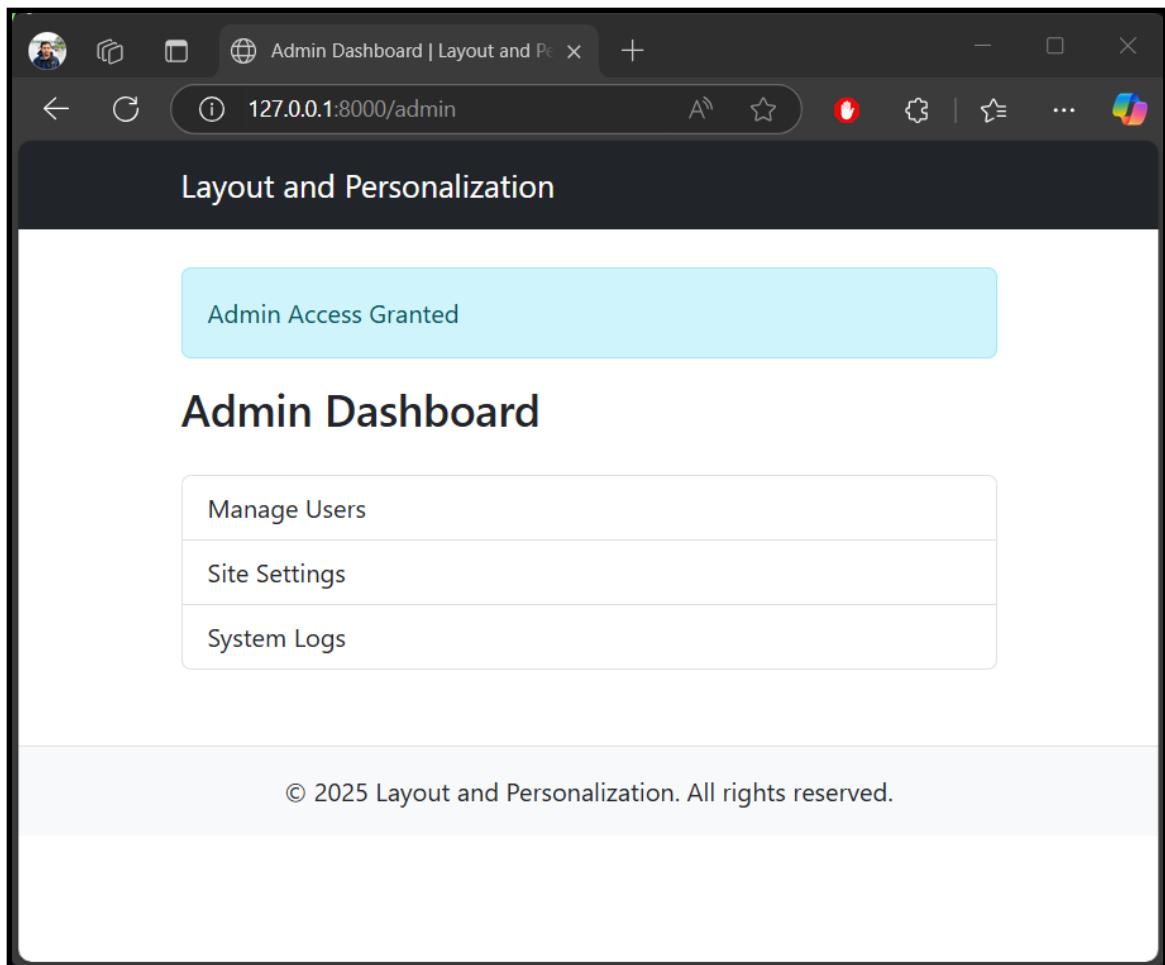


Figure 6.3: Admin Dashboard

Here's the user dashboard:

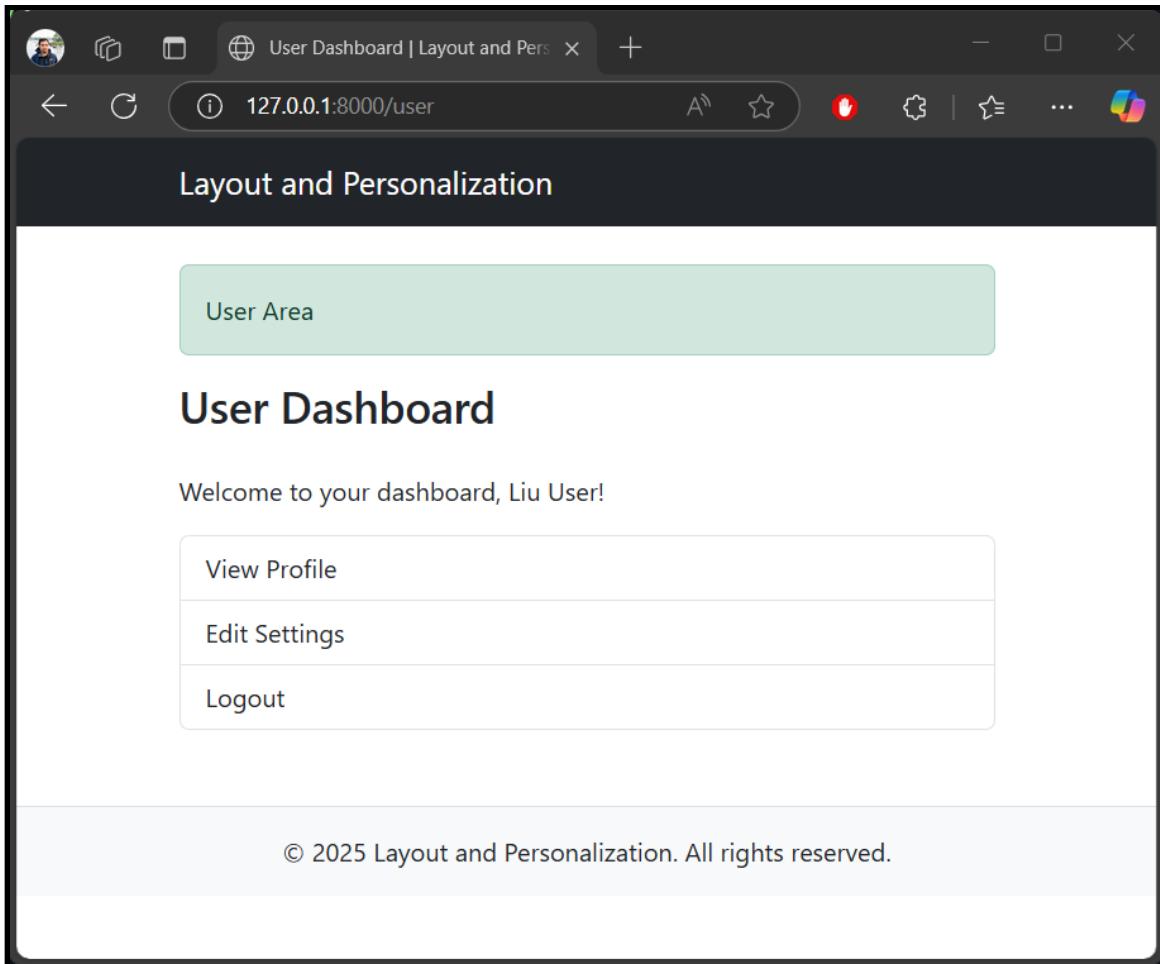


Figure 6.4: User Dashboard

6.13.5 Summary

In this lab, you:

- Created a shared Blade layout using Bootstrap for styling.
- Used `@extends`, `@section`, and `@yield` to manage layout content.
- Personalised the view based on user role with role-specific alerts and menus.
- Improved UI design using Bootstrap components like navbar, alerts, and list groups.

You now have a clean and reusable layout system for your Laravel app that supports both visual consistency and content personalization.

6.14 Exercise 13: Using Partial Views in Laravel 12

6.14.1 Description

In this lab, we will build a Laravel 12 web application that demonstrates the use of **partial views** in Blade. We will create reusable UI parts such as a navigation bar, footer, and alert messages, and include them in multiple views using Blade's `@include` directive. This promotes a clean, modular structure in your Laravel views.

6.14.2 Objectives

By the end of this lab, you will be able to:

- Understand the concept of partial views in Laravel.
- Create reusable Blade partials (e.g., navbar, footer, alerts).
- Use the `@include` directive to insert partial views into layouts or other views.
- Pass data to partial views.

6.14.3 Prerequisites

- Laravel 12 installed (`laravel new blade-partials-demo`)
- Visual Studio Code installed
- Basic knowledge of Laravel controllers, routes, and views
- Laravel development server running (`php artisan serve`)

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

6.14.4 Steps

Here are the steps to create a simple Laravel application that demonstrates the use of partial views:

6.14.4.1 Step 1: Create a New Laravel Project

First, create a new Laravel project:

```
| laravel new blade-partials-demo  
| cd blade-partials-demo  
| code .
```

You should see the default Laravel directory structure in Visual Studio Code.

6.14.4.2 Step 2: Create a Controller

Generate a controller to handle our routes and logic:

```
| php artisan make:controller PageController
```

You will find the new controller in `app/Http/Controllers/PageController.php`. This controller will handle the logic for our views.

6.14.4.3 Step 3: Add Routes

Open `routes/web.php` and define:

```
| use App\Http\Controllers\PageController;  
|  
| Route::get('/home', [PageController::class, 'home']);  
| Route::get('/about', [PageController::class, 'about']);
```

This will create two routes: `/home` (home) and `/about`, which will be handled by the `home` and `about` methods in the `PageController`, respectively.

6.14.4.4 Step 4: Update Controller

Edit `app/Http/Controllers/PageController.php`:

```
| public function home()  
{  
|     $alertMessage = 'Welcome to the homepage!';  
|     return view('home', compact('alertMessage'));  
| }  
  
| public function about()  
{  
|     $alertMessage = 'This is the about page!';  
|     return view('about', compact('alertMessage'));  
| }
```

6.14.4.5 Step 5: Create Base Layout

We will create a base layout file that uses Bootstrap for styling. This layout will be reused in both home and about views.

Create `layouts` directory in `resources/views` if it doesn't exist. Then, create `resources/views/layouts/app.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>@yield('title') | Partial Views</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>

    @include('partials.navbar')

    <div class="container mt-4">
        @include('partials.alert', ['message' => $alertMessage ?? null])

        @yield('content')
    </div>

    @include('partials.footer')

</body>
</html>
```

Save the file and make sure it is in the correct directory.

6.14.4.6 Step 6: Create Partial Views

We will create three partial views: `navbar`, `footer`, and `alert`.

Create a `partials` directory in `resources/views` if it doesn't exist. Then, create the following files:

Create `navbar.blade.php` file in `resources/views/partials`:

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    <div class="container">
        <a class="navbar-brand" href="/home">Partial Views</a>
        <ul class="navbar-nav ms-auto">
            <li class="nav-item"><a class="nav-link" href="/home">Home</a></li>
            <li class="nav-item"><a class="nav-link" href="/about">About</a></li>
        </ul>
    </div>
</nav>
```

This navbar will be included in both the home and about views.

Create `footer.blade.php` file in `resources/views/partials`:

```
<footer class="text-center py-4 border-top mt-5">
    <p class="mb-0">&copy; 2025 Partial Views. All rights reserved.</p>
</footer>
```

This footer will also be included in both views.

Create `alert.blade.php` file in `resources/views/partials`:

```
| @if(!empty($message))
|     <div class="alert alert-info">
|         {{ $message }}
|     </div>
| @endif
```

This alert will display a message if it is passed from the controller.

Save all the files and make sure they are in the correct directory.

6.14.4.7 Step 7: Create Main Views

We will create two main views: `home` and `about`. Create the following files:

- `home.blade.php` in `resources/views`
- `about.blade.php` in `resources/views`

Create `resources/views/home.blade.php`:

```
| @extends('layouts.app')
|
| @section('title', 'Home')
|
| @section('content')
|     <h1 class="mb-3">Home Page</h1>
|     <p>This is the home page content.</p>
| @endsection
```

This home view will extend the base layout and include the navbar, footer, and alert.

Create `resources/views/about.blade.php`:

```
| @extends('layouts.app')
|
| @section('title', 'About')
|
| @section('content')
|     <h1 class="mb-3">About Us</h1>
|     <p>This is the about page content.</p>
| @endsection
```

This about view will also extend the base layout and include the navbar, footer, and alert.

Save all the files and make sure they are in the correct directory.

6.14.4.8 Step 8: Run and Test

Start the development server:

```
| php artisan serve
```

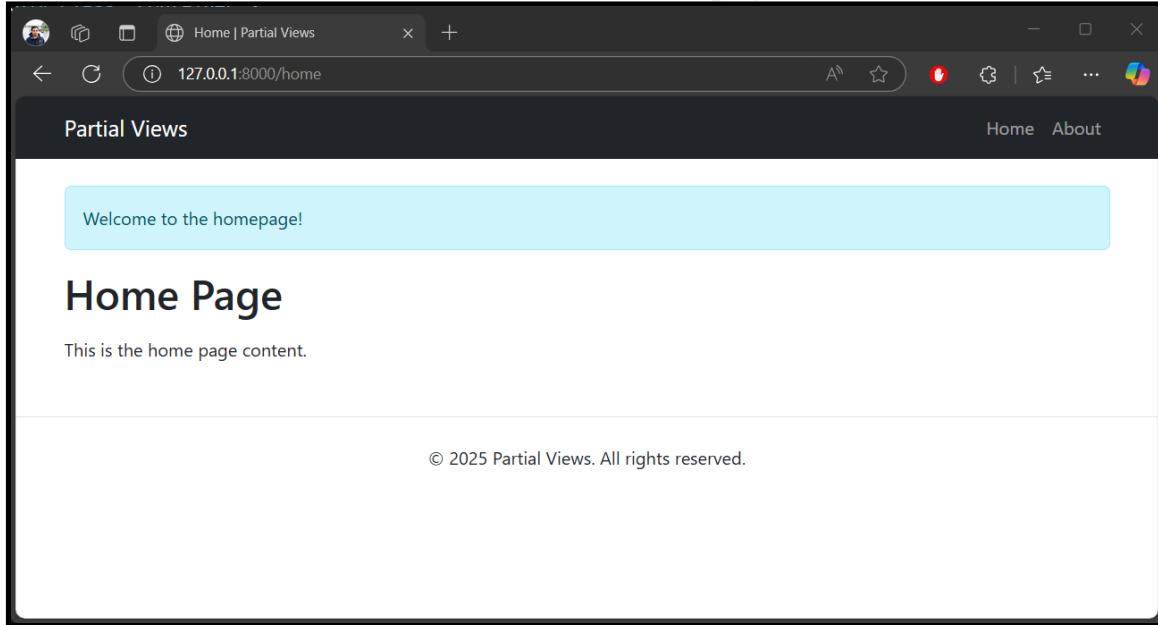
Open your browser and test:

- Home: <http://127.0.0.1:8000/home>
- About: <http://127.0.0.1:8000/about>

You should see:

- A **shared navbar** at the top.
- A **shared footer** at the bottom.
- An **alert box** displaying a custom message on each page.

Here are the screenshots of the home and about pages:



Here's the about page:

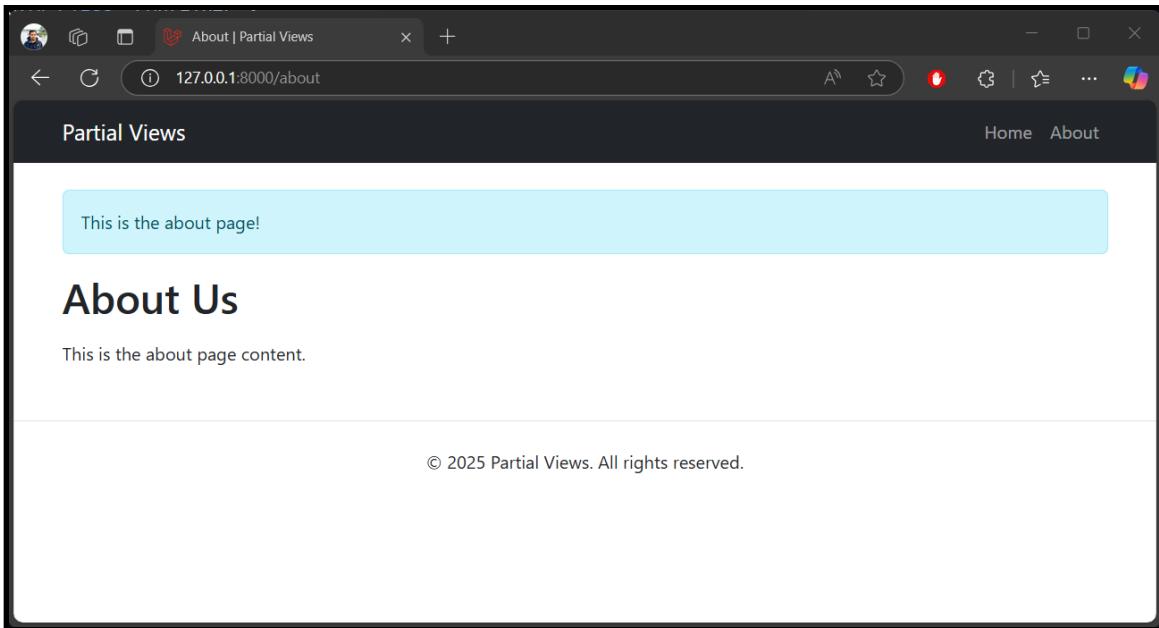


Figure 6.6: About Page

6.14.5 Summary

In this lab, you:

- Created reusable Blade **partial views** (`navbar`, `footer`, `alert`).
- Used `@include` to embed those partials in your layout.
- Passed dynamic data (alert message) to the alert partial.
- Created two views (Home and About) that share the same layout and partials.

This approach helps keep your views modular, DRY (Don't Repeat Yourself), and easier to maintain.

6.15 Exercise 14: Using Blade Components in Laravel 12

6.15.1 Description

In this lab, you'll build a Laravel 12 web application that uses **Blade components** to create reusable and modular UI elements. Blade components allow you to encapsulate parts of your layout, such as alert boxes, buttons, or cards, and use them across different pages or layouts with clean syntax.

6.15.2 Objectives

By the end of this lab, you will be able to:

- Create class-based and anonymous Blade components.
- Use `<x-component-name />` syntax in views.
- Pass dynamic content and attributes to components.
- Integrate components into layouts for consistent UI.

6.15.3 Prerequisites

- Laravel 12 installed
- Visual Studio Code installed
- Basic Laravel knowledge (routes, controllers, views)
- Laravel dev server running

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

6.15.4 Steps

Here are the steps to create a simple Laravel application that demonstrates the use of Blade components:

6.15.4.1 Step 1: Create Laravel Project

We will start a new Laravel project:

```
| laravel new blade-component-demo  
| cd blade-component-demo  
| code .
```

You should see the default Laravel directory structure in Visual Studio Code.

6.15.4.2 Step 2: Create a Controller

Next, we will create a controller to handle our routes and logic:

```
| php artisan make:controller PageController
```

This will create `app/Http/Controllers/PageController.php`.

6.15.4.3 Step 3: Define Routes

We will define two routes: `/home` (home) and `/contact`, which will be handled by the `home` and `contact` methods in the `PageController`, respectively.

Open `routes/web.php` and add:

```
use App\Http\Controllers\PageController;

Route::get('/home', [PageController::class, 'home']);
Route::get('/contact', [PageController::class, 'contact']);
```

6.15.4.4 Step 4: Add Controller Logic

In `app/Http/Controllers/PageController.php`:

```
public function home()
{
    return view('home');
}

public function contact()
{
    return view('contact');
```

6.15.4.5 Step 5: Create a Layout View

Create a layout file that will be used across different views. This layout will include the navbar and footer components.

Create `resources/views/layouts` directory if it doesn't exist. Then, create `resources/views/layouts/app.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>@yield('title') | Blade Component</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>

    <x-navbar />

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

    <x-footer />
```

```
| </body>
| </html>
```

Save the file and make sure it is in the correct directory.

6.15.4.6 Step 6: Create Blade Components

We will create three Blade components: `Navbar`, `Footer`, and `Alert`.

Create a new component for a navbar component using the Artisan command:

```
| php artisan make:component Navbar
```

This will create a new component class and view file for the navbar. The class will be located at `app/View/Components/Navbar.php` and the view at `resources/views/components/navbar.blade.php`.

Edit the view: `resources/views/components/navbar.blade.php`

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    <div class="container">
        <a class="navbar-brand" href="/home">Blade Component</a>
        <ul class="navbar-nav ms-auto">
            <li class="nav-item"><a class="nav-link" href="/home">Home</a></li>
            <li class="nav-item"><a class="nav-link" href="/contact">Contact</a></li>
        </ul>
    </div>
</nav>
```

Create a new component for a footer component using the Artisan command:

```
| php artisan make:component Footer
```

Edit the view: `resources/views/components/footer.blade.php`

```
<footer class="text-center py-4 border-top mt-5">
    <p class="mb-0">&copy; 2025 Blade Component. All rights reserved.</p>
</footer>
```

Create a new component for an alert box using the Artisan command:

```
| php artisan make:component Alert
```

Edit the view: `resources/views/components/alert.blade.php`

```
<div class="alert alert-{{ $type ?? 'info' }}">
    {{ $slot }}
</div>
```

Save the file and make sure it is in the correct directory.

6.15.4.7 Step 7: Create Views That Use Components

We will create two views: `home` and `contact`. These views will extend the layout and use the components. Create `resources/views/home.blade.php`:

```
| @extends('layouts.app')  
|  
| @section('title', 'Home')  
|  
| @section('content')  
|   <h1>Home Page</h1>  
|  
|   <x-alert type="success">  
|     Welcome to the homepage!  
|   </x-alert>  
| @endsection
```

Create `resources/views/contact.blade.php`:

```
| @extends('layouts.app')  
|  
| @section('title', 'Contact')  
|  
| @section('content')  
|   <h1>Contact Page</h1>  
|  
|   <x-alert type="warning">  
|     This is the contact page.  
|   </x-alert>  
| @endsection
```

Save the files and make sure they are in the correct directory.

6.15.4.8 Step 8: Run and Test

Make sure you already completed all steps, then start the Laravel development server:

```
| php artisan serve
```

Open your browser:

- Home: <http://127.0.0.1:8000/home>
- Contact: <http://127.0.0.1:8000/contact>

You should see:

- A reusable **navbar** and **footer** via components.
- An **alert box** with dynamic types and messages passed via props and slots.

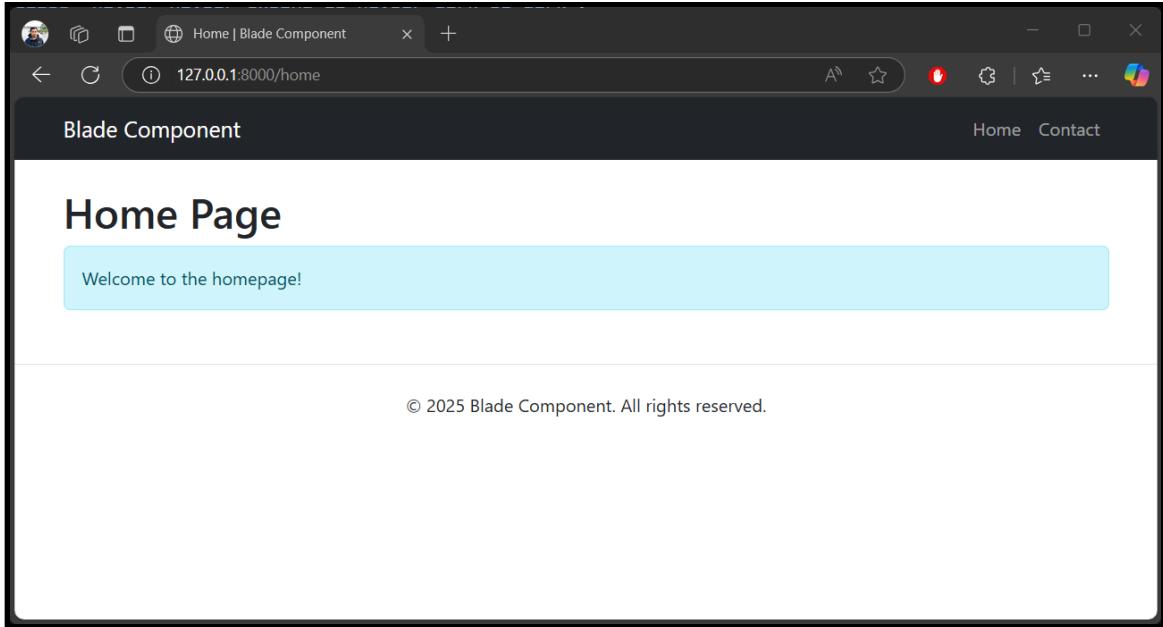


Figure 6.7: Home Page with Alert

Here's the contact page:

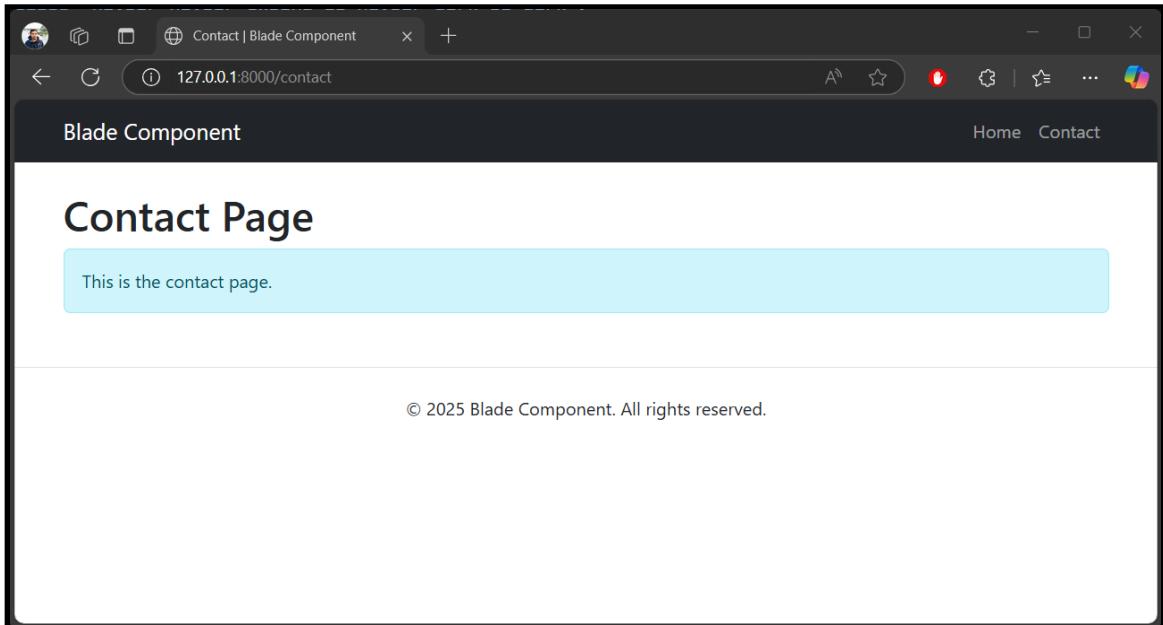


Figure 6.8: Contact Page with Alert

6.15.5 Summary

In this lab, you:

- Created reusable Blade components (`Navbar`, `Footer`, `Alert`).

- Used `<x-component-name />` syntax to inject components in views and layouts.
- Passed props and content using attributes and slots.
- Simplified your code and followed DRY principles using component-based design.

Blade components make your Laravel applications more maintainable and consistent—especially as the UI grows.

6.16 Exercise 15: Implementing Light and Dark Themes in Laravel 12

6.16.1 Description

In this lab, you'll build a Laravel 12 web application that supports both **light and dark themes**. You'll allow users to toggle between themes and apply the selected theme globally using Laravel sessions and Blade templates. The layout will adapt styles accordingly.

6.16.2 Objectives

By the end of this lab, you will be able to:

- Create a layout with theme switching logic.
- Store the user's theme preference using Laravel session.
- Switch between light and dark modes using a button or link.
- Dynamically load CSS classes based on the selected theme.

6.16.3 Prerequisites

- Laravel 12 installed
- Visual Studio Code installed
- Basic Laravel knowledge (routes, controllers, views, sessions)
- Laravel dev server running

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

6.16.4 Steps

Here are the steps to create a simple Laravel application that demonstrates light and dark themes:

6.16.4.1 Step 1: Create a New Laravel Project

We will start a new Laravel project:

```
| laravel new blade-theme-demo  
| cd blade-theme-demo  
| code .
```

You should see the default Laravel directory structure in Visual Studio Code.

6.16.4.2 Step 2: Create a Controller

We will create a controller to handle our routes and logic:

```
| php artisan make:controller ThemeController
```

This will create `app/Http/Controllers/ThemeController.php`.

6.16.4.3 Step 3: Define Routes

We will define two routes: `/home` (home) and `/switch-theme/{theme}`, which will be handled by the `home` and `switchTheme` methods in the `ThemeController`, respectively.

Open `routes/web.php`:

```
| use App\Http\Controllers\ThemeController;  
  
| Route::get('/home', [ThemeController::class, 'home']);  
| Route::get('/switch-theme/{theme}', [ThemeController::class, 'switchTheme'])->name('switch-
```

6.16.4.4 Step 4: Update the Controller

We will add the logic to the controller methods. This will include passing the selected theme to the view and switching themes based on user input.

Open `app/Http/Controllers/ThemeController.php`:

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

```

class ThemeController extends Controller
{
    public function home(Request $request)
    {
        $theme = session('theme', 'light');
        return view('home', compact('theme'));
    }

    public function switchTheme($theme, Request $request)
    {
        if (in_array($theme, ['light', 'dark'])) {
            session(['theme' => $theme]);
        }

        return redirect('/home');
    }
}

```

6.16.4.5 Step 5: Create Layout with Theme Support

We will create a base layout file that uses Bootstrap for styling. This layout will be reused in both home and about views.

Create `layouts` directory in `resources/views` if it doesn't exist. Then, create `resources/views/layouts/app.blade.php`:

```

<!DOCTYPE html>
<html lang="en" data-bs-theme="{{ $theme }}>
<head>
    <meta charset="UTF-8">
    <title>@yield('title', 'Theme Demo')</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
    <style>
        body {
            padding-top: 4rem;
        }
    </style>
</head>
<body class="{{ $theme === 'dark' ? 'bg-dark text-light' : 'bg-light text-dark' }}>

<nav class="navbar navbar-expand-lg {{ $theme === 'dark' ? 'navbar-dark bg-dark' : 'navbar-light bg-light' }}>
    <div class="container">
        <a class="navbar-brand" href="#">ThemeApp</a>
        <ul class="navbar-nav ms-auto">
            <li class="nav-item">
                <a href="{{ route('switch-theme', 'light') }}" class="nav-link">Light</a>
            </li>
            <li class="nav-item">
                <a href="{{ route('switch-theme', 'dark') }}" class="nav-link">Dark</a>
            </li>
        </ul>
    </div>
</nav>

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

```

```
| </div>
| </body>
| </html>
```

6.16.4.6 Step 6: Create Main View

Now we will create the main view that will extend the layout and display the current theme.

Create `resources/views/home.blade.php`:

```
| @extends('layouts.app')
| 
| @section('title', 'Home')
|
| @section('content')
|   <div class="p-4 border rounded">
|     <h1>Welcome to the Theme Demo!</h1>
|     <p>Current Theme: <strong>{{ ucfirst($theme) }}</strong></p>
|     <p>Use the navigation links to switch themes.</p>
|   </div>
| @endsection
```

Save the file and make sure it is in the correct directory.

6.16.4.7 Step 7: Run the Application

After completing all steps, start the Laravel development server:

```
| php artisan serve
```

Open your browser and visit:

- Home: <http://127.0.0.1:8000/home>

Click on **Light** or **Dark** in the navbar to switch themes. The selected theme is stored in the session and affects the entire layout.

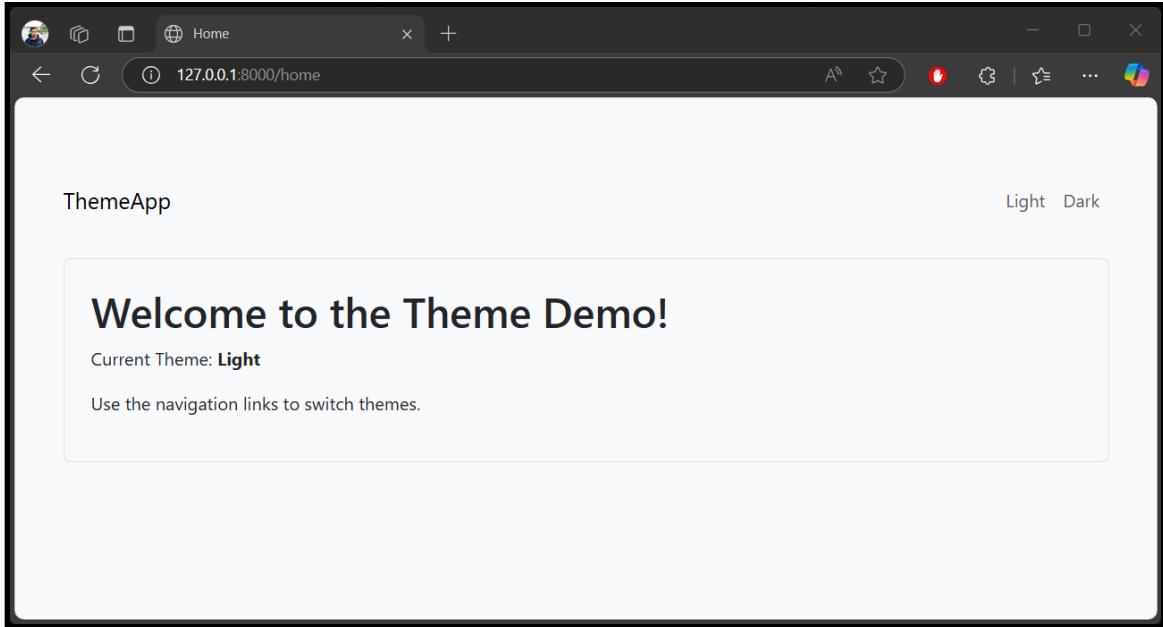


Figure 6.9: Light Theme.

Here's the dark theme:

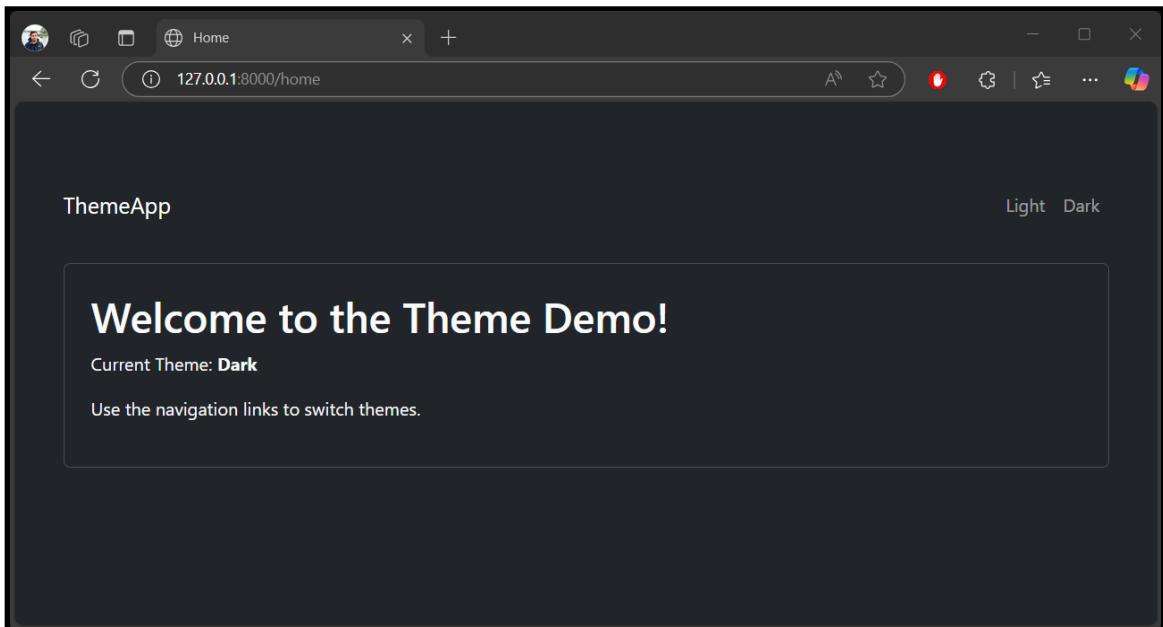


Figure 6.10: Dark Theme.

6.16.5 Summary

In this hands-on lab, you:

- Created a Blade layout that adapts to light and dark modes using Bootstrap and Blade logic.
- Used Laravel sessions to store the user's theme preference.
- Provided a simple navigation UI to toggle between light and dark themes.
- Dynamically adjusted CSS classes and Bootstrap themes.

This is a practical way to add user-friendly customization to Laravel apps with minimal effort.

6.17 Conclusion

In this chapter, we explored the powerful features of Blade templating in Laravel. We covered various aspects of Blade, including control structures, layouts, partial views, components, and theming. By the end of this chapter, you should have a solid understanding of how to create dynamic and reusable views in your Laravel applications.

We also learned how to implement light and dark themes, enhancing the user experience by allowing users to choose their preferred interface. This chapter provided hands-on exercises to reinforce your understanding of Blade templating and its capabilities.

As you continue your journey with Laravel, remember that Blade is a powerful tool that can help you create clean, maintainable, and dynamic views. The skills you've learned in this chapter will serve as a strong foundation for building more complex applications in the future.

OceanofPDF.com

7 Form Submission and Data Validation

7.1 Introduction

Form submission is a fundamental aspect of web development, allowing users to interact with your application. Laravel 12 provides powerful tools to handle form inputs, validate data, and respond appropriately.

7.2 Creating a Basic HTML Form

Let's build a simple form to add a new product. We will create a form with fields for the product name and price.

Blade View: resources/views/products/create.blade.php

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

@section('content')
<h1>Add New Product</h1>
<form method="POST" action="{{ route('products.store') }}">
    @csrf
    <div>
        <label>Name:</label>
        <input type="text" name="name" value="{{ old('name') }}">
        @error('name')
            <div>{{ $message }}</div>
        @enderror
    </div>
    <div>
        <label>Price:</label>
        <input type="number" name="price" value="{{ old('price') }}">
        @error('price')
            <div>{{ $message }}</div>
        @enderror
    </div>
    <button type="submit">Save</button>
</form>
@endsection
```

This form includes CSRF protection and displays validation errors if any.

7.3 Handling Form Submission in Controller

To process the form submission, we need to create a controller method that handles the incoming request.

Here's how to handle the form submission and validate the input,

app/Http/Controllers/ProductController.php:

```
public function store(Request $request)
{
    $validated = $request->validate([
        'name' => 'required|string|max:255',
        'price' => 'required|numeric|min:0',
    ]);

    Product::create($validated);

    return redirect()->route('products.index')->with('success', 'Product added successfully');
}
```

7.4 Understanding CSRF Protection

Laravel automatically protects forms from cross-site request forgery using the `@csrf` directive. This ensures only forms originating from your app are processed.

7.5 Displaying Validation Errors

Errors are automatically shared with views when using `validate()`. Use the `@error` directive or `session('errors')` for more custom handling.

We can create a Form Request class to handle validation logic separately.

7.6 Exercise 16: Handle Form Submission with Validation in Laravel 12

7.6.1 Description

In this lab, we will build a Laravel 12 web application that includes a form with various input types. The form will validate the inputs before submission, enable the submit button only if a checkbox is checked, and then redirect to a page showing the submitted data. We'll also use Bootstrap for styling the form.

7.6.2 Objectives

By the end of this lab, you will be able to:

- Create a form with different input types using Blade and Bootstrap.
- Implement server-side validation using Laravel's validation rules.
- Enable the submit button only when the confirmation checkbox is checked using JavaScript.
- Redirect to a different route to display the submitted data.

7.6.3 Prerequisites

- PHP 8.1+ and Composer installed
- Laravel 12 installed
- Basic understanding of Laravel routes, controllers, and views
- Visual Studio Code or any code editor
- Bootstrap 5 (via CDN)

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

7.6.4 Steps

Here's a step-by-step guide to creating the form submission system:

7.6.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project using the Laravel installer. If you don't have it installed, you can use Composer to create a new project.

```
| laravel new form-app  
| cd form-app  
| code .
```

You should have a new Laravel project set up. Now, let's create the necessary routes and views.

7.6.4.2 Step 2: Set Up the Routes

We will define the routes for displaying the form, handling the form submission, and showing the result.

Edit routes/web.php:

```
| use App\Http\Controllers\FormController;  
|  
| Route::get('/form', [FormController::class, 'showForm'])->name('form.show');  
| Route::post('/form', [FormController::class, 'handleForm'])->name('form.handle');  
| Route::get('/result', [FormController::class, 'showResult'])->name('form.result');
```

Save the file.

7.6.4.3 Step 3: Create the Controller

Create a controller to handle the form logic. Run the following command in your terminal:

```
| php artisan make:controller FormController
```

This will create a new controller file in app/Http/Controllers/FormController.php. Open this file and add the following methods:

```
| namespace App\Http\Controllers;  
|  
| use Illuminate\Http\Request;  
|  
| class FormController extends Controller  
{  
|     public function showForm()  
|     {  
|         return view('form');  
|     }  
|  
|     public function handleForm(Request $request)  
|     {  
|         $validated = $request->validate([  
|             'name' => 'required|string|max:255',  
|             'email' => 'required|email',  
|             'age' => 'required|integer|min:1',  
|             'password' => 'required|min:6',  
|             'gender' => 'required',  
|             'role' => 'required',  
|             'bio' => 'required',  
|             'confirm' => 'accepted',  
|         ]);  
|  
|         return redirect()->route('form.result')->with('data', $validated);  
|     }  
|  
|     public function showResult()  
|     {  
|         $data = session('data');  
|         return view('result', compact('data'));  
|     }  
| }
```

7.6.4.4 Step 4: Create the Form View

Create a new Blade view file for the form. Create a directory named `form` inside `resources/views` and create a file named `form.blade.php` inside it.

```
<!DOCTYPE html>
<html>
<head>
    <title>Form Submission</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container mt-5">

<h2>Registration Form</h2>
<form method="POST" action="{{ route('form.handle') }}">
    @csrf

    <div class="mb-3">
        <label>Name</label>
        <input type="text" name="name" class="form-control" value="{{ old('name') }}"/>
        @error('name') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label>Email</label>
        <input type="email" name="email" class="form-control" value="{{ old('email') }}"/>
        @error('email') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label>Age</label>
        <input type="number" name="age" class="form-control" value="{{ old('age') }}"/>
        @error('age') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label>Password</label>
        <input type="password" name="password" class="form-control"/>
        @error('password') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label>Gender</label><br>
        <div class="form-check form-check-inline">
            <input class="form-check-input" type="radio" name="gender" value="male"> Male
        </div>
        <div class="form-check form-check-inline">
            <input class="form-check-input" type="radio" name="gender" value="female">
        </div>
        @error('gender') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label>Role</label>
        <select name="role" class="form-select">
            <option value="">-- Select Role --</option>
            <option value="user">User</option>
            <option value="admin">Admin</option>
        </select>
        @error('role') <div class="text-danger">{{ $message }}</div> @enderror
    </div>
```

```

<div class="mb-3">
    <label>Bio</label>
    <textarea name="bio" class="form-control">{{ old('bio') }}</textarea>
    @error('bio') <div class="text-danger">{{ $message }}</div> @enderror
</div>

<div class="form-check mb-3">
    <input type="checkbox" id="confirm" name="confirm" class="form-check-input">
    <label class="form-check-label" for="confirm">I confirm the information is
correct</label>
    @error('confirm') <div class="text-danger">{{ $message }}</div> @enderror
</div>

    <button type="submit" class="btn btn-primary" id="submitBtn" disabled>Submit</button>
</form>

<script>
    document.getElementById('confirm').addEventListener('change', function () {
        document.getElementById('submitBtn').disabled = !this.checked;
    });
</script>

</body>
</html>

```

7.6.4.5 Step 5: Create the Result View

Create a new Blade view file for displaying the result. Create a file named `result.blade.php` **inside the** `resources/views` directory.

```

<!DOCTYPE html>
<html>
<head>
    <title>Form Result</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container mt-5">
    <h2>Form Submitted Data</h2>
    @if ($data)
        <ul class="list-group">
            @foreach ($data as $key => $value)
                <li class="list-group-item"><strong>{{ ucfirst($key) }}:</strong> {{
$value }}</li>
            @endforeach
        </ul>
    @else
        <p>No data available.</p>
    @endif
</body>
</html>

```

7.6.4.6 Step 6: Run the Application

After completing the above steps, run the Laravel development server:

```
| php artisan serve
```

Visit <http://localhost:8000/form> and test the form.

The screenshot shows a web browser window with a registration form titled "Registration Form". The form fields are as follows:

- Name: Tester 1
- Email: tester1@email.com
- Age: 35
- Password: *****
- Gender: Male (selected)
- Role: Admin
- Bio: Tester is
- A checkbox labeled "I confirm the information is correct" is checked.

At the bottom of the form is a blue "Submit" button.

Figure 7.1: Form Submission.

After filling in the form, click the submit button. You should be redirected to the result page displaying the submitted data.

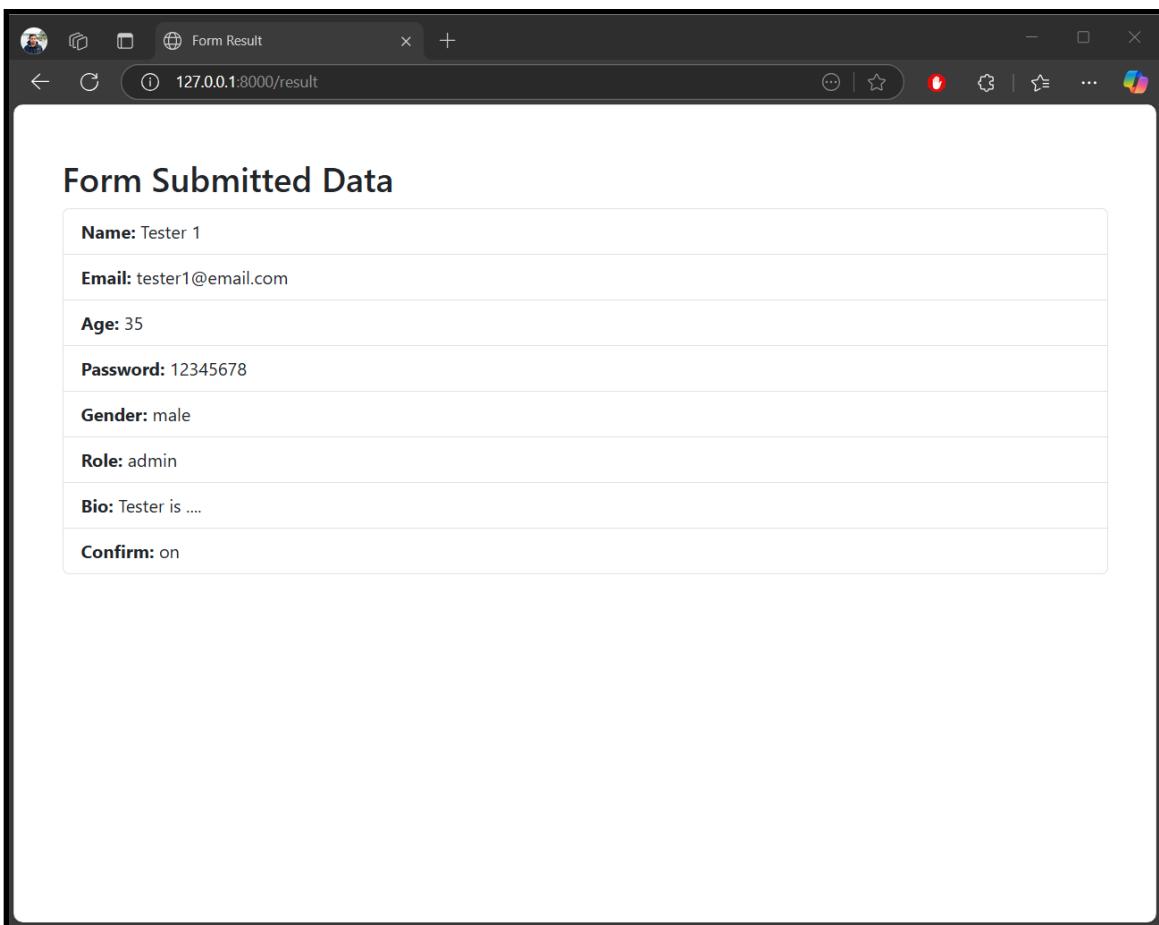


Figure 7.2: Form Result.

You may try to submit the form without filling in the required fields to see the validation errors. Also, try submitting the form without checking the confirmation checkbox to see that the submit button is disabled until checked.

The screenshot shows a registration form titled "Registration Form" on a web page at "127.0.0.1:8000/form". The form fields and their current values are:

- Name: Tester 1
- Email: tester1@email.com
- Age: 35
- Password: (empty field)
- Gender: Male (radio button selected)
- Role: -- Select Role -- (dropdown menu)
- Bio: (empty text area)
- I confirm the information is correct: (unchecked checkbox)

Validation errors are displayed in red text next to the fields:

- The password field is required.
- The gender field is required.
- The role field is required.
- The bio field is required.

Figure 7.3: Validation Error.

7.6.5 Summary

In this lab, we created a full-featured form submission system using Laravel 12 and Bootstrap. You learned how to:

- Use various form input types (text, email, number, password, select, radio, textarea).
- Perform server-side validation.
- Enable form submission only when a checkbox is checked using JavaScript.
- Redirect and display submitted form data on another page.

7.7 Exercise 17: Custom Validation Rules and Messages in Laravel 12

7.7.1 Description

This hands-on lab walks you through creating a Laravel 12 web application that validates form inputs with customized error messages and rules. You'll define your own validation rules and override default error messages to give users a more helpful and user-friendly experience.

7.7.2 Objectives

By completing this lab, you will learn how to:

- Use Laravel's validation system with custom rules
- Define and apply custom error messages
- Display those messages in the Blade view using Bootstrap
- Create a better user experience with clear, contextual validation feedback

7.7.3 Prerequisites

- PHP 8.1+ and Composer installed
- Laravel 12 installed
- Basic knowledge of Laravel routing, controllers, and Blade
- Visual Studio Code or any code editor
- Internet access to load Bootstrap via CDN

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

7.7.4 Steps

Here's a step-by-step guide to creating a custom validation system:

7.7.4.1 Step 1: Create a New Laravel Project

First, create a new Laravel project using the Laravel installer. If you don't have it installed, you can use Composer to create a new project.

```
| laravel new custom-validation  
| cd custom-validation  
| code .
```

You should see a new Laravel project set up. Now, let's create the necessary routes and views.

7.7.4.2 Step 2: Define Routes

Edit `routes/web.php`:

```
use App\Http\Controllers\RegisterController;

Route::get('/register', [RegisterController::class, 'showForm'])->name('register.show');
Route::post('/register', [RegisterController::class, 'handleForm'])->name('register.handle')
```

7.7.4.3 Step 3: Create the Controller

We will create a controller to handle the form logic. Run the following command in your terminal:

```
| php artisan make:controller RegisterController
```

You should see a new controller file in `app/Http/Controllers/RegisterController.php`. Open this file and add the following methods:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class RegisterController extends Controller
{
    public function showForm()
    {
        return view('register');
    }

    public function handleForm(Request $request)
    {
        $customMessages = [
            'name.required' => 'We need to know your name!',
            'email.required' => 'Your email is important to us.',
            'email.email' => 'Hmm... that doesn't look like a valid email.',
            'password.required' => 'Don't forget to set a password.',
            'password.min' => 'Password must be at least :min characters.',
            'username.regex' => 'Username must contain only letters and numbers.',
        ];

        $request->validate([
            'name' => 'required|string|max:100',
            'email' => 'required|email',
            'username' => ['required', 'regex:/^[\w\-\.\_]+@[a-zA-Z0-9]+\.[a-zA-Z]{2,}/'],
            'password' => 'required|min:6',
        ], $customMessages);

        return redirect()->route('register.show')->with('success', 'Registration successful');
    }
}
```

Save the file.

7.7.4.4 Step 4: Create the Blade View

Create a new Blade view file for the form. Create a file named `register.blade.php` inside the `resources/views` directory.

```
<!DOCTYPE html>
<html>
<head>
    <title>Register Form</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container mt-5">

<h2>Custom Validation Example</h2>

@if(session('success'))
    <div class="alert alert-success">
        {{ session('success') }}
    </div>
@endif

<form method="POST" action="{{ route('register.handle') }}">
    @csrf

    <div class="mb-3">
        <label for="name">Full Name</label>
        <input name="name" class="form-control" value="{{ old('name') }}">
        @error('name') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label for="email">Email Address</label>
        <input name="email" class="form-control" value="{{ old('email') }}">
        @error('email') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label for="username">Username</label>
        <input name="username" class="form-control" value="{{ old('username') }}">
        @error('username') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label for="password">Password</label>
        <input type="password" name="password" class="form-control">
        @error('password') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

        <button type="submit" class="btn btn-primary">Register</button>
</form>

</body>
</html>
```

7.7.4.5 Step 5: Run the App

Run the Laravel development server:

```
| php artisan serve
```

Open <http://localhost:8000/register> to test the form.

Try:

- Leaving fields empty
- Entering invalid email
- Typing non-alphanumeric in username
- Using a short password

You'll see **custom validation messages** based on your rules and definitions.

The screenshot shows a web browser window titled "Register Form" with the URL "127.0.0.1:8000/register". The page displays a registration form with four fields: "Full Name", "Email Address", "Username", and "Password". Each field has a red error message below it. The "Full Name" field contains the placeholder "We need to know your name!". The "Email Address" field contains "tester1email" and the message "Hmm... that doesn't look like a valid email.". The "Username" field has the message "The username field is required.". The "Password" field has the message "Password must be at least 6 characters.". A blue "Register" button is at the bottom left of the form.

Figure 7.4: Custom Validation Form.

7.7.5 Summary

In this lab, you learned how to:

- Define custom validation rules and messages using Laravel's built-in validation system
- Create a user-friendly and clear feedback system for forms

- Style forms and error messages using Bootstrap
- Validate alphanumeric usernames using regex

This technique enhances user experience and improves form clarity, which is crucial in modern web development.

7.8 Exercise 18: Use Regex and Conditional Validation in Laravel 12

7.8.1 Description

In this lab, we will create a Laravel 12 web application that uses **regex validation** for complex input patterns (like phone numbers) and **conditional validation** (e.g., requiring additional fields based on a selected option). We'll use Bootstrap for form styling and display validation feedback to the user.

7.8.2 Objectives

By the end of this lab, you will:

- Use **regex** to validate custom patterns such as phone numbers or usernames
- Apply **conditional validation** rules based on the values of other fields
- Use Laravel's validation system to handle complex scenarios
- Build a clean, styled form using **Bootstrap 5**

7.8.3 Prerequisites

- PHP 8.1+ and Composer installed
- Laravel 12 installed
- Basic understanding of Laravel routing, controllers, Blade, and form submission
- Visual Studio Code or any code editor
- Internet access to load Bootstrap via CDN

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

7.8.4 Steps

Here's a step-by-step guide to creating a regex and conditional validation system:

7.8.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project using the Laravel installer. If you don't have it installed, you can use Composer to create a new project.

```
| laravel new regex-conditional  
| cd regex-conditional  
| code .
```

You should see a new Laravel project set up. Now, let's create the necessary routes and views.

7.8.4.2 Step 2: Define Routes

We define the routes for displaying the form and handling the form submission.

Edit `routes/web.php`:

```
| use App\Http\Controllers\ProfileController;  
  
| Route::get('/profile', [ProfileController::class, 'showForm'])->name('profile.show');  
| Route::post('/profile', [ProfileController::class, 'handleForm'])->name('profile.handle');
```

7.8.4.3 Step 3: Create the Controller

We will create a controller to handle the form logic. Run the following command in your terminal:

```
| php artisan make:controller ProfileController
```

You should see a new controller file in `app/Http/Controllers/ProfileController.php`. Open this file and add the following methods:

```
| namespace App\Http\Controllers;  
  
| use Illuminate\Http\Request;  
  
| class ProfileController extends Controller  
{  
|     public function showForm()  
|     {  
|         return view('profile');  
|     }  
  
|     public function handleForm(Request $request)  
|     {  
|         // Validation logic here  
|     }  
| }
```

```

    {
        $request->validate([
            'username' => ['required', 'regex:/^([a-zA-Z0-9_]{5,20})$/'],
            'phone' => ['required', 'regex:/^08[0-9]{8,11}$/'],
            'contact_method' => 'required|in:email,phone',
            'email' => 'required_if:contact_method,email|nullable|email',
        ], [
            'username.regex' => 'Username must be 5-20 characters and contain only letters',
            'phone.regex' => 'Phone must start with 08 and be 10 to 13 digits long.',
            'email.required_if' => 'Email is required if contact method is email.',
        ]);
    }

    return redirect()->route('profile.show')->with('success', 'Profile updated successfully');
}

```

Save the file.

7.8.4.4 Step 4: Create the Blade View

Create a new Blade view file for the form. Create a file named `profile.blade.php` inside the `resources/views` directory.

```

<!DOCTYPE html>
<html>
<head>
    <title>Profile Form</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container mt-5">

<h2>Update Profile</h2>

@if(session('success'))
    <div class="alert alert-success">
        {{ session('success') }}
    </div>
@endif

<form method="POST" action="{{ route('profile.handle') }}">
    @csrf

    <div class="mb-3">
        <label>Username</label>
        <input name="username" class="form-control" value="{{ old('username') }}">
        @error('username') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label>Phone Number</label>
        <input name="phone" class="form-control" value="{{ old('phone') }}">
        @error('phone') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label>Preferred Contact Method</label>
        <select name="contact_method" class="form-select" id="contactMethod">
            <option value="">-- Select --</option>

```

```

        <option value="email" {{ old('contact_method') == 'email' ? 'selected' : '' }}>Email</option>
        <option value="phone" {{ old('contact_method') == 'phone' ? 'selected' : '' }}>Phone</option>
    </select>
    @error('contact_method') <div class="text-danger">{{ $message }}</div> @enderror
</div>

<div class="mb-3" id="emailField" style="display: none;">
    <label>Email Address</label>
    <input name="email" class="form-control" value="{{ old('email') }}">
    @error('email') <div class="text-danger">{{ $message }}</div> @enderror
</div>

<button type="submit" class="btn btn-primary">Submit</button>
</form>

<script>
    const method = document.getElementById('contactMethod');
    const emailField = document.getElementById('emailField');

    function toggleEmailField() {
        emailField.style.display = method.value === 'email' ? 'block' : 'none';
    }

    method.addEventListener('change', toggleEmailField);
    toggleEmailField(); // call on load
</script>
</body>
</html>

```

7.8.4.5 Step 5: Run and Test the App

After completing the above steps, run the Laravel development server:

```
| php artisan serve
```

Open your browser at:

<http://localhost:8000/profile>

The screenshot shows a web browser window with a title bar "Profile Form" and a URL "127.0.0.1:8000/profile". The main content area is titled "Update Profile". It includes three input fields: "Username" (empty), "Phone Number" (empty), and "Preferred Contact Method" (a dropdown menu showing "-- Select --"). Below these is a blue "Submit" button.

Figure 7.5: Profile Form.

7.8.4.6 Test cases:

- Enter a username like `abc` → should fail (too short)
- Enter a phone number like `0812345678` → should pass
- Select “email” as the contact method but leave email blank → should fail
- Select “phone” and leave email blank → should pass

The screenshot shows a web browser window with a title bar "Profile Form" and a URL "127.0.0.1:8000/profile". The main content area is titled "Update Profile". It includes four input fields: "Username" (containing "abc", with a red validation message below it: "Username must be 5-20 characters and contain only letters, numbers, or underscores."), "Phone Number" (containing "0812345678"), "Preferred Contact Method" (a dropdown menu showing "Email"), and "Email Address" (empty, with a red validation message below it: "Email is required if contact method is email"). Below these is a blue "Submit" button.

Figure 7.6: Validation Error.

7.8.5 Summary

In this lab, you learned how to:

- Use **regex** to validate usernames and phone numbers
- Use **conditional validation** with `required_if` to control when certain fields are required
- Display validation feedback clearly using **Bootstrap**
- Enhance forms with JavaScript to dynamically show/hide fields

These techniques are essential for building robust and user-friendly forms in real-world Laravel applications.

7.9 Exercise 19: Multi-Step Form Submission with Session Data in Laravel 12

7.9.1 Description

In this lab, you will build a multi-step form in Laravel 12 where user input is collected across multiple pages and stored in the session. After all steps are completed, the data will be saved or displayed. This is useful for large forms like job applications or registrations.

7.9.2 Objectives

By the end of this lab, you will be able to:

- Break a form into multiple steps
- Store and retrieve form data from session
- Use Bootstrap to style the form
- Manage form state across requests

7.9.3 Prerequisites

- PHP 8.1+ and Composer installed
- Laravel 12 installed
- Basic knowledge of Laravel routing, controllers, and views
- Visual Studio Code or any code editor
- Internet access to load Bootstrap via CDN

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

7.9.4 Steps

Here's a step-by-step guide to creating a multi-step form:

7.9.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project using the Laravel installer. If you don't have it installed, you can use Composer to create a new project.

```
| laravel new multistep-form  
| cd multistep-form  
| code .
```

You should see a new Laravel project set up. Now, let's create the necessary routes and views.

7.9.4.2 Step 2: Create Routes

Open the `routes/web.php` file and define the routes for each step of the form.

```
use App\Http\Controllers\MultiStepFormController;  
  
Route::get('/form/step1', [MultiStepFormController::class, 'step1'])->name('form.step1');  
Route::post('/form/step1', [MultiStepFormController::class, 'storeStep1']);  
  
Route::get('/form/step2', [MultiStepFormController::class, 'step2'])->name('form.step2');  
Route::post('/form/step2', [MultiStepFormController::class, 'storeStep2']);  
  
Route::get('/form/summary', [MultiStepFormController::class, 'summary'])->name('form.summary')
```

Save the file.

7.9.4.3 Step 3: Create the Controller

We will create a controller to handle the multi-step form logic. Run the following command in your terminal:

```
| php artisan make:controller MultiStepFormController
```

You should see a new controller file in `app/Http/Controllers/MultiStepFormController.php`. Open this file and add the following methods:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class MultiStepFormController extends Controller
{
    public function step1()
    {
        return view('form.step1');
    }

    public function storeStep1(Request $request)
    {
        $request->validate([
            'name' => 'required|string|max:100',
            'email' => 'required|email',
        ]);

        session([
            'form.name' => $request->name,
            'form.email' => $request->email,
        ]);

        return redirect()->route('form.step2');
    }

    public function step2()
    {
        return view('form.step2');
    }

    public function storeStep2(Request $request)
    {
        $request->validate([
            'age' => 'required|numeric|min:18',
            'bio' => 'required|string|min:10',
        ]);

        session([
            'form.age' => $request->age,
            'form.bio' => $request->bio,
        ]);

        return redirect()->route('form.summary');
    }

    public function summary()
    {
        $data = session('form');
        return view('form.summary', compact('data'));
    }
}
```

7.9.4.4 Step 4: Create Blade Views

Create a new directory named `form` inside `resources/views` and create the following Blade view files:

- `step1.blade.php`
- `step2.blade.php`
- `summary.blade.php`

`step1.blade.php` will collect basic information, `step2.blade.php` will collect additional information, and `summary.blade.php` will display the collected data.

Here's the code for `step1.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Step 1 - Basic Info</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container mt-5">

<h2>Step 1: Basic Info</h2>
<form method="POST" action="{{ route('form.step1') }}">
    @csrf

    <div class="mb-3">
        <label>Name</label>
        <input type="text" name="name" class="form-control" value="{{ old('name', session('form.name')) }}"/>
        @error('name') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label>Email</label>
        <input type="email" name="email" class="form-control" value="{{ old('email', session('form.email')) }}"/>
        @error('email') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <button type="submit" class="btn btn-primary">Next</button>
</form>

</body>
</html>
```

Here's the code for `step2.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Step 2 - Additional Info</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container mt-5">
```

```

<h2>Step 2: Additional Info</h2>
<form method="POST" action="{{ route('form.step2') }}">
    @csrf

    <div class="mb-3">
        <label>Age</label>
        <input type="number" name="age" class="form-control" value="{{ old('age', session('form.age')) }}"/>
        @error('age') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <div class="mb-3">
        <label>Bio</label>
        <textarea name="bio" class="form-control">{{ old('bio', session('form.bio')) }}</textarea>
        @error('bio') <div class="text-danger">{{ $message }}</div> @enderror
    </div>

    <button type="submit" class="btn btn-primary">Submit</button>
</form>

</body>
</html>

```

Here's the code for `summary.blade.php`:

```

<!DOCTYPE html>
<html>
<head>
    <title>Summary</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container mt-5">

<h2>Form Submission Summary</h2>

@if($data)
    <ul class="list-group">
        <li class="list-group-item"><strong>Name:</strong> {{ $data['name'] }}</li>
        <li class="list-group-item"><strong>Email:</strong> {{ $data['email'] }}</li>
        <li class="list-group-item"><strong>Age:</strong> {{ $data['age'] }}</li>
        <li class="list-group-item"><strong>Bio:</strong> {{ $data['bio'] }}</li>
    </ul>
@else
    <p>No data available.</p>
@endif

</body>
</html>

```

Save all the files.

7.9.4.5 Step 5: Run and Test

After completing the above steps, run the Laravel development server:

```
| php artisan serve
```

Visit <http://localhost:8000/form/step1> and complete each step.

- Step 1: Fill in name and email.
- Step 2: Fill in age and bio.
- Step 3: View the summary.

Each step uses validation and stores data in the session.

The screenshot shows a web browser window titled "Step 1 - Basic Info". The URL in the address bar is "127.0.0.1:8000/form/step1". The page content is titled "Step 1: Basic Info". It contains two input fields: "Name" with the value "Tester 1" and "Email" with the value "tester1@ilmudata.id". A blue "Next" button is at the bottom left.

Figure 7.7: Multi-Step Form Step 1.

The screenshot shows a web browser window titled "Step 2 - Additional Info". The URL in the address bar is "127.0.0.1:8000/form/step2". The page content is titled "Step 2: Additional Info". It contains two input fields: "Age" with the value "25" and "Bio" with the value "Tester 1 is a king". A blue "Submit" button is at the bottom left.

Figure 7.8: Multi-Step Form Step 2.

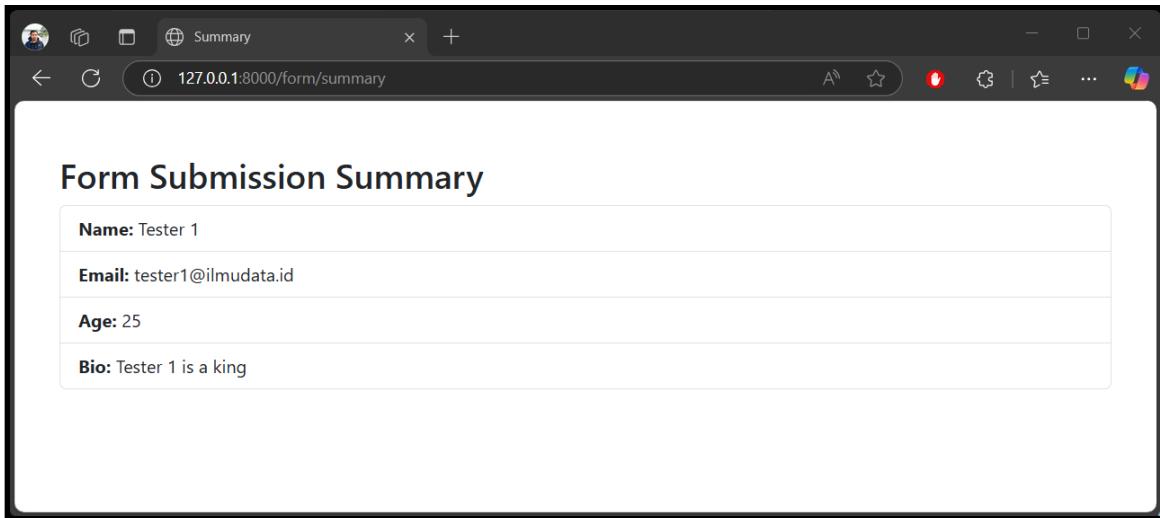


Figure 7.9: Multi-Step Form Summary.

7.9.5 Summary

In this lab, you built a **multi-step form with session data storage** in Laravel 12. You learned how to:

- Break forms into steps using multiple routes and views
- Validate each step's data independently
- Store user inputs in session between steps
- Style the forms using Bootstrap

Multi-step forms improve UX for lengthy forms and are commonly used in wizards, checkout processes, and registration flows.

7.10 Exercise 20: Form Submission with AJAX and Validation Response in Laravel 12

7.10.1 Description

In this lab, you will build a Laravel 12 web application that submits a form using JavaScript's `fetch()` API (AJAX) and handles validation errors in JSON format. Error messages will be shown directly in the form without refreshing the page. This approach improves the user experience by providing immediate feedback.

7.10.2 Objectives

By the end of this lab, you will be able to:

- Create a form that submits data asynchronously using `fetch()`
- Validate the data on the server and return a JSON response
- Display error messages dynamically using JavaScript
- Use Bootstrap for form styling and error display

7.10.3 Prerequisites

- PHP 8.1+ and Composer installed
- Laravel 12 installed
- Basic understanding of Laravel routing, controllers, and Blade
- Visual Studio Code or any code editor
- Browser with JavaScript enabled
- Internet access to load Bootstrap via CDN

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

7.10.4 Steps

Here's a step-by-step guide to creating an AJAX form submission system:

7.10.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project using the Laravel installer. If you don't have it installed, you can use Composer to create a new project.

```
| laravel new ajax-form  
| cd ajax-form  
| code .
```

You should see a new Laravel project set up. Now, let's create the necessary routes and views.

7.10.4.2 Step 2: Define Routes

We define the routes for displaying the form and handling the AJAX submission.

Edit `routes/web.php`:

```
| use App\Http\Controllers\AjaxFormController;  
|  
| Route::get('/ajax-form', [AjaxFormController::class, 'showForm'])->name('ajax.form');  
| Route::post('/ajax-submit', [AjaxFormController::class, 'submitForm'])->name('ajax.submit')  
|
```

Save the file.

7.10.4.3 Step 3: Create Controller

We will create a controller to handle the AJAX form logic. Run the following command in your terminal:

```
| php artisan make:controller AjaxFormController
```

You should see a new controller file in `app/Http/Controllers/AjaxFormController.php`. Open this file and add the following methods:

```
| namespace App\Http\Controllers;  
|  
| use Illuminate\Http\Request;  
| use Illuminate\Support\Facades\Validator;  
|  
| class AjaxFormController extends Controller  
{  
|     public function showForm()  
|     {  
|         return view('ajax-form');  
|     }  
|  
|     public function submitForm(Request $request)  
|     {  
|         $validator = Validator::make($request->all(), [  
|             'username' => 'required|min:5',  
|             'email' => 'required|email',  
|         ]);  
|  
|         if ($validator->fails()) {  
|             return response()->json([  
|                 'errors' => $validator->errors(),  
|             ], 422);  
|         }  
|  
|         return response()->json([  
|             'message' => 'Form submitted successfully!',  
|         ]);  
|     }  
| }
```

Save the file.

7.10.4.4 Step 4: Create Blade View

Create a new Blade view file for the AJAX form. Create a file named `ajax-form.blade.php` inside the `resources/views` directory.

```
<!DOCTYPE html>
<html>
<head>
    <title>AJAX Form with Validation</title>
    <meta name="csrf-token" content="{{ csrf_token() }}">
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container mt-5">

<h2>AJAX Form Submission</h2>

<div id="success-message" class="alert alert-success d-none"></div>

<form id="ajaxForm">
    <div class="mb-3">
        <label>Username</label>
        <input name="username" id="username" class="form-control">
        <div class="text-danger" id="username-error"></div>
    </div>

    <div class="mb-3">
        <label>Email</label>
        <input name="email" id="email" class="form-control">
        <div class="text-danger" id="email-error"></div>
    </div>

    <button type="submit" class="btn btn-primary">Submit</button>
</form>

<script>
document.getElementById('ajaxForm').addEventListener('submit', async function(e) {
    e.preventDefault();

    // Clear previous errors
    document.getElementById('username-error').textContent = '';
    document.getElementById('email-error').textContent = '';
    document.getElementById('success-message').classList.add('d-none');

    const formData = {
        username: document.getElementById('username').value,
        email: document.getElementById('email').value,
    };

    const token = document.querySelector('meta[name="csrf-token"]').getAttribute('content');

    const response = await fetch('{{ route("ajax.submit") }}', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'X-CSRF-TOKEN': token,
        },
        body: JSON.stringify(formData),
    });

    if (response.status === 422) {
        const data = await response.json();
        const errors = data.errors;

        if (errors.username) {
            document.getElementById('username-error').textContent = errors.username[0];
        }
    }
});</script>
```

```
        }
        if (errors.email) {
            document.getElementById('email-error').textContent = errors.email[0];
        }
    } else if (response.ok) {
    const data = await response.json();
    document.getElementById('success-message').textContent = data.message;
    document.getElementById('success-message').classList.remove('d-none');
    document.getElementById('ajaxForm').reset();
}
});
</script>

</body>
</html>
```

Explain the code:

- The form has two fields: username and email.
- When the form is submitted, it prevents the default action and sends an AJAX request to the server using `fetch()`.
- The CSRF token is included in the request headers for security.
- If validation fails, the server responds with a 422 status code and the errors are displayed under the respective fields.
- If the submission is successful, a success message is displayed, and the form is reset.
- The success message is hidden by default and shown only when the form submission is successful.

Save the file.

7.10.4.5 Step 5: Run the App

After completing the above steps, run the Laravel development server:

```
| php artisan serve
```

Open <http://localhost:8000/ajax-form>

A screenshot of a web browser window titled "AJAX Form with Validation". The URL in the address bar is "127.0.0.1:8000/ajax-form". The page content is a form titled "AJAX Form Submission" with two fields: "Username" and "Email", each with a text input field. Below the inputs is a blue "Submit" button.

Figure 7.10: AJAX Form.

7.10.4.6 Test Cases:

- Leave fields empty → JSON validation errors shown under each field
- Use invalid email or short username → specific validation messages appear
- Submit valid data → success message appears without page reload

A screenshot of a web browser window titled "AJAX Form with Validation". The URL in the address bar is "127.0.0.1:8000/ajax-form". The page content is a form titled "AJAX Form Submission" with two fields: "Username" and "Email". The "Username" field contains "abc" and has a red validation message below it: "The username field must be at least 5 characters.". The "Email" field contains "tester1ilmudata.id" and has a red validation message below it: "The email field must be a valid email address.". Below the inputs is a blue "Submit" button.

Figure 7.11: Validation Error.

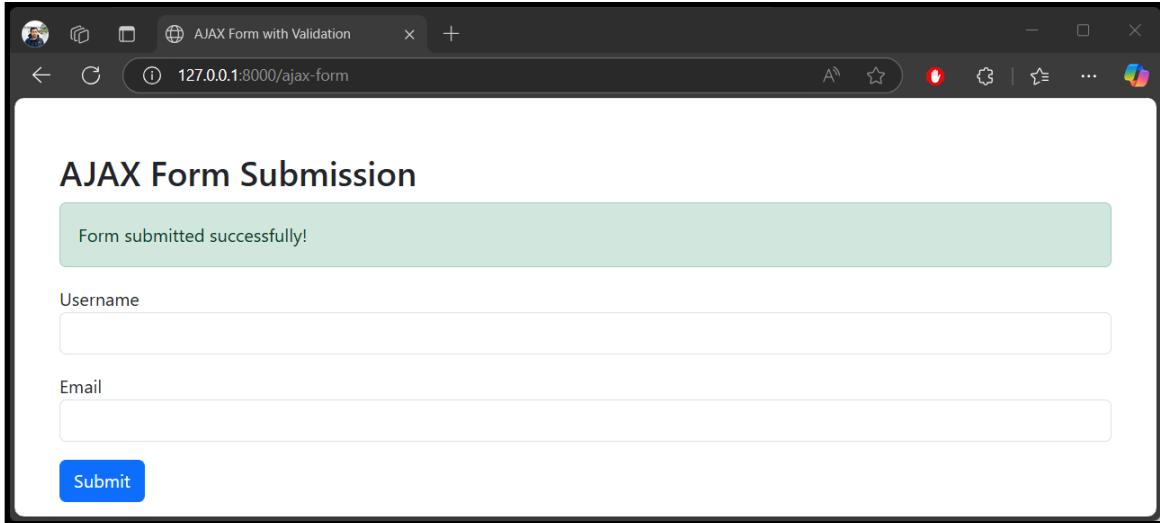


Figure 7.12: Success Message.

7.10.5 Summary

In this lab, you created a **form submission system using AJAX and JSON validation responses** in Laravel 12. You learned how to:

- Submit form data using JavaScript `fetch()`
- Validate data on the server and return structured error messages
- Dynamically show validation messages in the form using JavaScript
- Style the form and errors using Bootstrap

This approach enables seamless user interaction and a more responsive form experience for modern web applications.

7.11 Conclusion

In this chapter, we covered various aspects of form handling in Laravel 12, including validation, multi-step forms, AJAX submissions, and custom validation rules. These techniques are essential for building robust and user-friendly web applications. By mastering these concepts, you will be well-equipped to handle complex form scenarios in your Laravel projects.

8 Model

8.1 Introduction

In Laravel, a model is a central part of the MVC (Model-View-Controller) architecture. It represents the application's data structure and is used to interact with the database.

8.2 Understanding Models in Laravel

- Models are typically linked to a table in the database.
- Laravel uses Eloquent ORM to handle database operations with models.

Example:

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

This command creates a `Product` model in `app/Models/Product.php`.

Code Example:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Product extends Model
{
    protected $fillable = ['name', 'price', 'stock'];
}
```

8.3 Entities and POCO (Plain Old Class Object)

While Laravel doesn't enforce strict entity definitions like in DDD (Domain-Driven Design), it's useful to treat models as entities representing business logic.

POCO in Laravel can be understood as simple PHP classes without Laravel-specific behavior.

Example:

```

class ProductEntity
{
    public function __construct(
        public string $name,
        public float $price,
        public int $stock,
    ) {}
}

```

This class can be used independently of the framework.

8.4 Data Transfer Object (DTO)

DTOs are used to transfer data between layers in a structured format. They help separate raw data (often from requests) from business logic.

Example:

```

namespace App\DTO;

class ProductDTO
{
    public function __construct(
        public string $name,
        public float $price,
        public int $stock,
    ) {}

    public static function fromRequest(array $data): self
    {
        return new self(
            $data['name'],
            $data['price'],
            $data['stock']
        );
    }
}

```

8.5 Repository Pattern

The repository pattern abstracts the data access logic and promotes a separation of concerns.

Interface:

```

namespace App\Repositories;

interface ProductRepositoryInterface
{
    public function all();
    public function find(int $id);
}

```

```
|     public function create(array $data);  
| }
```

Implementation:

```
| namespace App\Repositories;  
  
use App\Models\Product;  
  
class ProductRepository implements ProductRepositoryInterface  
{  
    public function all()  
    {  
        return Product::all();  
    }  
  
    public function find(int $id)  
    {  
        return Product::find($id);  
    }  
  
    public function create(array $data)  
    {  
        return Product::create($data);  
    }  
}
```

Service Provider Binding:

```
| use App\Repositories\ProductRepository;  
use App\Repositories\ProductRepositoryInterface;  
  
public function register()  
{  
    $this->app->bind(ProductRepositoryInterface::class, ProductRepository::class);  
}
```

8.6 Exercise 21: Using Model for Form Binding and Display

8.6.1 Description

In this exercise, we will create a simple Laravel 12 application that uses a model to bind data from a form and display it on a new page. We will not use a database in this lab; instead, we will use a plain PHP class (POCO-style) to simulate model behavior.

We will create a simple form to collect product information (name, price, and description) and display the submitted data on a new page. We will use Bootstrap

for styling.

8.6.2 Objectives

- Create a simple model (POCO-style) to hold product data.
- Bind form data to the model using a static method.
- Display the data on a new page using Bootstrap for styling.
- Use Laravel's routing and controller to handle form submission and display results.
- No database connection is required.

8.6.3 Prerequisites

- Laravel 12 installed.
- No database connection is required.
- Basic understanding of routes, controllers, and views in Laravel.
- Visual Studio Code or any text editor.
- Basic knowledge of HTML and Bootstrap.

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

8.6.4 Steps

Here's a step-by-step guide to creating the application:

8.6.4.1 Step 1: Create Laravel Project

We will create a new Laravel project. If you already have a project, you can skip this step.

```
| laravel new productapp  
| cd productapp  
| code .
```

You can also use the existing project from the previous lab.

8.6.4.2 Step 2: Create a Simple Data Model (POCO)

Create a folder, `ViewModels`, inside the `app` directory to hold our model class:

```
| mkdir app/ViewModels
```

Then create a simple model class to hold product data. This class will have properties for the product name, price, and description. It will also have a static method `fromRequest()` to map data from the request to the model.

Create `ProductViewModel.php` inside the `app/ViewModels` directory. Here's the code for the model class:

```
<?php
namespace App\ViewModels;

class ProductViewModel
{
    public string $name;
    public float $price;
    public string $description;

    public function __construct(string $name = '', float $price = 0, string $description =
    {
        $this->name = $name;
        $this->price = $price;
        $this->description = $description;
    }

    public static function fromRequest(array $data): self
    {
        return new self(
            $data['name'] ?? '',
            (float)($data['price'] ?? 0),
            $data['description'] ?? ''
        );
    }
}
```

Save the file. This class will be used to hold product data without using a database.

8.6.4.3 Step 3: Create Controller

We will create a controller to handle the form submission and display the results. The controller will use the `ProductViewModel` class.

Create the controller using the following command:

```
| php artisan make:controller ProductController
```

Then, modify the controller to handle the form submission and display the results. The controller will use the `ProductViewModel` class to bind data from the form.

Edit file `app/Http/Controllers/ProductController.php`:

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\ViewModels\ProductViewModel;

class ProductController extends Controller
{
    public function create()
    {
        return view('product.create');
    }

    public function result(Request $request)
    {
        $product = ProductViewModel::fromRequest($request->all());
        return view('product.result', compact('product'));
    }
}
```

Save the file. This controller will handle the form submission and display the results using the `ProductViewModel` class.

8.6.4.4 Step 4: Define Routes

We define the routes for the application. The first route will display the form, and the second route will handle the form submission and display the results.

Edit `routes/web.php`:

```
use App\Http\Controllers\ProductController;

Route::get('/product/create', [ProductController::class, 'create'])->name('product.create')
Route::post('/product/result', [ProductController::class, 'result'])->name('product.result')
```

Save the file. This will set up the routes for our application.

8.6.4.5 Step 5: Create Views with Bootstrap

Create `product` directory inside `resources/views`:

```
mkdir resources/views/product
```

Then create two files: `create.blade.php` and `result.blade.php`.

Here are the contents of each file:

View 1: resources/views/product/create.blade.php

```
<!DOCTYPE html>
<html>
<head>
    <title>Create Product</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container py-5">
    <h2>Create Product (No Database)</h2>
    <form method="POST" action="{{ route('product.result') }}>
        @csrf
        <div class="mb-3">
            <label class="form-label">Name</label>
            <input name="name" class="form-control" required>
        </div>
        <div class="mb-3">
            <label class="form-label">Price</label>
            <input name="price" type="number" step="0.01" class="form-control" required>
        </div>
        <div class="mb-3">
            <label class="form-label">Description</label>
            <textarea name="description" class="form-control"></textarea>
        </div>
        <button type="submit" class="btn btn-primary">Submit Product</button>
    </form>
</body>
</html>
```

View 2: resources/views/product/result.blade.php

```
<!DOCTYPE html>
<html>
<head>
    <title>Product Result</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container py-5">
    <h2>Submitted Product Details</h2>
    <ul class="list-group">
        <li class="list-group-item"><strong>Name:</strong> {{ $product->name }}</li>
        <li class="list-group-item"><strong>Price:</strong> {{$product->price, 2}}</li>
        <li class="list-group-item"><strong>Description:</strong> {{ $product->description }}</li>
    </ul>
    <a href="{{ route('product.create') }}" class="btn btn-link mt-3">Submit Another Product</a>
</body>
</html>
```

Save the files. These views will be used to display the form and the results.

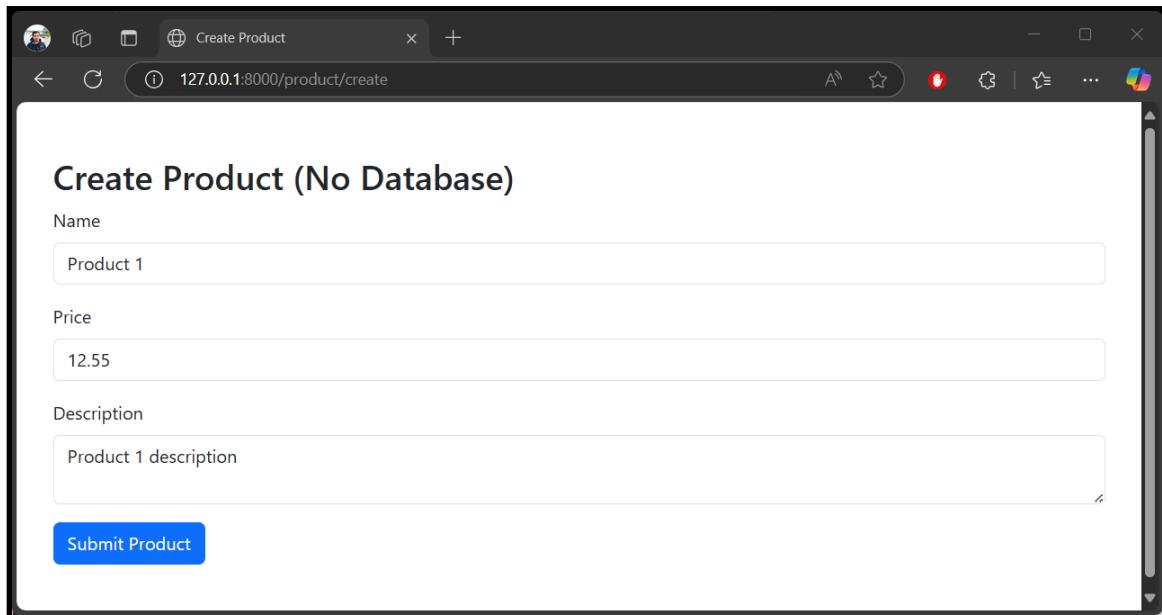
8.6.4.6 Step 6: Test the App

After completing the above steps, run the application using the following command:

```
| php artisan serve
```

Open browser and visit:

<http://localhost:8000/product/create>

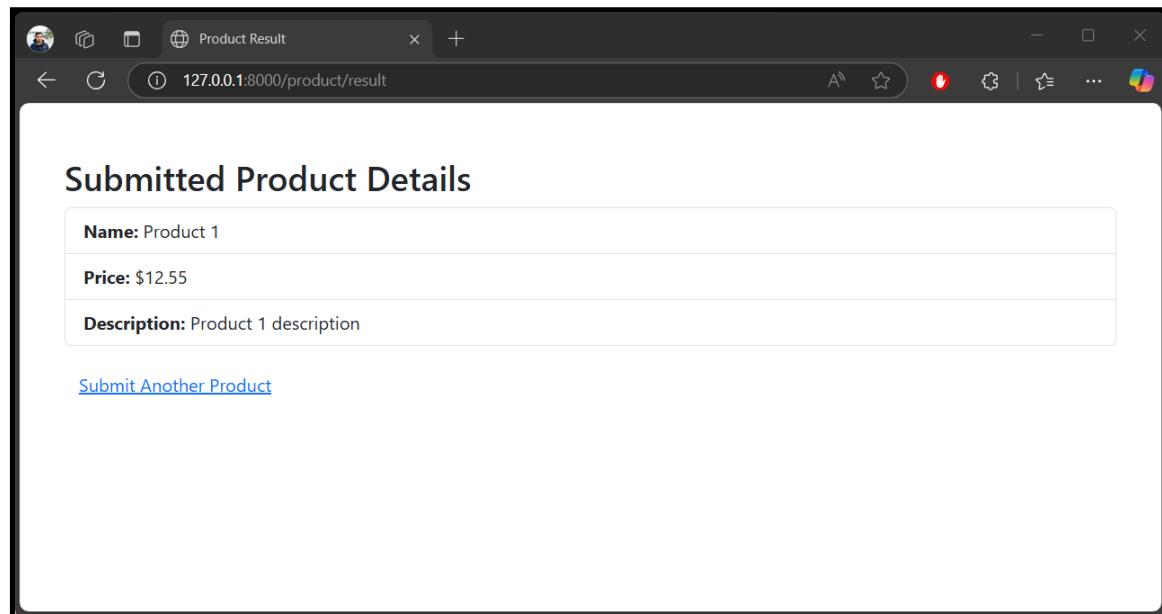


The screenshot shows a web browser window titled "Create Product". The URL in the address bar is "127.0.0.1:8000/product/create". The page content is a form titled "Create Product (No Database)". It contains three input fields: "Name" with value "Product 1", "Price" with value "12.55", and "Description" with value "Product 1 description". Below the form is a blue "Submit Product" button.

Figure 8.1: Product Form.

Fill in the form and submit. You will see the results displayed on a new page without saving to a database.

!



The screenshot shows a web browser window titled "Product Result". The URL in the address bar is "127.0.0.1:8000/product/result". The page content displays the submitted product details under the heading "Submitted Product Details". The details are: **Name:** Product 1, **Price:** \$12.55, and **Description:** Product 1 description. Below the details is a blue "Submit Another Product" button.

8.6.5 Summary

In this lab, we learned how to:

- Create a simple model (POCO) to hold data without using a database.
- Bind form data to the model using a static method.
- Display the data on a new page using Bootstrap for styling.
- Use Laravel's routing and controller to handle form submission and display results.

8.7 Exercise 22: Build and Use a Data Transfer Object (DTO)

8.7.1 Description

In this lab, we'll build a simple Laravel 12 web application that uses a **Data Transfer Object (DTO)** to pass data between the controller and the service layer. We won't use a database in this lab — the goal is to understand how to structure data and use DTOs for clean, maintainable code.

8.7.2 Objectives

- Create a `ProductDTO` class inside `app/DTO`.
- Add a `fromRequest()` static method to map data from a form.
- Use the DTO in a controller and pass it to a service class.
- Style the form and result page using Bootstrap.

8.7.3 Prerequisites

- Laravel 12 installed.
- No database connection is required.
- Visual Studio Code or any text editor.
- Basic understanding of routes, controllers, and views in Laravel.

This lab uses PHP 8.4, Laravel 12 and Visual Studio Code as the primary editor.

8.7.4 Steps

Here's a step-by-step guide to creating the application:

8.7.4.1 Step 1: Create Laravel Project

We will create a new Laravel project. If you already have a project, you can skip this step.

```
| laravel new productdtoapp  
| cd productdtoapp  
| code .
```

You can also use the existing project from the previous lab.

8.7.4.2 Step 2: Create the DTO Class

We will create a DTO class to hold product data. This class will have properties for the product name, price, and description. It will also have a static method `fromRequest()` to map data from the request to the DTO.

Create `DTO` folder inside `app`:

```
| mkdir app/DTO
```

Then create the file `app/DTO/ProductDTO.php`:

```
<?php  
namespace App\DTO;  
  
class ProductDTO  
{  
    public string $name;  
    public float $price;  
    public string $description;  
  
    public function __construct(string $name, float $price, string $description)  
    {  
        $this->name = $name;  
        $this->price = $price;  
        $this->description = $description;  
    }  
  
    public static function fromRequest(array $data): self  
    {  
        return new self(  
            $data['name'] ?? '',  
            (float)($data['price'] ?? 0),  
            $data['description'] ?? ''  
        );  
    }  
}
```

Save the file. This class will be used to transfer data between the controller and the service layer.

8.7.4.3 Step 3: Create the Service Layer

Create a service class that will handle the business logic. This class will take the DTO as input and return the formatted data.

Create `Services` folder inside `app`:

```
| mkdir app/Services
```

Create the file `app/Services/ProductService.php`:

```
<?php
namespace App\Services;

use App\DTO\ProductDTO;

class ProductService
{
    public function display(ProductDTO $product): array
    {
        return [
            'name' => $product->name,
            'price' => $product->price,
            'description' => $product->description,
        ];
    }
}
```

Save the file. This service class will be responsible for processing the data from the DTO and returning it in a structured format.

8.7.4.4 Step 4: Create the Controller

We will create a controller to handle the form submission and display the results. The controller will use the `ProductDTO` and `ProductService` classes.

Create the controller using the following command:

```
| php artisan make:controller ProductController
```

You can also create the controller manually by creating a file named `ProductController.php` in the `app/Http/Controllers` directory.

Edit the file `app/Http/Controllers/ProductController.php`:

```

<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\DTO\ProductDTO;
use App\Services\ProductService;

class ProductController extends Controller
{
    public function create()
    {
        return view('product.create');
    }

    public function result(Request $request)
    {
        $dto = ProductDTO::fromRequest($request->all());
        $service = new ProductService();
        $product = $service->display($dto);

        return view('product.result', compact('product'));
    }
}

```

8.7.4.5 Step 5: Define the Routes

We define the routes for the application. The first route will display the form, and the second route will handle the form submission and display the results.

Edit `routes/web.php`:

```

use App\Http\Controllers\ProductController;

Route::get('/product/create', [ProductController::class, 'create'])->name('product.create')
Route::post('/product/result', [ProductController::class, 'result'])->name('product.result')

```

8.7.4.6 Step 6: Create the Views (with Bootstrap)

Create `product` directory inside `resources/views`:

After creating the directory, create two files: `create.blade.php` and `result.blade.php`.

Here are the contents of each file: View 1: `resources/views/product/create.blade.php`

```

<!DOCTYPE html>
<html>
<head>
    <title>Create Product DTO</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>

```

```

<body class="container py-5">
    <h2>Create Product</h2>
    <form method="POST" action="{{ route('product.result') }}">
        @csrf
        <div class="mb-3">
            <label class="form-label">Name</label>
            <input name="name" class="form-control" required>
        </div>
        <div class="mb-3">
            <label class="form-label">Price</label>
            <input name="price" type="number" step="0.01" class="form-control" required>
        </div>
        <div class="mb-3">
            <label class="form-label">Description</label>
            <textarea name="description" class="form-control"></textarea>
        </div>
        <button type="submit" class="btn btn-primary">Submit Product</button>
    </form>
</body>
</html>

```

View 2: resources/views/product.blade.php

```

<!DOCTYPE html>
<html>
<head>
    <title>Product Result</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container py-5">
    <h2>Product DTO Result</h2>
    <ul class="list-group">
        <li class="list-group-item"><strong>Name:</strong> {{ $product['name'] }}</li>
        <li class="list-group-item"><strong>Price:</strong> {{$product['price'] | number_format(2)}}</li>
        <li class="list-group-item"><strong>Description:</strong> {{ $product['description'] }}</li>
    </ul>
    <a href="{{ route('product.create') }}" class="btn btn-link mt-3">Submit Another Product</a>
</body>
</html>

```

Save the files. These views will be used to display the form and the results.

8.7.4.7 Step 7: Run and Test the App

After completing the above steps, run the application using the following command:

```
| php artisan serve
```

Visit: <http://localhost:8000/product/create>

The screenshot shows a web browser window titled "Create Product DTO". The URL is "127.0.0.1:8000/product/create". The page content is a "Create Product" form. It contains three input fields: "Name" with value "Product 5", "Price" with value "9.99", and "Description" with value "Product 5 description". Below the form is a blue "Submit Product" button.

Figure 8.3: Product Form.

Fill in the form → Submit → View the results passed via DTO and service.

The screenshot shows a web browser window titled "Product Result". The URL is "127.0.0.1:8000/product/result". The page content is titled "Product DTO Result" and displays the submitted product details:
Name: Product 5
Price: \$9.99
Description: Product 5 description

Figure 8.4: Product Result.

8.7.5 Summary

In this lab, you learned how to:

- Create a Data Transfer Object (DTO) for handling form data.
- Use `fromRequest()` to map raw request data to a structured object.

- Pass the DTO to a service layer for further processing.
- Use Bootstrap to design clean, modern forms and output.

8.8 Exercise 23: Use Model Accessors and Mutators

8.8.1 Description

In this lab, we'll build a simple Laravel 12 web application **without using a database** to demonstrate how **Accessors** and **Mutators** work. We'll use a **plain model class** to mimic Laravel model behavior and apply accessors to format the price and mutators to convert the product name to uppercase. This is useful to understand the concept before using Laravel's Eloquent ORM.

8.8.2 Objectives

- Create a plain model class (POCO-style) with accessors and mutators.
- Add an accessor to format the price as currency.
- Add a mutator to convert the name to uppercase.
- Display the transformed data using a controller and views.
- Use Bootstrap for a clean UI.

8.8.3 Prerequisites

- Laravel 12 installed.
- No database required.
- Visual Studio Code or any text editor.
- Familiarity with controllers, views, and routing in Laravel.

This lab uses PHP 8.4, Laravel 12 and Visual Studio Code as the primary editor.

8.8.4 Steps

Here's a step-by-step guide to creating the application:

8.8.4.1 Step 1: Create Laravel Project

We will create a new Laravel project. If you already have a project, you can skip this step.

```
| laravel new modelaccessorapp  
| cd modelaccessorapp  
| code .
```

You should see the Laravel project structure in your editor.

8.8.4.2 Step 2: Create a Plain Model Class

We will create a plain model class (POCO-style) to hold product data. This class will have properties for the product name and price. We will also add accessors and mutators to format the data.

Create `ViewModels` folder inside `app`:

```
| mkdir app/ViewModels
```

Then create a plain model class `ProductViewModel.php` inside the `app/ViewModels` directory:

```
<?php  
namespace App\ViewModels;  
  
class ProductViewModel  
{  
    private string $name;  
    private float $price;  
  
    public function __construct(string $name, float $price)  
    {  
        $this->setName($name);  
        $this->price = $price;  
    }  
  
    // Mutator: Convert name to uppercase  
    public function setName(string $value): void  
    {  
        $this->name = strtoupper($value);  
    }  
  
    // Accessor: Get name  
    public function getName(): string  
    {  
        return $this->name;  
    }  
  
    // Accessor: Format price as currency  
    public function getFormattedPrice(): string  
    {  
        return '$' . number_format($this->price, 2);  
    }  
}
```

```
| } }
```

Save the file. This class will be used to hold product data and apply accessors and mutators.

8.8.4.3 Step 3: Create Controller

We will create a controller to handle the form submission and display the results. The controller will use the `ProductViewModel` class.

```
| php artisan make:controller ProductController
```

You should see the controller file created in `app/Http/Controllers/ProductController.php`.

Update `app/Http/Controllers/ProductController.php`:

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\ViewModels\ProductViewModel;

class ProductController extends Controller
{
    public function create()
    {
        return view('product.create');
    }

    public function result(Request $request)
    {
        $product = new ProductViewModel(
            $request->input('name'),
            (float) $request->input('price')
        );

        return view('product.result', [
            'name' => $product->getName(),
            'formattedPrice' => $product->getFormattedPrice()
        ]);
    }
}
```

8.8.4.4 Step 4: Define Routes

We define the routes for the application. The first route will display the form, and the second route will handle the form submission and display the results.

Edit `routes/web.php`:

```
| use App\Http\Controllers\ProductController;  
|  
| Route::get('/product/create', [ProductController::class, 'create'])->name('product.create')  
| Route::post('/product/result', [ProductController::class, 'result'])->name('product.result')  
|
```

Save the file. This will set up the routes for our application.

8.8.4.5 Step 5: Create Views with Bootstrap

We create `product` directory inside `resources/views`:

```
| mkdir resources/views/product
```

Then, we create two files: `create.blade.php` and `result.blade.php`.

Here are the contents of each file:

View 1: `resources/views/product/create.blade.php`

```
<!DOCTYPE html>  
<html>  
<head>  
    <title>Create Product (No DB)</title>  
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">  
</head>  
<body class="container py-5">  
    <h2>Create Product</h2>  
    <form method="POST" action="{{ route('product.result') }}>  
        @csrf  
        <div class="mb-3">  
            <label class="form-label">Name</label>  
            <input name="name" class="form-control" required>  
        </div>  
        <div class="mb-3">  
            <label class="form-label">Price</label>  
            <input name="price" type="number" step="0.01" class="form-control" required>  
        </div>  
        <button type="submit" class="btn btn-primary">Submit</button>  
    </form>  
</body>  
</html>
```

View 2: `resources/views/product/result.blade.php`

```
<!DOCTYPE html>  
<html>  
<head>  
    <title>Product Result</title>  
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">  
</head>  
<body class="container py-5">  
    <h2>Product Summary</h2>  
    <ul class="list-group">
```

```
<li class="list-group-item"><strong>Name (mutated to uppercase):</strong> {{ $name }}</li>
    <li class="list-group-item"><strong>Formatted Price (accessor):</strong> {{ $formattedPrice }}</li>
</ul>
<a href="{{ route('product.create') }}" class="btn btn-link mt-3">Submit Another Product</a>
</body>
</html>
```

8.8.4.6 Step 6: Run and Test the App

After completing the above steps, run the application using the following command:

```
| php artisan serve
```

Visit: <http://localhost:8000/product/create>

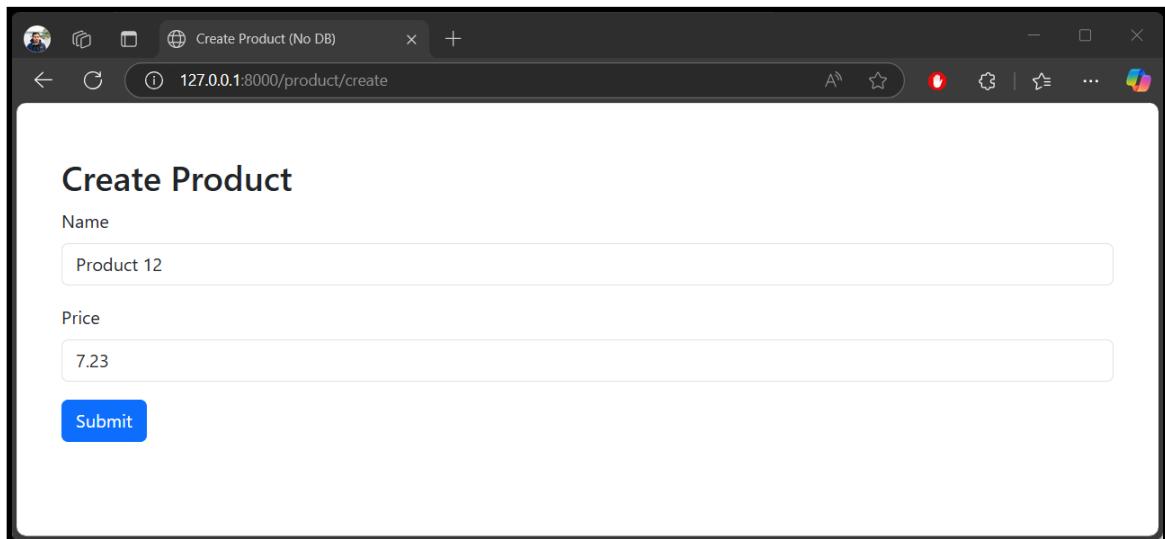
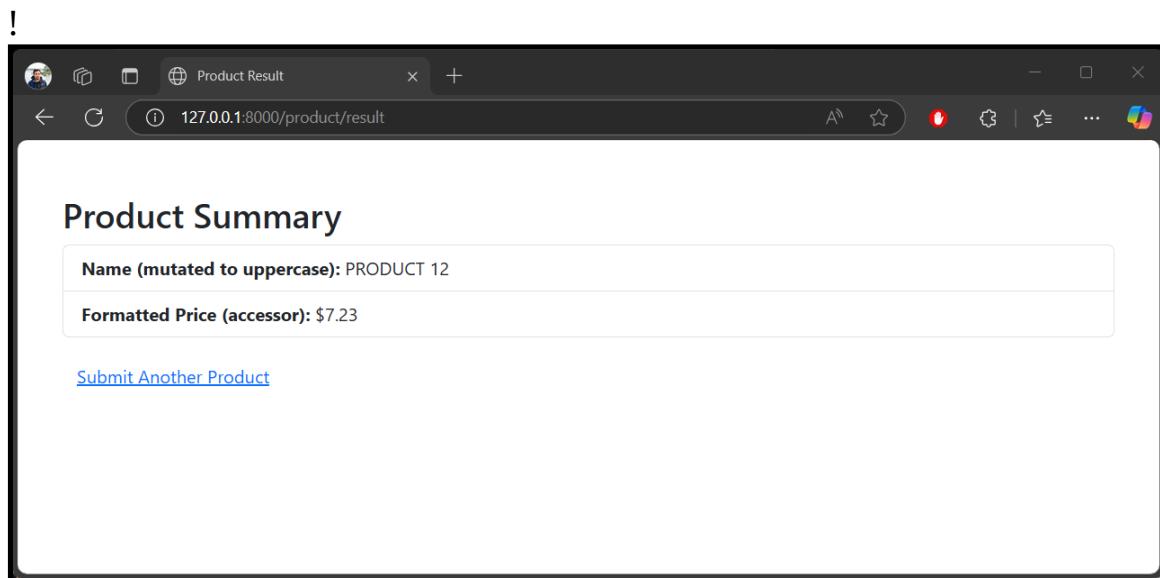


Figure 8.5: Product Form.

Submit the form and see the results on the next page with **uppercase name** and **formatted price**.



8.8.5 Summary

In this lab, you learned how to:

- Use a plain PHP model class (POCO-style) in Laravel without a database.
- Create and apply **mutators** to modify input values (e.g., name to uppercase).
- Create and apply **accessors** to format output values (e.g., price to currency).
- Pass processed model data from controller to views.
- Build a clean UI using Bootstrap.

8.9 Conclusion

We have covered the following key concepts in this chapter:

- Laravel models are the backbone of data handling in Laravel applications.
- POCOs and DTOs help in structuring data without relying on Eloquent ORM.
- The repository pattern abstracts data access, promoting separation of concerns.
- Accessors and mutators allow for data transformation and validation within models.
- Using plain PHP classes can simplify the understanding of data handling in Laravel.
- Laravel's routing and controller system makes it easy to handle form submissions and display results.

- Bootstrap can be used to create clean and modern user interfaces for web applications.

OceanofPDF.com

9 File and Storage Management

9.1 Introduction

File handling is a crucial aspect of web applications, especially when dealing with user-generated content. Laravel provides a powerful and flexible file storage system that allows developers to easily manage files across different storage systems, including local disks and cloud services like Amazon S3.

This chapter will cover the basics of file handling in Laravel, including how to upload, retrieve, and delete files. We will also explore the differences between public and private storage, and how to work with remote disks.

9.2 Configuration

Laravel's filesystem configuration is located in `config/filesystems.php`. By default, Laravel uses the local disk for file storage, but you can easily configure it to use other disks like Amazon S3 or Google Cloud Storage.

Environment variables are used to set up the filesystem configuration. For example, you can set the default disk in your `.env` file:

```
| FILESYSTEM_DISK=local
```

9.3 Storing Files

In this section, we will cover how to store files in Laravel. We will discuss the different storage disks available and how to configure them.

We will also explore how to upload files using forms and handle file uploads in controllers.

9.3.1 Uploading via Form

We will create a controller method to handle file uploads. Here's an example of how to do this:

```
public function upload(Request $request)
{
    $path = $request->file('document')->store('documents');
    return back()->with('success', "File stored at: $path");
}
```

We can customize the storage path and file name using the `storeAs` method:

```
public function upload(Request $request)
{
    $path = $request->file('document')->storeAs('documents', 'myfile.pdf');
    return back()->with('success', "File stored at: $path");
}
```

9.3.2 Retrieving Files

Getting files from storage is straightforward. You can use the `storage` facade to retrieve files.

```
public function show($filename)
{
    $path = storage_path("app/documents/$filename");
    return response()->file($path);
}
```

We can display files in views using the `storage` facade:

```

```

9.3.3 Downloading Files

We can also allow users to download files. Here's how to do it:

```
return Storage::download($path);
```

9.3.4 Deleting Files

To delete files, we can use the `delete` method:

```
Storage::delete($path);
```

9.3.5 File Existence and Metadata

Checking if a file exists and retrieving its metadata is also easy with Laravel:

```
Storage::exists($path);
```

We also have methods to get file metadata like size and last modified time:

```
| Storage::size($path);  
| Storage::lastModified($path);
```

9.4 Public vs Private Files

Laravel provides two types of storage: public and private. Public files are accessible via a URL, while private files are not. To store files publicly, you can use the `public` disk:

```
| Storage::disk('public')->put('file.txt', 'Contents');
```

To access public files, you can create a symbolic link using the `storage:link` command:

```
| php artisan storage:link
```

This will create a symbolic link from `public/storage` to `storage/app/public`, allowing you to access files stored in the public disk.

9.5 Working with Remote Disks

Laravel supports remote disks like Amazon S3, Google Cloud Storage, and others. You can configure these disks in the `config/filesystems.php` file.

To upload files to a remote disk, you can use the same methods as with local disks. For example, to upload a file to Amazon S3:

```
| Storage::disk('s3')->put('folder/file.txt', 'Contents');
```

9.6 Exercise 24: Upload and Display Image

9.6.1 Description

In this lab, we'll build a Laravel 12 web app that allows users to upload an image with a custom file name and description. After uploading, the app redirects to a new page to display the image and the input data — all **without storing anything in a database**.

9.6.2 Objectives

You will learn to:

- Handle image uploads and validations in Laravel
- Rename uploaded files using user input
- Store files in Laravel's public disk
- Use Bootstrap to style the form and result
- Use `session()` to pass data between requests

9.6.3 Prerequisites

- Laravel 12 installed
- PHP \geq 8.2
- Laravel cli installed
- Visual Studio Code or any text editor
- `php artisan serve` or local web server
- Storage link created (`php artisan storage:link`)

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

9.6.4 Steps

Here's how to implement the image upload feature without a database:

9.6.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project for this lab. Open your terminal and run:

```
| laravel new image-upload-no-db  
| cd image-upload-no-db  
| code .
```

You should see the Laravel project in your editor.

9.6.4.2 Step 2: Create Controller

We will create a controller to handle the image upload logic. Run the following command in your terminal:

```
| php artisan make:controller ImageUploadController
```

You can find the controller in `app/Http/Controllers/ImageUploadController.php`. This controller will handle the form display, image upload, and result display.

Edit app/Http/Controllers/ImageUploadController.php:

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Str;

class ImageUploadController extends Controller
{
    public function showForm()
    {
        return view('upload');
    }

    public function handleUpload(Request $request)
    {
        $validated = $request->validate([
            'file_name' => 'required|string|max:255',
            'description' => 'required|string|max:1000',
            'image' => 'required|image|mimes:jpeg,png,jpg,gif|max:2048',
        ]);

        $filename = Str::uuid() . '.' . $request->file('image')->getClientOriginalExtension();
        $path = $request->file('image')->storeAs('uploads', $filename, 'public');

        return redirect()->route('upload.result')->with([
            'file_path' => $path,
            'file_name' => $validated['file_name'],
            'description' => $validated['description'],
        ]);
    }

    public function showResult()
    {
        if (!session()->has('file_path')) {
            return redirect()->route('upload.form');
        }

        return view('result', [
            'file_path' => session('file_path'),
            'file_name' => session('file_name'),
            'description' => session('description'),
        ]);
    }
}
```

Save the file. This controller has three methods:

- `showForm()`: Displays the upload form.
- `handleUpload()`: Handles the image upload and stores it in the `public/uploads` directory.
- `showResult()`: Displays the uploaded image and its metadata.

9.6.4.3 Step 3: Define Routes

We need to define routes for our controller methods. Open `routes/web.php` and add the following code:

```
use App\Http\Controllers\ImageUploadController;

Route::get('/upload', [ImageUploadController::class, 'showForm'])->name('upload.form');
Route::post('/upload', [ImageUploadController::class, 'handleUpload'])->name('upload.handle');
Route::get('/result', [ImageUploadController::class, 'showResult'])->name('upload.result');
```

Save the file. This code defines three routes:

- `/upload`: Displays the upload form.
- `/upload`: Handles the image upload.
- `/result`: Displays the uploaded image and its metadata.

9.6.4.4 Step 4: Create Views with Bootstrap

We will create two views: one for the upload form and another for displaying the uploaded image and its metadata.

Create a new directory named `views` in `resources` and create two files: `upload.blade.php` and `result.blade.php`.

Here's the code for each view:

View `resources/views/upload.blade.php`:

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

@section('content')
<h2>Upload Image</h2>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul class="mb-0">
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<form method="POST" action="{{ route('upload.handle') }}" enctype="multipart/form-data">
    @csrf
    <div class="mb-3">
        <label class="form-label">File Name</label>
        <input type="text" name="file_name" class="form-control" required>
    </div>
```

```

<div class="mb-3">
    <label class="form-label">Description</label>
    <textarea name="description" class="form-control" rows="3" required></textarea>
</div>
<div class="mb-3">
    <label class="form-label">Choose Image</label>
    <input type="file" name="image" class="form-control" required>
</div>
<button class="btn btn-primary">Upload</button>
</form>
@endsection

```

View resources/views/result.blade.php:

```

@extends('layouts.app')

@section('content')
<h2>Uploaded Image</h2>

<div class="card">
    
    <div class="card-body">
        <h5 class="card-title">{{ $file_name }}</h5>
        <p class="card-text">{{ $description }}</p>
        <a href="{{ route('upload.form') }}" class="btn btn-secondary">Upload Another</a>
    </div>
</div>
@endsection

```

Next, we need to create a layout file for our views. Create a new directory named layouts in resources/views and create a file named app.blade.php.

Here's the code for the layout file:

Layout resources/views/layouts/app.blade.php:

```

<!DOCTYPE html>
<html>
<head>
    <title>Laravel 12 Image Upload</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="bg-light">
    <div class="container mt-5">
        @yield('content')
    </div>
</body>
</html>

```

9.6.4.5 Step 5: Create Storage Symlink

We need to create a symbolic link to the storage directory so that we can access the uploaded images via the browser. Run the following command:

```
| php artisan storage:link
```

You should see a message indicating that the link has been created successfully. This command creates a symbolic link from `public/storage` to `storage/app/public`, allowing you to access files stored in the public disk.

9.6.4.6 Step 6: Run the Laravel Development Server

After completing the above steps, you can run the Laravel development server to test the application. Run the following command in your terminal:

```
| php artisan serve
```

Access in browser:

`http://localhost:8000/upload`

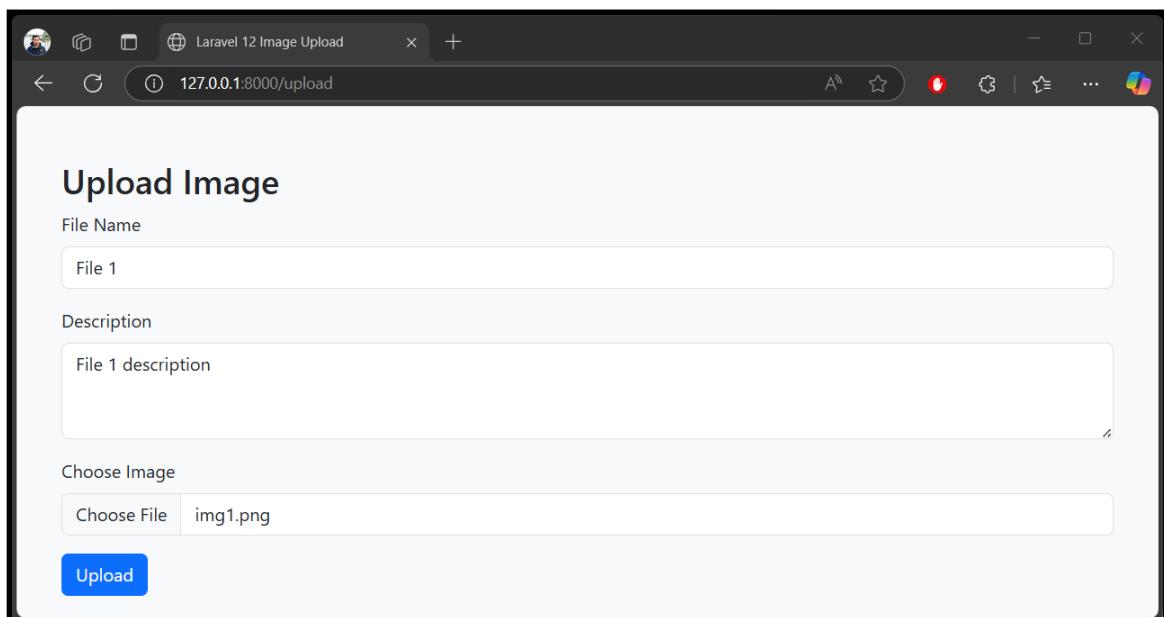


Figure 9.1: Upload form.

Enter a file name, description, and choose an image to upload. After submitting the form, you should be redirected to a new page displaying the uploaded image and its metadata.

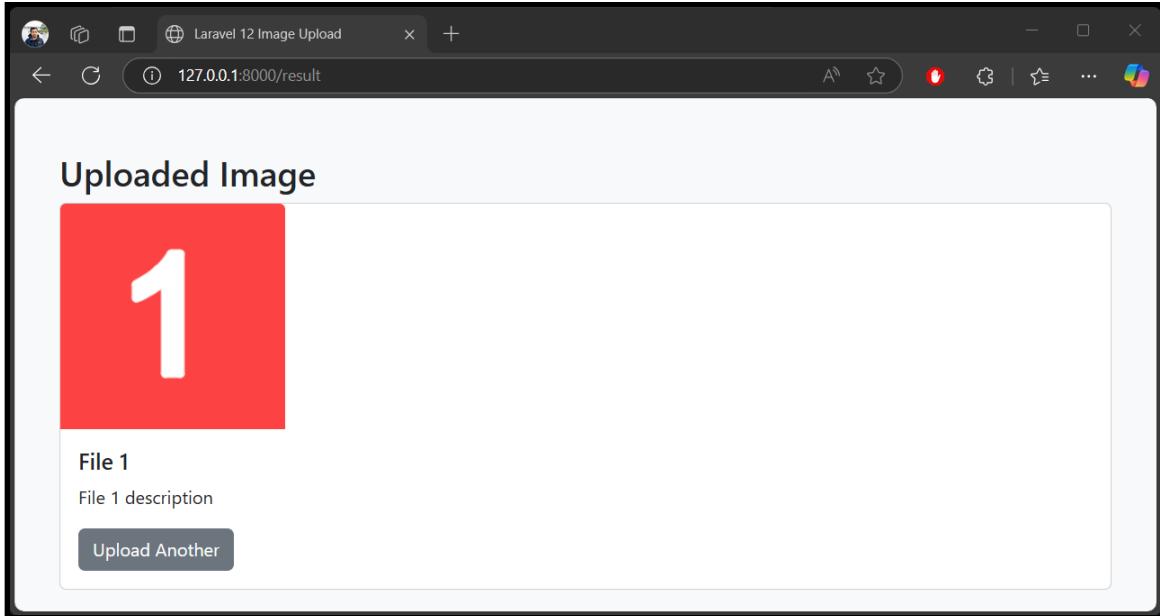


Figure 9.2: Displaying image.

9.6.5 Summary

In this lab, you learned how to:

- Create an image upload form with custom file name and description
- Store image in `public/storage/uploads`
- Redirect to a new page to display image and its metadata
- Style with Bootstrap
- Pass data using Laravel's session

9.7 Exercise 25: Upload and Display Image to MinIO (S3-Compatible)

9.7.1 Description

This lab shows how to upload an image with a custom name and description in Laravel 12 and store it in **MinIO**, an S3-compatible storage system running via Docker. The image and data will be displayed after upload without using a database.

9.7.2 Objectives

You will learn to:

- Configure Laravel to connect to MinIO as an S3-compatible disk
- Upload an image with validation to MinIO
- Generate public URL for display
- Use Docker to run MinIO locally
- Style the form and output with Bootstrap

9.7.3 Prerequisites

- Laravel 12 installed
- PHP >= 8.2
- Laravel cli installed
- Composer installed
- Visual Studio Code or any text editor
- Docker installed

If you don't have Docker installed, you can download it from Docker's official website, <https://www.docker.com/get-started>.

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

9.7.4 Steps

Here are the steps to implement the image upload feature using MinIO:

9.7.4.1 Step 1: Run MinIO with Docker

First, we need to run MinIO using Docker. Open your terminal and run the following command:

```
docker run -p 9000:9000 -p 9001:9001 \
-e "MINIO_ROOT_USER=minioadmin" \
-e "MINIO_ROOT_PASSWORD=minioadmin" \
quay.io/minio/minio server /data --console-address ":9001"
```

If you have Windows, you can use the following command:

```
docker run -p 9000:9000 -p 9001:9001 ^
-e "MINIO_ROOT_USER=minioadmin" ^
```

```
-e "MINIO_ROOT_PASSWORD=minioadmin" ^
quay.io/minio/minio server /data --console-address ":9001"
```

```
Command Prompt - docker r + ▾
quay.io/minio/minio:latest

What's next:
  View a summary of image vulnerabilities and recommendations → docker scout quickview quay.io/minio/minio

C:\Users\lenovo>docker run -p 9000:9000 -p 9001:9001 ^
More? -e "MINIO_ROOT_USER=minioadmin" ^
More? -e "MINIO_ROOT_PASSWORD=minioadmin" ^
More? quay.io/minio/minio server /data --console-address ":9001"
INFO: Formatting 1st pool, 1 set(s), 1 drives per set.
INFO: WARNING: Host local has more than 0 drives of set. A host failure will result in data becoming unavailable.
MinIO Object Storage Server
Copyright: 2015-2025 MinIO, Inc.
License: GNU AGPLv3 - https://www.gnu.org/licenses/agpl-3.0.html
Version: RELEASE.2025-03-12T18-04-18Z (go1.24.1 linux/amd64)

API: http://172.17.0.2:9000 http://127.0.0.1:9000
WebUI: http://172.17.0.2:9001 http://127.0.0.1:9001

Docs: https://docs.min.io
WARN: Detected default credentials 'minioadmin:minioadmin', we recommend that you change these values with 'MINIO_ROOT_U
SER' and 'MINIO_ROOT_PASSWORD' environment variables
```

Figure 9.3: MinIO running in Docker.

Open MinIO Console:

<http://localhost:9001>

Login with:

- Username: minioadmin
- Password: minioadmin

Create a bucket named: **laravel-uploads**

You can do this in the MinIO Console by clicking on **Buckets** and then **Create Bucket**. Name it laravel-uploads.

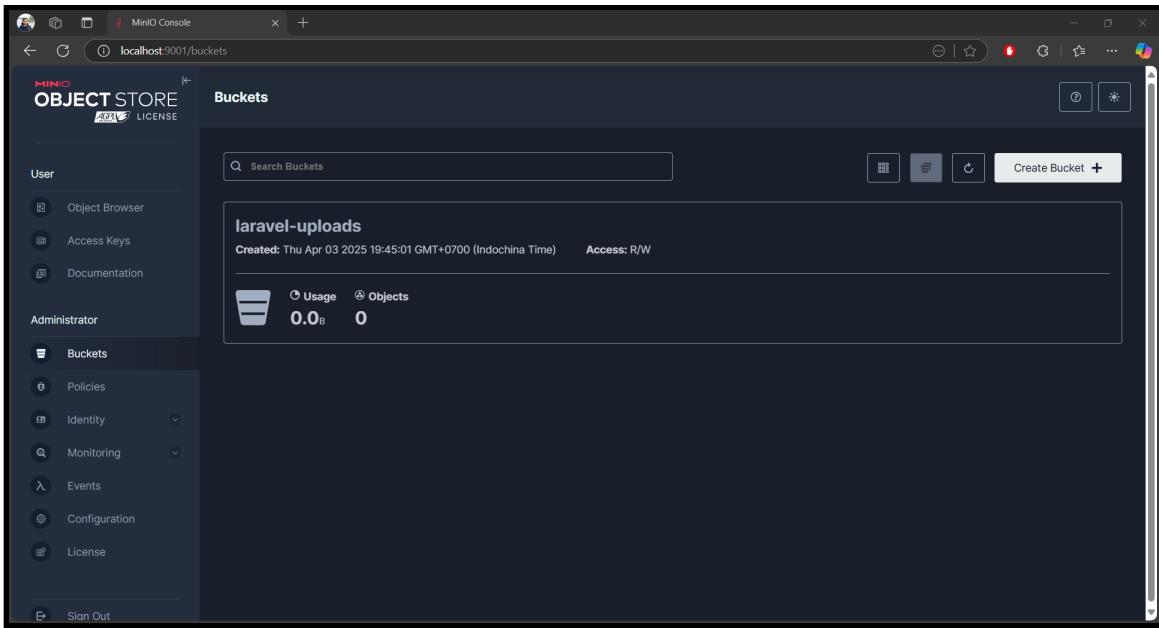


Figure 9.4: Create bucket in MinIO.

Generate an access key and secret key for your MinIO instance.

You can do this in the MinIO Console by clicking on **Access Keys** and then **Create Access Key**. Set name as `laravel`.

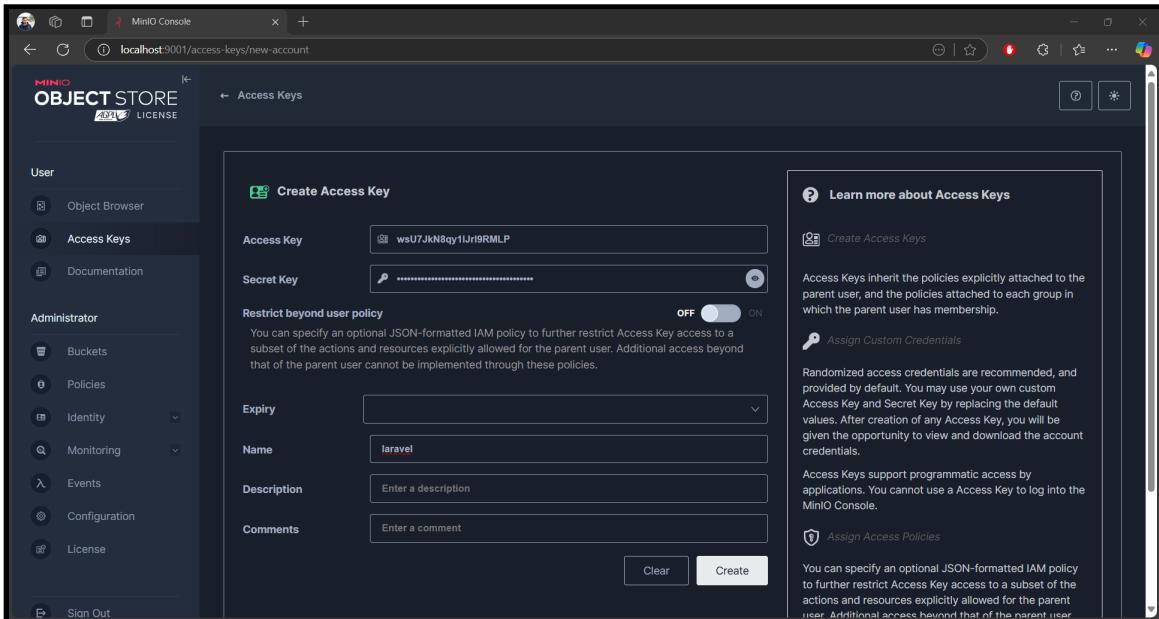


Figure 9.5: Create access key in MinIO.

Keep the access key and secret key handy, as you will need them to configure your Laravel application.

Configure bucket policy to allow public access.

You can do this in the MinIO Console by clicking on **Buckets**, selecting your bucket, and then **Edit Anonymous**. Set the policy to allow public access:

- Prefix: /
- Policy: readOnly

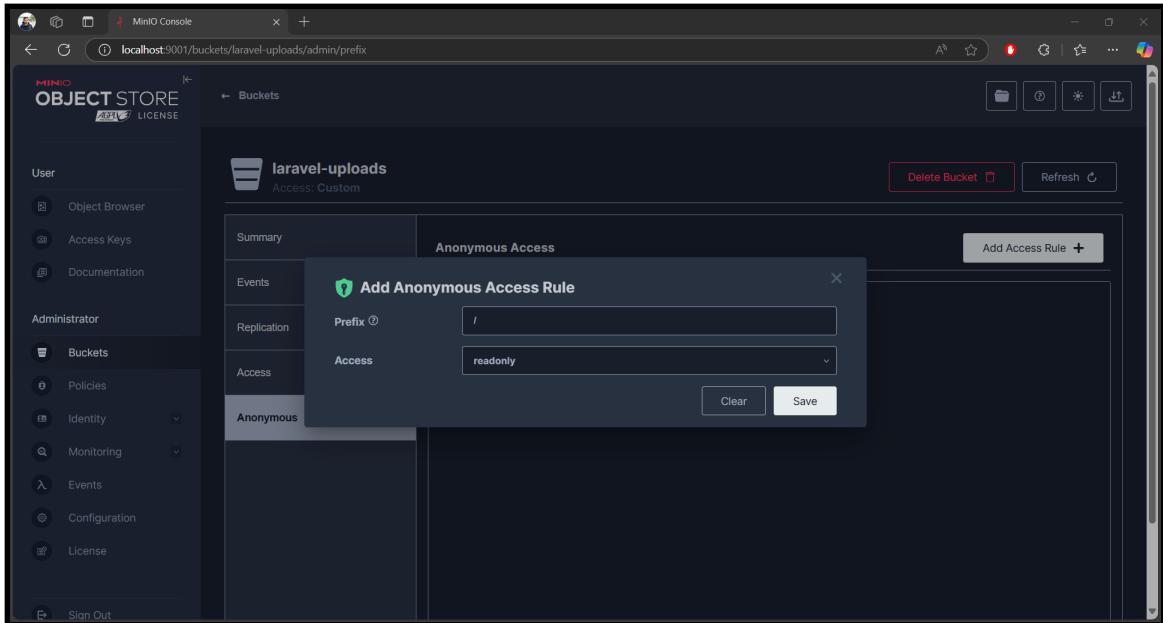


Figure 9.6: Configure bucket policy in MinIO for public access.

9.7.4.2 Step 2: Create a New Laravel Project

We will create a new Laravel project for this lab. Open your terminal and run:

```
| laravel new image-upload-minio  
| cd image-upload-minio  
| code .
```

You should see the Laravel project in your editor.

9.7.4.3 Step 3: Install Laravel Filesystem AWS Adapter

We need to install the AWS S3 adapter for Laravel's filesystem to work with MinIO. Run the following command in your terminal:

```
| composer require league/flysystem-aws-s3-v3 "^3.0" --with-all-dependencies
```

You can find the adapter in `config/filesystems.php`.

9.7.4.4 Step 4: Configure `.env` for S3 / MinIO

We need to set up the environment variables for MinIO in the `.env` file. Open `.env` and add the following lines:

```
FILESYSTEM_DISK=s3  
AWS_ACCESS_KEY_ID=<your-access-key>  
AWS_SECRET_ACCESS_KEY=<your-secret-key>  
AWS_DEFAULT_REGION=us-east-1  
AWS_BUCKET=laravel-uploads  
AWS_URL=http://localhost:9000/local  
AWS_ENDPOINT=http://localhost:9000  
AWS_USE_PATH_STYLE_ENDPOINT=true
```

Change the `AWS_BUCKET` to the name of the bucket you created in MinIO (e.g., `laravel-uploads`). Change `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` to the access key and secret key you generated in MinIO.

9.7.4.5 Step 5: Update `config/filesystems.php`

Find the `s3` disk section and make sure it looks like this:

```
's3' => [  
    'driver' => 's3',  
    'key' => env('AWS_ACCESS_KEY_ID'),  
    'secret' => env('AWS_SECRET_ACCESS_KEY'),  
    'region' => env('AWS_DEFAULT_REGION'),  
    'bucket' => env('AWS_BUCKET'),  
    'url' => env('AWS_URL'),  
    'use_path_style_endpoint' => env('AWS_USE_PATH_STYLE_ENDPOINT', false),  
],
```

Save the file. This code configures the `s3` disk to use MinIO with the credentials provided in the `.env` file.

9.7.4.6 Step 6: Create Controller

We will create a controller to handle the image upload logic. Run the following command in your terminal:

```
| php artisan make:controller MinioImageUploadController
```

In `app/Http/Controllers/MinioImageUploadController.php`, add the following code:

```

<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Str;

class MinioImageUploadController extends Controller
{
    public function showForm()
    {
        return view('upload');
    }

    public function handleUpload(Request $request)
    {
        $validated = $request->validate([
            'file_name' => 'required|string|max:255',
            'description' => 'required|string|max:1000',
            'image' => 'required|image|mimes:jpeg,png,jpg,gif|max:2048',
        ]);

        $filename = Str::uuid() . '.' . $request->file('image')->getClientOriginalExtension();
        $path = Storage::disk('s3')->putFileAs(
            '',
            $request->file('image'),
            $filename,
            ['visibility' => 'public',
            'disk' => 's3']
        );

        // Manually construct public URL
        $bucket = config('filesystems.disks.s3.bucket');
        $endpoint = rtrim(config('filesystems.disks.s3.endpoint'), '/');
        $publicUrl = "{$endpoint}/{$bucket}/{$path}";

        return redirect()->route('upload.result')->with([
            'file_url' => $publicUrl,
            'file_name' => $validated['file_name'],
            'description' => $validated['description'],
        ]);
    }

    public function showResult()
    {
        if (!session()->has('file_url')) {
            return redirect()->route('upload.form');
        }

        return view('result', [
            'file_url' => session('file_url'),
            'file_name' => session('file_name'),
            'description' => session('description'),
        ]);
    }
}

```

This controller has three methods:

- `showForm()`: Displays the upload form.
- `handleUpload()`: Handles the image upload and stores it in MinIO.
- `showResult()`: Displays the uploaded image and its metadata.

Save the file.

9.7.4.7 Step 7: Define Routes

We define routes for our controller methods. Open `routes/web.php` and add the following code:

```
use App\Http\Controllers\MinioImageUploadController;

Route::get('/upload', [MinioImageUploadController::class, 'showForm'])->name('upload.form')
Route::post('/upload', [MinioImageUploadController::class, 'handleUpload'])->name('upload')
Route::get('/result', [MinioImageUploadController::class, 'showResult'])->name('upload.result')
```

This code defines three routes:

- GET `/upload`: Displays the upload form.
- POST `/upload`: Handles the image upload.
- GET `/result`: Displays the uploaded image and its metadata.

9.7.4.8 Step 8: Create Views

We will create two views: one for the upload form and another for displaying the uploaded image and its metadata.

Create `layouts` directory in `resources/views` and create a file named `app.blade.php`.

Here's the code for the layout file:

```
<!DOCTYPE html>
<html>
<head>
    <title>Laravel MinIO Upload</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="bg-light">
    <div class="container mt-5">
        @yield('content')
    </div>
</body>
</html>
```

We also create `upload.blade.php` and `result.blade.php` in the `resources/views` directory.

Add codes for resources/views/upload.blade.php:

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

@section('content')
<h2>Upload Image to MinIO</h2>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul class="mb-0">
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<form method="POST" action="{{ route('upload.handle') }}" enctype="multipart/form-data">
    @csrf
    <div class="mb-3">
        <label class="form-label">File Name</label>
        <input type="text" name="file_name" class="form-control" required>
    </div>
    <div class="mb-3">
        <label class="form-label">Description</label>
        <textarea name="description" class="form-control" rows="3" required></textarea>
    </div>
    <div class="mb-3">
        <label class="form-label">Choose Image</label>
        <input type="file" name="image" class="form-control" required>
    </div>
    <button class="btn btn-primary">Upload</button>
</form>
@endsection
```

Add codes for resources/views/result.blade.php:

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

@section('content')
<h2>Uploaded Image</h2>

<div class="card">
    
    <div class="card-body">
        <h5 class="card-title">{{ $file_name }}</h5>
        <p class="card-text">{{ $description }}</p>
        <a href="{{ route('upload.form') }}" class="btn btn-secondary">Upload Another</a>
    </div>
</div>
@endsection
```

Save the files. The upload.blade.php file contains the upload form, while result.blade.php displays the uploaded image and its metadata.

9.7.4.9 Step 9: Run Laravel and Upload

After completing the above steps, you can run the Laravel development server to test the application. Run the following command in your terminal:

```
| php artisan serve
```

Then go to:

<http://localhost:8000/upload>

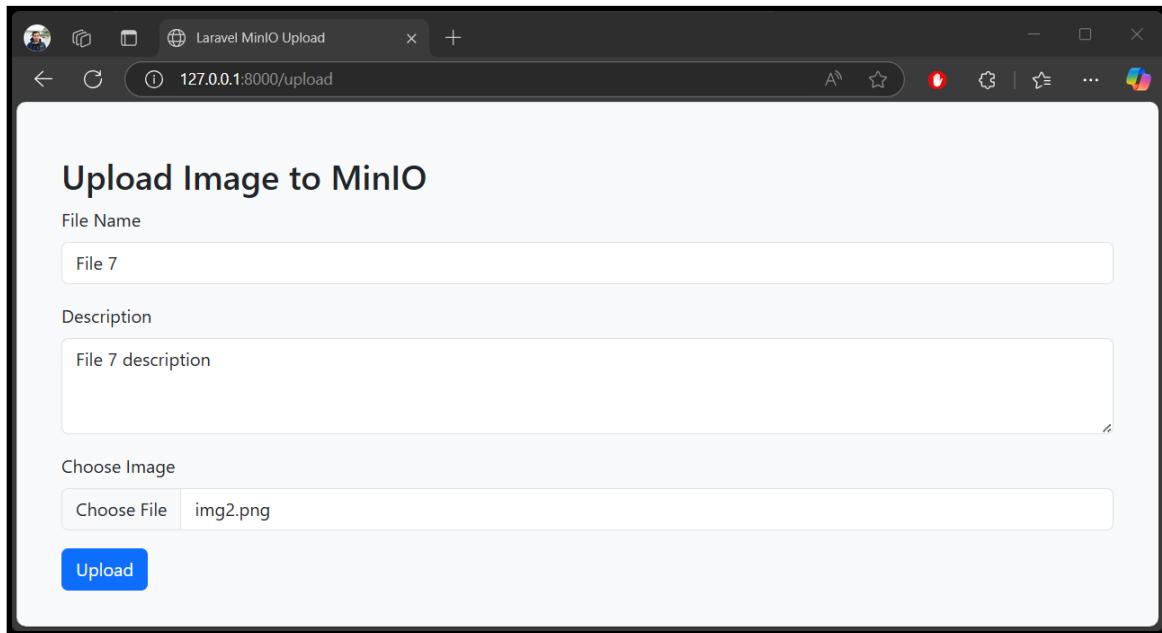


Figure 9.7: Upload form in MinIO.

Upload an image and test it! The image will be uploaded to MinIO and shown using the public URL generated by Laravel.

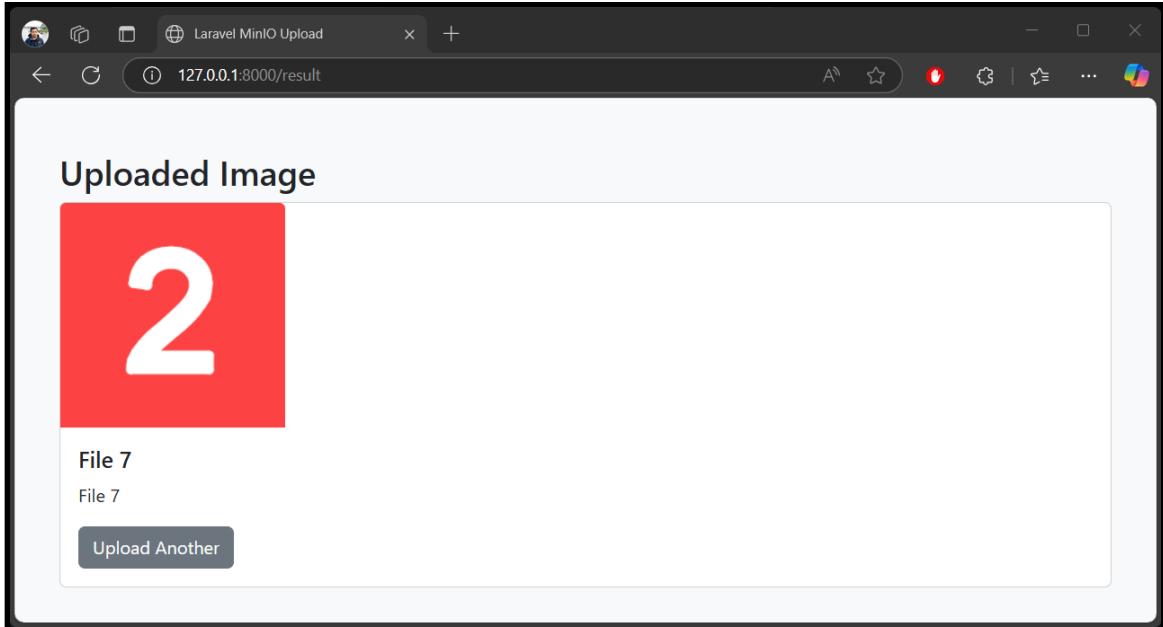


Figure 9.8: Displaying image from MinIO.

You can also check the uploaded image in the MinIO Console under the `laravel-uploads` bucket.

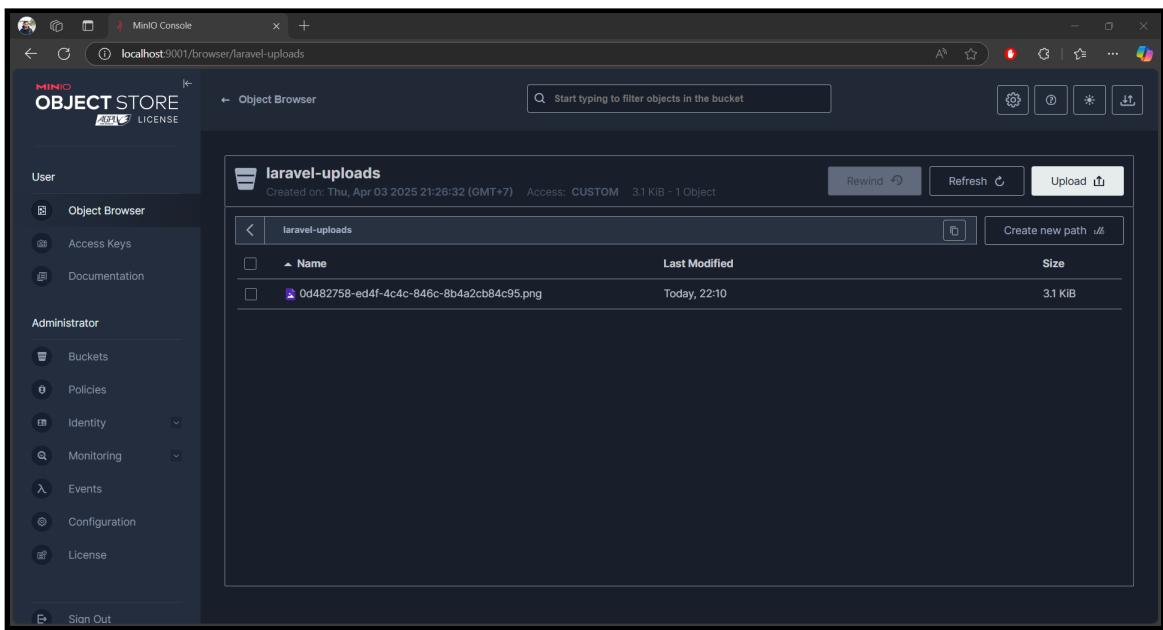


Figure 9.9: Uploaded image in MinIO Console.

9.7.5 Summary

In this lab, you learned how to:

- Use Docker to run MinIO as an S3-compatible service
- Configure Laravel to connect to MinIO using the `s3` driver
- Upload images and display them using session data
- Style the app using Bootstrap

9.8 Exercise 26: Upload and Display Image from Private Storage

9.8.1 Description

This hands-on lab demonstrates how to upload an image with a custom file name and description, store it securely in Laravel's **private storage** (`local` disk), and display it on another page **without saving anything to the database**. The image is streamed via a controller to prevent public access.

9.8.2 Objectives

You will learn to:

- Upload and validate images in Laravel
- Store files in private (`local`) storage
- Display images by streaming them securely
- Style the UI with Bootstrap
- Pass image metadata using Laravel session

9.8.3 Prerequisites

- Laravel 12 installed
- PHP ≥ 8.2
- Laravel CLI installed
- Visual Studio Code or any text editor
- Basic Laravel knowledge
- `php artisan serve` to run the app

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

9.8.4 Steps

Here are the steps to implement the image upload feature using private storage:

9.8.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project for this lab. Open your terminal and run:

```
| laravel new image-upload-private  
| cd image-upload-private  
| code .
```

You should see the Laravel project in your editor.

9.8.4.2 Step 2: Set Storage to Local (Private)

Open the `.env` file and set the `FILESYSTEM_DISK` to `local`. This will ensure that files are stored in the private storage directory (`storage/app`).

```
| FILESYSTEM_DISK=local
```

9.8.4.3 Step 3: Create the Controller

We will create a controller to handle the image upload logic. Run the following command in your terminal:

```
| php artisan make:controller PrivateImageUploadController
```

You can find the controller in `app/Http/Controllers/PrivateImageUploadController.php`. This controller will handle the form display, image upload, and result display.

Edit `app/Http/Controllers/PrivateImageUploadController.php`:

```
<?php  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
use Illuminate\Support\Facades\Storage;  
use Symfony\Component\HttpFoundation\Response;  
use Illuminate\Support\Str;  
  
class PrivateImageUploadController extends Controller  
{  
    public function showForm()  
    {  
        return view('upload');  
    }  
  
    public function handleUpload(Request $request)  
    {  
        $image = $request->file('image');  
        $name = Str::random(10) . '.' . $image->getClientOriginalExtension();  
        $path = $image->store($name, 'public');  
        $response = ['path' => $path];  
        return response()->json($response);  
    }  
}
```

```

{
    $validated = $request->validate([
        'file_name' => 'required|string|max:255',
        'description' => 'required|string|max:1000',
        'image' => 'required|image|mimes:jpeg,png,jpg,gif|max:2048',
    ]);

    $extension = $request->file('image')->getClientOriginalExtension();
    $filename = Str::uuid() . '.' . $extension;

    // Store in local (private) storage
    $path = $request->file('image')->storeAs('private-images', $filename, 'local');

    return redirect()->route('private.result')->with([
        'file_path' => $path,
        'file_name' => $validated['file_name'],
        'description' => $validated['description'],
    ]);
}

public function showResult()
{
    if (!session()->has('file_path')) {
        return redirect()->route('private.form');
    }

    return view('result', [
        'file_path' => session('file_path'),
        'file_name' => session('file_name'),
        'description' => session('description'),
    ]);
}

public function viewImage($filename)
{
    $path = 'private-images/' . $filename;

    if (!Storage::disk('local')->exists($path)) {
        abort(404);
    }

    $file = Storage::disk('local')->get($path);
    $mime = Storage::disk('local')->mimeType($path);

    return new Response($file, 200, ['Content-Type' => $mime]);
}
}

```

Save the file. This controller has four methods:

- `showForm()`: Displays the upload form.
- `handleUpload()`: Handles the image upload and stores it in the `private-images` directory in local storage.
- `showResult()`: Displays the uploaded image and its metadata.
- `viewImage()`: Streams the image file securely to the browser.

9.8.4.4 Step 4: Define Routes

We need to define routes for our controller methods. Open `routes/web.php` and add the following code:

```
use App\Http\Controllers\PrivateImageUploadController;

Route::get('/private/upload', [PrivateImageUploadController::class, 'showForm'])->name('private.upload.form');
Route::post('/private/upload', [PrivateImageUploadController::class, 'handleUpload'])->name('private.upload');
Route::get('/private/result', [PrivateImageUploadController::class, 'showResult'])->name('private.result');
Route::get('/private/image/{filename}', [PrivateImageUploadController::class, 'viewImage']);
```

9.8.4.5 Step 5: Create Views

We will create two views: one for the upload form and another for displaying the uploaded image and its metadata.

Create a new directory named `layouts` in `resources/views` and create a file named `app.blade.php`.

Codes for `resources/views/layouts/app.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Laravel Private Image Upload</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="bg-light">
    <div class="container mt-5">
        @yield('content')
    </div>
</body>
</html>
```

We also create `upload.blade.php` and `result.blade.php` in the `resources/views` directory.

Codes for `resources/views/upload.blade.php`:

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

@section('content')
<h2>Upload Image to Private Storage</h2>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul class="mb-0">
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
```

```

        </div>
    @endif

    <form method="POST" action="{{ route('private.handle') }}" enctype="multipart/form-data">
        @csrf
        <div class="mb-3">
            <label class="form-label">File Name</label>
            <input type="text" name="file_name" class="form-control" required>
        </div>
        <div class="mb-3">
            <label class="form-label">Description</label>
            <textarea name="description" class="form-control" rows="3" required></textarea>
        </div>
        <div class="mb-3">
            <label class="form-label">Choose Image</label>
            <input type="file" name="image" class="form-control" required>
        </div>
        <button class="btn btn-primary">Upload</button>
    </form>
    @endsection

```

Codes for resources/views/result.blade.php:

```

@extends('layouts.app')

@section('content')
<h2>Uploaded Image (Private Access)</h2>

<div class="card">
    
    <div class="card-body">
        <h5 class="card-title">{{ $file_name }}</h5>
        <p class="card-text">{{ $description }}</p>
        <a href="{{ route('private.form') }}" class="btn btn-secondary">Upload Another</a>
    </div>
</div>
@endsection

```

Save the files. The upload.blade.php file contains the upload form, while result.blade.php displays the uploaded image and its metadata.

9.8.4.6 Step 6: Run the Application

After completing the above steps, you can run the Laravel development server to test the application. Run the following command in your terminal:

```
| php artisan serve
```

Visit the upload form:

<http://localhost:8000/private/upload>

Laravel Private Image Upload

File Name
Private file 1

Description
Private file 1 description

Choose Image
Choose File img1.png

Upload

Figure 9.10: Upload form for private storage.

Fill in the form and upload an image. After submitting, you should be redirected to a new page displaying the uploaded image and its metadata.

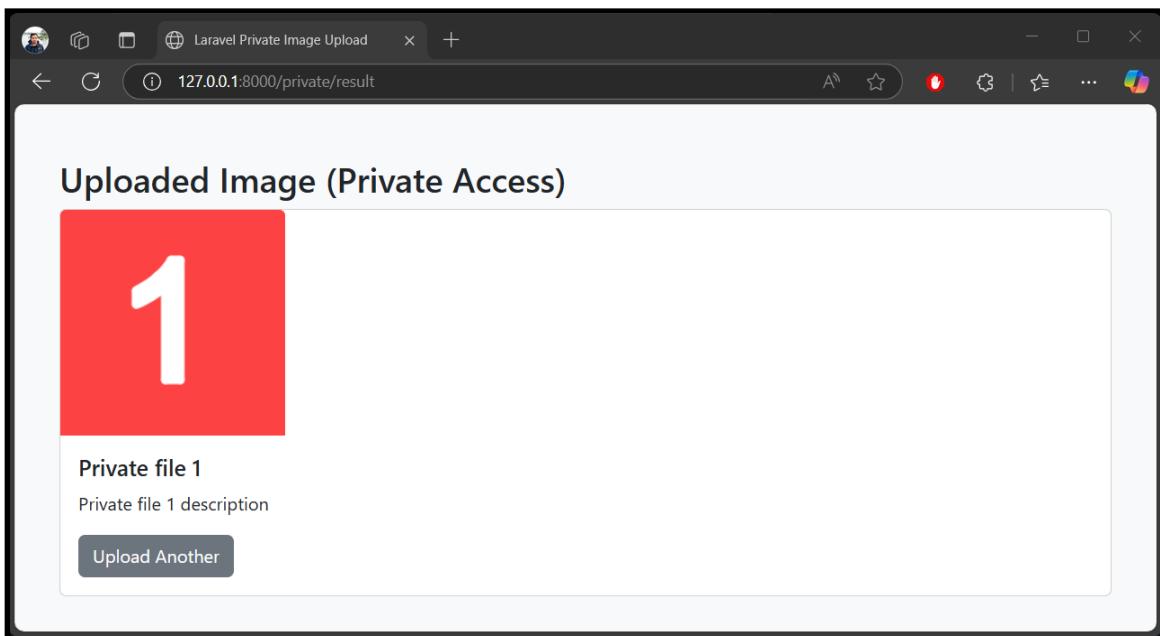


Figure 9.11: Displaying image from private storage.

9.8.5 Summary

In this lab, you:

- Uploaded images securely to **private disk**
- Used **controller streaming** to display private files
- Avoided database use by passing data via session
- Used Bootstrap for UI

This pattern is ideal for **secure downloads**, **restricted media**, or **authenticated access** to user files.

9.9 Exercise 27: Upload and Display Image to MinIO using Signed URLs

9.9.1 Description

This lab shows how to generate **signed URLs** in Laravel 12 for uploading and accessing images stored in MinIO (S3-compatible). We will simulate a secure upload workflow without a database, using `Storage::temporaryUrl()` to give temporary access to the file.

9.9.2 Objectives

You will learn to:

- Configure Laravel for MinIO as an S3-compatible disk
- Generate signed URLs for uploading and downloading
- Upload files using HTTP PUT via JavaScript
- Stream image display securely using signed URL
- Use Bootstrap for styling

9.9.3 Prerequisites

- Laravel 12
- PHP \geq 8.2
- Laravel cli installed
- Visual Studio Code or any text editor
- Docker (for MinIO)
- Basic Laravel knowledge
- `php artisan serve` to run Laravel

If you don't have Docker installed, you can download it from Docker's official website, <https://www.docker.com/get-started>.

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

9.9.4 Steps

Here are the steps to implement the image upload feature using signed URLs with MinIO:

9.9.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project for this lab. Open your terminal and run:

```
| laravel new image-temp-minio  
| cd image-temp-minio  
| code .
```

You should see the Laravel project in your editor.

9.9.4.2 Step 2: Run MinIO Server with Docker

We will run MinIO using Docker. Open your terminal and run the following command:

```
| docker run -p 9000:9000 -p 9001:9001 \  
|   -e "MINIO_ROOT_USER=minioadmin" \  
|   -e "MINIO_ROOT_PASSWORD=minioadmin" \  
|   quay.io/minio/minio server /data --console-address ":9001"
```

If you have Windows, you can use the following command:

```
| docker run -p 9000:9000 -p 9001:9001 ^  
|   -e "MINIO_ROOT_USER=minioadmin" ^  
|   -e "MINIO_ROOT_PASSWORD=minioadmin" ^  
|   quay.io/minio/minio server /data --console-address ":9001"
```

MinIO Console: <http://localhost:9001>

Login:

- User: minioadmin
- Pass: minioadmin

Create a bucket: `private-uploads`

You can do this in the MinIO Console by clicking on **Buckets** and then **Create Bucket**. Name it `private-uploads`.

Generate an access key and secret key for your MinIO instance.

You can do this in the MinIO Console by clicking on **Access Keys** and then **Create Access Key**. Set name as `laravel-private`.

Keep the access key and secret key handy, as you will need them to configure your Laravel application.

9.9.4.3 Step 3: Laravel MinIO Configuration

Install the AWS S3 adapter:

```
| composer require league/flysystem-aws-s3-v3 "^3.0" --with-all-dependencies
```

We need to set up the environment variables for MinIO in the `.env` file. Open `.env` and add the following lines:

```
| FILESYSTEM_DISK=s3  
| AWS_ACCESS_KEY_ID=<your-access-key>  
| AWS_SECRET_ACCESS_KEY=<your-secret-key>  
| AWS_DEFAULT_REGION=us-east-1  
| AWS_BUCKET=private-uploads  
| AWS_URL=http://localhost:9000/local  
| AWS_ENDPOINT=http://localhost:9000  
| AWS_USE_PATH_STYLE_ENDPOINT=true
```

Change the `AWS_BUCKET` to the name of the bucket you created in MinIO (e.g., `private-uploads`). Change `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` to the access key and secret key you generated in MinIO.

Verify the `s3` disk configuration in `config/filesystems.php`:

```
| 's3' => [  
|   'driver' => 's3',  
|   'key' => env('AWS_ACCESS_KEY_ID'),  
|   'secret' => env('AWS_SECRET_ACCESS_KEY'),  
|   'region' => env('AWS_DEFAULT_REGION'),  
|   'bucket' => env('AWS_BUCKET'),  
|   'endpoint' => env('AWS_ENDPOINT'),  
|   'use_path_style_endpoint' => env('AWS_USE_PATH_STYLE_ENDPOINT', false),  
| ],
```

9.9.4.4 Step 4: Create Controller

We will create a controller to handle the image upload logic. Run the following command in your terminal:

```
| php artisan make:controller SignedImageUploadController
```

You can find the controller in `app/Http/Controllers/SignedImageUploadController.php`. This controller will handle the form display, image upload, and result display.

Edit `app/Http/Controllers/SignedImageUploadController.php`:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Str;

class SignedImageUploadController extends Controller
{
    public function showForm()
    {
        return view('upload_signed');
    }

    public function generateSignedUpload(Request $request)
    {
        $validated = $request->validate([
            'file_name' => 'required|string|max:255',
            'description' => 'required|string|max:1000',
        ]);

        $filename = Str::slug($validated['file_name']) . '-' . time() . '.png';
        $path = $filename;

        // Generate pre-signed PUT URL
        $client = Storage::disk('s3')->getClient();
        $bucket = config('filesystems.disks.s3.bucket');
        $command = $client->getCommand('PutObject', [
            'Bucket' => $bucket,
            'Key' => $path,
            'ContentType' => 'image/png'
        ]);
        $requestUrl = (string) $client->createPresignedRequest($command, '+5 minutes')->getUri();

        session()->put([
            'file_name' => $validated['file_name'],
            'description' => $validated['description'],
            'filename' => $filename
        ]);

        return response()->json([
            'upload_url' => $requestUrl
        ]);
    }
}
```

```

public function showResult()
{
    $filename = session('filename');

    if (!$filename) {
        return redirect()->route('signed.form');
    }

    $url = Storage::disk('s3')->temporaryUrl(
        $filename,
        now()->addMinutes(5)
    );

    return view('result_signed', [
        'file_name' => session('file_name'),
        'description' => session('description'),
        'image_url' => $url,
    ]);
}
}

```

Explanation of the controller methods:

- `showForm()`: Displays the upload form.
- `generateSignedUpload()`: Validates the input, generates a signed URL for uploading the image, and stores metadata in the session.
- `showResult()`: Displays the uploaded image and its metadata. It generates a temporary URL for the image using `Storage::temporaryUrl()`.

Save the file.

9.9.4.5 Step 5: Define Routes

We need to define routes for our controller methods. Open `routes/web.php` and add the following code:

```

use App\Http\Controllers\SignedImageUploadController;

Route::get('/signed/upload', [SignedImageUploadController::class, 'showForm'])->name('signed.upload');
Route::post('/signed/upload-url', [SignedImageUploadController::class, 'generateSignedUpload']);
Route::get('/signed/result', [SignedImageUploadController::class, 'showResult'])->name('signed.result');

```

This code defines three routes:

- GET `/signed/upload`: Displays the upload form.
- POST `/signed/upload-url`: Generates a signed URL for uploading the image.
- GET `/signed/result`: Displays the uploaded image and its metadata.

9.9.4.6 Step 6: Create Views

We will create two views: one for the upload form and another for displaying the uploaded image and its metadata.

Create a new directory named `layouts` in `resources/views` and create a file named `app.blade.php`.

Codes for `resources/views/layouts/app.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Laravel Signed MinIO Upload</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="bg-light">
    <div class="container mt-5">
        @yield('content')
    </div>
</body>
</html>
```

We also create `upload_signed.blade.php` and `result_signed.blade.php` in the `resources/views` directory.

Codes for `resources/views/upload_signed.blade.php`:

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

@section('content')
<h2>Upload Image to MinIO (Signed URL)</h2>

<div id="alert" class="alert d-none"></div>

<form id="uploadForm">
    @csrf
    <div class="mb-3">
        <label class="form-label">File Name</label>
        <input type="text" name="file_name" id="file_name" class="form-control" required>
    </div>
    <div class="mb-3">
        <label class="form-label">Description</label>
        <textarea name="description" id="description" class="form-control" rows="3" required></textarea>
    </div>
    <div class="mb-3">
        <label class="form-label">Choose Image (PNG only)</label>
        <input type="file" id="image" accept="image/png" class="form-control" required>
    </div>
    <button type="submit" class="btn btn-primary">Upload</button>
</form>

<script>
    const form = document.getElementById('uploadForm');
    const alertBox = document.getElementById('alert');
```

```

form.addEventListener('submit', async function (e) {
    e.preventDefault();

    const file = document.getElementById('image').files[0];
    const file_name = document.getElementById('file_name').value;
    const description = document.getElementById('description').value;

    const response = await fetch('{{ route('signed.upload.url') }}', {
        method: 'POST',
        headers: {
            'X-CSRF-TOKEN': '{{ csrf_token() }}',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({ file_name, description })
    });

    const data = await response.json();

    if (data.upload_url) {
        const uploadResp = await fetch(data.upload_url, {
            method: 'PUT',
            headers: {
                'Content-Type': 'image/png'
            },
            body: file
        });

        if (uploadResp.ok) {
            window.location.href = "{{ route('signed.result') }}";
        } else {
            alertBox.textContent = "Upload failed.";
            alertBox.className = "alert alert-danger";
        }
    }
});

</script>
@endsection

```

Explanation of the JavaScript code:

- The form submission is intercepted to prevent the default behavior.
- A POST request is made to the server to get a signed URL for uploading the image.
- Once the signed URL is received, a PUT request is made to upload the image to that URL.
- If the upload is successful, the user is redirected to the result page.
- If the upload fails, an error message is displayed.
- The `alert` div is used to show success or error messages.

Codes for `resources/views/result_signed.blade.php`:

```

@extends('layouts.app')

@section('content')
<h2>Uploaded Image via Signed URL</h2>

<div class="card">

```

```

    
    <div class="card-body">
        <h5 class="card-title">{{ $file_name }}</h5>
        <p class="card-text">{{ $description }}</p>
        <a href="{{ route('signed.form') }}" class="btn btn-secondary">Upload Another</a>
    </div>
</div>
@endsection

```

Save the files. The `upload_signed.blade.php` file contains the upload form, while `result_signed.blade.php` displays the uploaded image and its metadata.

9.9.4.7 Step 7: Run Laravel Server

After completing the above steps, you can run the Laravel development server to test the application. Run the following command in your terminal:

```
| php artisan serve
```

Go to:

`http://localhost:8000/signed/upload`

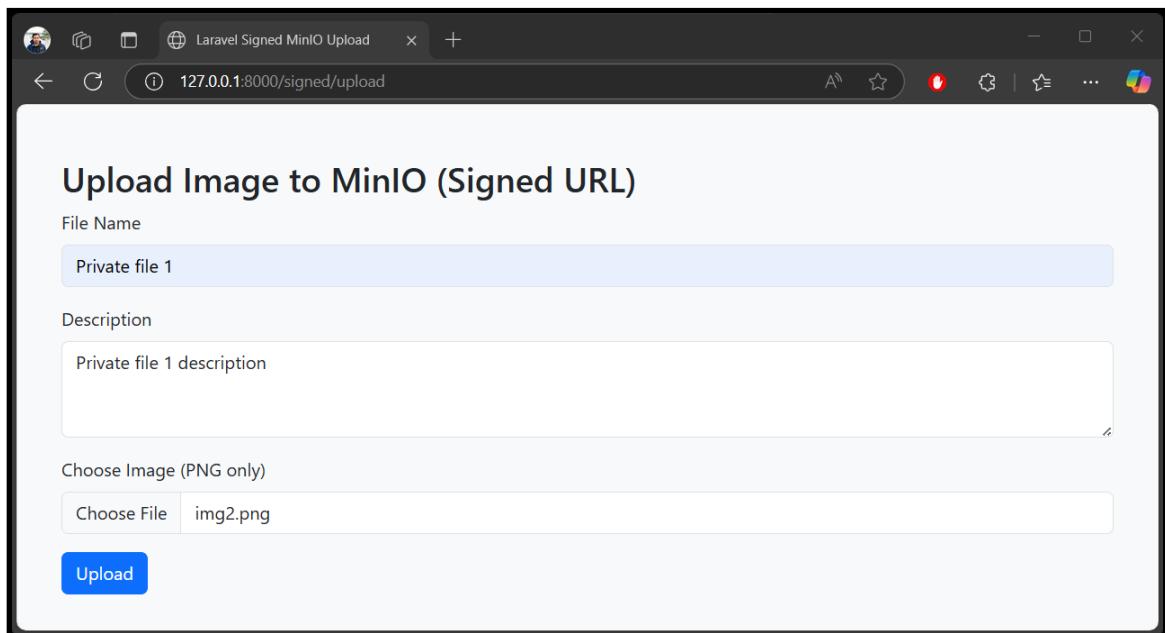


Figure 9.12: Upload form for signed URL.

Upload a JPG image and see it displayed securely using a **signed URL** (temporary access).

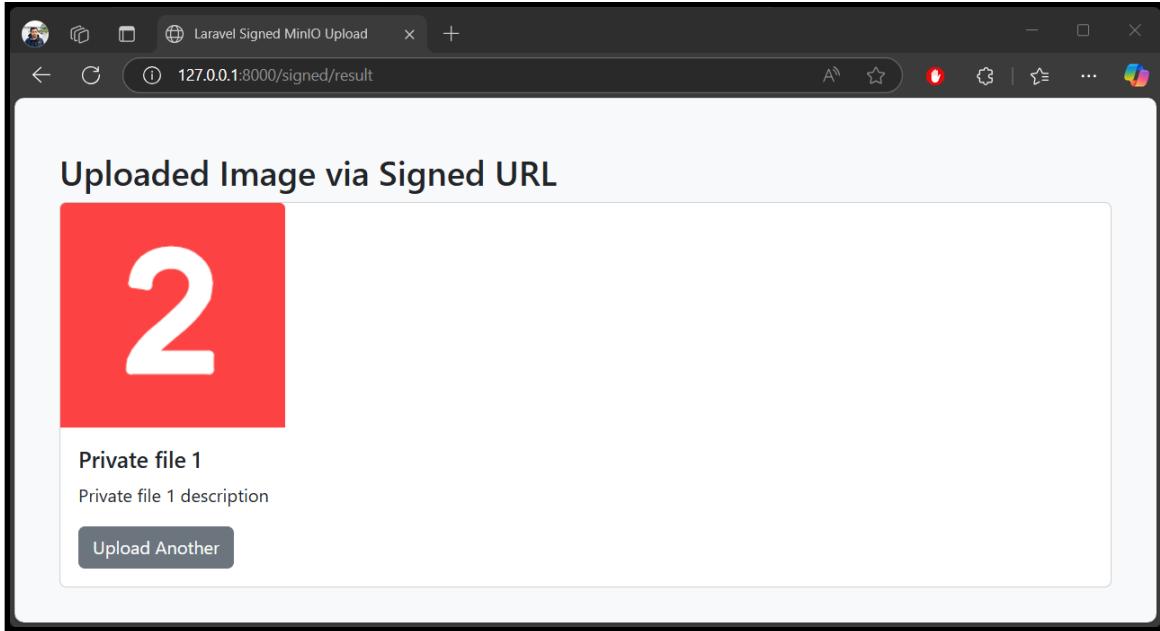


Figure 9.13: Displaying image from signed URL.

You can also check the uploaded image in the MinIO Console under the `private-uploads` bucket.

9.9.5 Summary

In this lab, you:

- Configured Laravel 12 with MinIO as an S3-compatible backend
- Generated **pre-signed URLs** for secure upload and download
- Used **JavaScript + fetch API** for uploading via signed PUT request
- Rendered the uploaded image using a temporary signed GET URL

9.10 Conclusion

In this chapter, we covered various methods for uploading and displaying images in Laravel 12. We explored how to use local storage, MinIO (S3-compatible), and signed URLs for secure uploads. Each method has its own advantages and use cases, allowing you to choose the best approach for your application.

We also learned how to use Bootstrap for styling the upload forms and result pages, making our application visually appealing and user-friendly.

10 Database and Eloquent ORM

Laravel 12 provides a robust database abstraction layer with support for various database systems such as MySQL, PostgreSQL, SQLite, and SQL Server. It includes migrations for schema management, seeders for populating initial data, and Eloquent ORM for interacting with databases in an object-oriented manner. This chapter will cover essential topics related to Laravel's database system.

10.1 Laravel Database Configuration

Before using Laravel's database features, you need to configure the database connection. Laravel's configuration file for database connections is located at:

```
| config/database.php
```

However, the easiest way to configure your database is by updating the `.env` file. Here's an example configuration for a MySQL database:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=myapp
DB_USERNAME=root
DB_PASSWORD=secret
```

Once the database credentials are set, you can verify the connection by running the following Artisan command:

```
| php artisan migrate:status
```

If Laravel connects to the database successfully, you will see a list of migration statuses.

Note: Laravel 12 uses SQLite as the default database. If you want to use a different database, you need to install the corresponding PHP extension and update the `.env` file with the appropriate database credentials. Make sure database drivers are enabled in your PHP configuration.

10.2 Migrations and Seeders

In this section, we will cover Laravel's migration and seeder features, which help manage database schemas and seed initial data.

10.2.1 Migrations

Migrations allow you to define database schemas using PHP code, making schema version control easy. To create a migration, run:

```
| php artisan make:migration create_users_table
```

This will generate a migration file in the `database/migrations/` directory. Open the file and define the table structure:

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

return new class extends Migration {
    public function up(): void
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password');
            $table->timestamps();
        });
    }

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

Run the migration to create the table:

```
| php artisan migrate
```

10.2.2 Seeders

Seeders are used to populate the database with test or initial data. To create a seeder:

```
| php artisan make:seeder UserSeeder
```

Open the generated file inside `database/seeders/UserSeeder.php` and define the logic:

```
use Illuminate\Database\Seeder;
use App\Models\User;
use Illuminate\Support\Facades\Hash;

class UserSeeder extends Seeder
{
    public function run(): void
```

```
| {
|     User::create([
|         'name' => 'Adi Siauw',
|         'email' => 'adi@ilmudata.id',
|         'password' => Hash::make('password'),
|     ]);
| }
}
```

To run the seeder:

```
| php artisan db:seed --class=UserSeeder
```

You can also run all seeders at once using:

```
| php artisan db:seed
```

10.3 Working with Eloquent ORM

Eloquent ORM is Laravel's built-in Object-Relational Mapping (ORM) system, allowing database interaction using models instead of writing raw SQL queries.

10.3.1 Creating an Eloquent Model

To generate a model for the `users` table, use:

```
| php artisan make:model User
```

This will create `app/Models/User.php`, which looks like:

```
| namespace App\Models;

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

class User extends Model
{
    use HasFactory;

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

10.3.2 Basic Eloquent Operations

We will cover some basic Eloquent operations.

With Eloquent, you can easily retrieve data from the database:

```
| $users = User::all(); // Get all users
| $user = User::find(1); // Find user by ID
```

We can create a new record using the `create` method:

```
User::create([
    'name' => 'Adi Siauw',
    'email' => 'adi@ilmudata.id',
    'password' => Hash::make('password'),
]);
```

We also have the `save` method to update a record:

```
$user = User::find(1);
$user->name = 'Updated Name';
$user->save();
```

Last, if you want to delete a record, you can use the `delete` method:

```
User::destroy(1); // Delete user with ID 1
```

10.4 Query Builder vs. Eloquent ORM

Laravel provides two primary ways to interact with databases: **Query Builder** and **Eloquent ORM**.

10.4.1 Query Builder

Query Builder provides a fluent interface for building SQL queries:

```
$users = DB::table('users')->where('email', 'adi@ilmudata.id')->get();
```

10.4.2 Eloquent ORM

Eloquent ORM is an object-oriented way to work with databases:

```
$users = User::where('email', 'adi@ilmudata.id')->get();
```

10.4.3 Comparison

Here's a comparison between Query Builder and Eloquent ORM:

Feature	Query Builder	Eloquent ORM
Performance	Faster	Slightly slower (due to object mapping)
Readability	SQL-like	More readable, object-oriented

Feature	Query Builder	Eloquent ORM
Flexibility	More control	Works well with Laravel features
Relationship Handling	Requires joins	Supports relationships natively

Eloquent ORM is ideal for most cases where models represent database tables. However, for complex queries involving joins, aggregates, or large datasets, Query Builder may be a better choice.

We will explore Query Builder in the next chapter.

10.5 Exercise 28: Building a Simple Todo Web App with Eloquent ORM and SQLite

10.5.1 Description

In this hands-on lab, we will build a **simple CRUD Todo web application** using **Laravel 12, Eloquent ORM, and SQLite**. The app will allow users to **create, read, update, and delete** todo tasks.

We will use **Bootstrap** for styling and create a **migration and seeder** to populate initial data.

10.5.2 Objectives

By the end of this lab, you will:

- Set up a **Laravel 12** web app with **SQLite** as the database
- Use **Eloquent ORM** for **CRUD operations**
- Create a **migration and seeder** for the `todos` table
- Build a **TodoController** to handle CRUD logic
- Create **Blade** views with **Bootstrap styling**

10.5.3 Prerequisites

Before starting, ensure you have:

- **PHP 8.2+** and Laravel cli installed
- **SQLite installed** (by default in Laravel 12)

- Visual Studio Code or any code editor
- **Basic knowledge of Laravel**

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

10.5.4 SQLite Driver Notes

We need to ensure the SQLite driver is installed and enabled in your PHP configuration.

- **Windows:** SQLite is included with Laravel 12 but needs `sqlite3.dll` in `php.ini` (`extension=sqlite3`).
 - Download SQLite from <https://www.sqlite.org/download.html>
 - Download SQLite 32-bit DLL
 - Put `sqlite3.dll` in `ext` folder.
- **Linux/macOS:** SQLite is pre-installed, but you may need to install the PHP extension:

```
| sudo apt install php8.x-sqlite3 # Ubuntu/Debian  
| sudo dnf install php-sqlite3 # Fedora/CentOS
```

We also ensure `php.ini` has the SQLite extension enabled:

```
| extension=pdo_sqlite  
| extension=sqlite3
```

Save the file. Ok, we're ready to build our Todo app!

You may check chapter 1 for more details about installing SQLite and PHP.

10.5.5 Steps

Here are the steps to complete this lab:

10.5.5.1 Step 1: Create a New Laravel 12 Project

First, we need to create a new Laravel project. Open a terminal and create a new Laravel project:

```
| laravel new todo-app  
| cd todo-app  
| php artisan serve
```

Your app should now be running at <http://127.0.0.1:8000>. Then, close the server with **Ctrl+C**.

You can open the project in your favorite code editor, such as Visual Studio Code:

```
| code .
```

10.5.5.2 Step 2: Configure SQLite Database

Laravel 12 **uses SQLite by default**, but we need to ensure the correct setup.

1. View an SQLite database file, `database.sqlite`, in the `database` folder.
2. Open `.env` and update the database settings:

```
| DB_CONNECTION=sqlite  
| DB_DATABASE=database/database.sqlite
```

3. Clear the config cache:

```
| php artisan config:clear
```

10.5.5.3 Step 3: Create a Migration for the `todos` Table

We will create a migration file to define the `todos` table structure.

Run the migration command:

```
| php artisan make:migration create.todos_table
```

You will see a new file in the `database/migrations/` directory. Open the generated file in `database/migrations/YYYY_MM_DD_create.todos_table.php` and update it:

```
<?php  
use Illuminate\Database\Migrations\Migration;  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Support\Facades\Schema;  
  
return new class extends Migration {  
    public function up()  
    {  
        Schema::create('todos', function (Blueprint $table) {  
            $table->id();  
            $table->string('task');  
            $table->boolean('completed')->default(false);  
            $table->timestamps();  
        });  
    }  
}
```

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

This code defines the `todos` table with the following columns:

- `id`: Primary key
- `task`: String column for the task name
- `completed`: Boolean column to indicate if the task is completed

After editing the migration file, we need to run the migration to create the `todos` table in the SQLite database.

Run the migration:

```
| php artisan migrate
```

You should see a message indicating that the migration was successful. You can check the `database/database.sqlite` file to confirm that the `todos` table has been created.

10.5.5.4 Step 4: Create a Seeder for Dummy Data

We will create a seeder to populate the `todos` table with some initial data.

Run this command to create a seeder:

```
| php artisan make:seeder TodoSeeder
```

This will create a new seeder file in the `database/seeders` directory. Open the generated file in `database/seeders/TodoSeeder.php` and update it:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Carbon\Carbon;

class TodoSeeder extends Seeder
{
    public function run()
    {
        DB::table('todos')->insert([
            [
                'task' => 'Buy groceries',
                'completed' => false,
                'created_at' => Carbon::now(),
            ]
        ]);
    }
}
```

```

        'updated_at' => Carbon::now()
    ],
    [
        'task' => 'Buy fruits',
        'completed' => false,
        'created_at' => Carbon::now(),
        'updated_at' => Carbon::now()
    ],
    [
        'task' => 'Complete Laravel project',
        'completed' => true,
        'created_at' => Carbon::now(),
        'updated_at' => Carbon::now()
    ],
),
]);
}
}

```

This code defines a seeder that inserts three dummy tasks into the `todos` table.

Seed the database:

```
| php artisan db:seed --class=TodoSeeder
```

Now you can see the `todos` table with dummy data in the SQLite database, `database/database.sqlite`.

10.5.5 Step 5: Create the Todo Model

We will create a model for the `todos` table to interact with it using Eloquent ORM.

Run:

```
| php artisan make:model Todo
```

This will create a new model file in the `app/Models` directory. Open the generated file in `app/Models/Todo.php` and update it:

```

<?php

namespace App\Models;

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

class Todo extends Model
{
    use HasFactory;

    protected $fillable = ['task', 'completed'];
}

```

Save the file. This code defines the `Todo` model and specifies that the `task` and `completed` attributes are mass assignable.

10.5.5.6 Step 6: Create a TodoController for CRUD Operations

We will create a controller to handle CRUD operations for the `todos` table.

Run:

```
| php artisan make:controller TodoController
```

This will create a new controller file in the `app/Http\Controllers` directory. Open the generated file in `app/Http\Controllers/TodoController.php` and update it:

```
<?php

namespace App\Http\Controllers;

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

class TodoController extends Controller
{
    public function index()
    {
        $todos = Todo::all();
        return view('todos.index', compact('todos'));
    }

    public function create()
    {
        return view('todos.create');
    }

    public function store(Request $request)
    {
        $request->validate(['task' => 'required|string']);
        Todo::create(['task' => $request->task]);
        return redirect()->route('todos.index')->with('success', 'Task added successfully!');
    }

    public function show(Todo $todo)
    {
        return view('todos.show', compact('todo'));
    }

    public function edit(Todo $todo)
    {
        return view('todos.edit', compact('todo'));
    }

    public function update(Request $request, Todo $todo)
    {
        $request->validate(['task' => 'required|string']);
        $todo->update(['task' => $request->task]);
        return redirect()->route('todos.index')->with('success', 'Task updated successfully!');
    }

    public function destroy(Todo $todo)
    {
        $todo->delete();
        return redirect()->route('todos.index')->with('success', 'Task deleted successfully!');
    }
}
```

```
| } }
```

Explanation of the methods:

- `index()`: Fetches all todos and returns the index view.
- `create()`: Returns the create view for adding a new todo.
- `store()`: Validates and stores a new todo in the database.
- `show()`: Displays a single todo.
- `edit()`: Returns the edit view for updating an existing todo.
- `update()`: Validates and updates an existing todo in the database.
- `destroy()`: Deletes a todo from the database.

Save the file. This code defines a controller with methods for handling CRUD operations for the `todos` table.

10.5.5.7 Step 7: Define Web Routes

We need to define routes for the Todo app in the `routes/web.php` file.

Edit `routes/web.php`:

```
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\TodoController;

Route::get('/', [TodoController::class, 'index'])->name('todos.index');
Route::get('/todos/create', [TodoController::class, 'create'])->name('todos.create');
Route::post('/todos', [TodoController::class, 'store'])->name('todos.store');
Route::get('/todos/{todo}', [TodoController::class, 'show'])->name('todos.show');
Route::get('/todos/{todo}/edit', [TodoController::class, 'edit'])->name('todos.edit');
Route::patch('/todos/{todo}', [TodoController::class, 'update'])->name('todos.update');
Route::delete('/todos/{todo}', [TodoController::class, 'destroy'])->name('todos.destroy');
```

This code defines the routes for the Todo app, mapping URLs to the corresponding controller methods.

Please comment out the default route to avoid conflicts.

```
// Route::get('/', function () {
//     return view('welcome');
// });
```

Save the file. This code defines the routes for the Todo app, mapping URLs to the corresponding controller methods.

10.5.5.8 Step 8: Create Blade Views with Bootstrap

In this step, we will create some Blade views for the Todo app using **Bootstrap** for styling.

First, we create a **layout view** that will be used by all other views. This layout will include the Bootstrap CSS and a navigation bar.

Layout View

Create `layouts` folder in `resources/views` and create a new file `resources/views/layouts/app.blade.php`. This file will serve as the base template for all views.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@yield('title', 'Todo App')</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container mt-4">

    <h1 class="text-center mb-4">Laravel 12 Todo App</h1>

    @if(session('success'))
        <div class="alert alert-success">{{ session('success') }}</div>
    @endif

    <nav class="mb-3">
        <a href="{{ route('todos.index') }}" class="btn btn-primary">Todo List</a>
        <a href="{{ route('todos.create') }}" class="btn btn-success">Add New Task</a>
    </nav>

    @yield('content')

</body>
</html>
```

This code defines a layout with a title, Bootstrap CSS, and a navigation bar with links to the Todo list and create new task pages.

Todo Pages

Now we will create the views for the Todo app. Create a new folder `todos` in `resources/views` and create the following files:

- `index.blade.php`: This view will display the list of todos.
- `create.blade.php`: This view will display a form to add a new todo.
- `edit.blade.php`: This view will display a form to edit an existing todo.
- `show.blade.php`: This view will display a single todo.
- To delete a todo, we will use the `index.blade.php` view.

Create a new file `resources/views/todos/index.blade.php`. This view will **display a list of todos** include links to create, edit, update, and delete todos.

Add the following code:

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

@section('title', 'Todo List')

@section('content')
    <h2>Todo List</h2>

    <ul class="list-group">
        @foreach($todos as $todo)
            <li class="list-group-item d-flex justify-content-between align-items-center">
                {{ $todo->task }}
                <div>
                    <form action="{{ route('todos.show', $todo->id) }}" method="GET" class="d-inline">
                        <button type="submit" class="btn btn-info btn-sm">Details</button>
                    </form>
                    <form action="{{ route('todos.edit', $todo->id) }}" method="GET" class="d-inline">
                        <button type="submit" class="btn btn-warning btn-sm">Edit</button>
                    </form>
                    <form action="{{ route('todos.destroy', $todo->id) }}" method="POST" class="d-inline">
                        @csrf @method('DELETE')
                        <button class="btn btn-danger btn-sm">Delete</button>
                    </form>
                </div>
            </li>
        @endforeach
    </ul>
@endsection
```

This view displays a form to add a new todo. Create a new file

resources/views/todos/create.blade.php and add the following code:

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

@section('title', 'Create New Task')

@section('content')
    <h2>Create New Task</h2>

    <form action="{{ route('todos.store') }}" method="POST" class="mt-3">
        @csrf
        <div class="mb-3">
            <label for="task" class="form-label">Task Name</label>
            <input type="text" name="task" id="task" class="form-control" required>
        </div>
        <button type="submit" class="btn btn-success">Add Task</button>
        <a href="{{ route('todos.index') }}" class="btn btn-secondary">Back to List</a>
    </form>
@endsection
```

This view displays a form to edit an existing todo. We create a new file

resources/views/todos/edit.blade.php and add the following code:

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

@section('title', 'Edit Task')

@section('content')
    <h2>Edit Task</h2>
```

```

    <form action="{{ route('todos.update', $todo->id) }}" method="POST" class="mt-3">
        @csrf @method('PATCH')
        <div class="mb-3">
            <label for="task" class="form-label">Task Name</label>
            <input type="text" name="task" id="task" class="form-control" value="{{ $todo->task }}"
        }" required>
        </div>
        <button type="submit" class="btn btn-warning">Update Task</button>
        <a href="{{ route('todos.index') }}" class="btn btn-secondary">Back to List</a>
    </form>

```

@endsection

The **delete button** is inside `index.blade.php` and uses a form:

```

<form action="{{ route('todos.destroy', $todo->id) }}" method="POST" style="display:inline;">
    @csrf @method('DELETE')
    <button class="btn btn-sm btn-danger">Delete</button>
</form>

```

Finally, we create a new file `resources/views/todos/show.blade.php` and add the following code:

```

@extends('layouts.app')
@section('title', 'Task Details')
@section('content')
    <h2>Task Details</h2>

    <div class="card mt-3">
        <div class="card-body">
            <h5 class="card-title">{{ $todo->task }}</h5>
            <p class="card-text">Status: {{ $todo->completed ? 'Completed' : 'Not Completed' }}</p>
        </div>
    </div>
    <a href="{{ route('todos.edit', $todo->id) }}" class="btn btn-warning">Edit</a>
    <a href="{{ route('todos.index') }}" class="btn btn-secondary">Back to List</a>

```

@endsection

Save all the files. This code defines the views for the Todo app, including the layout and individual views for listing, creating, and editing todos.

10.5.5.9 Testing the Application

After completing all the steps, we can now test the application.

Run:

```
| php artisan serve
```

Visit:

```
| http://127.0.0.1:8000
```

You should see the Todo list page with initial data populated from the seeder. You can add, edit, and delete todos using the buttons provided.

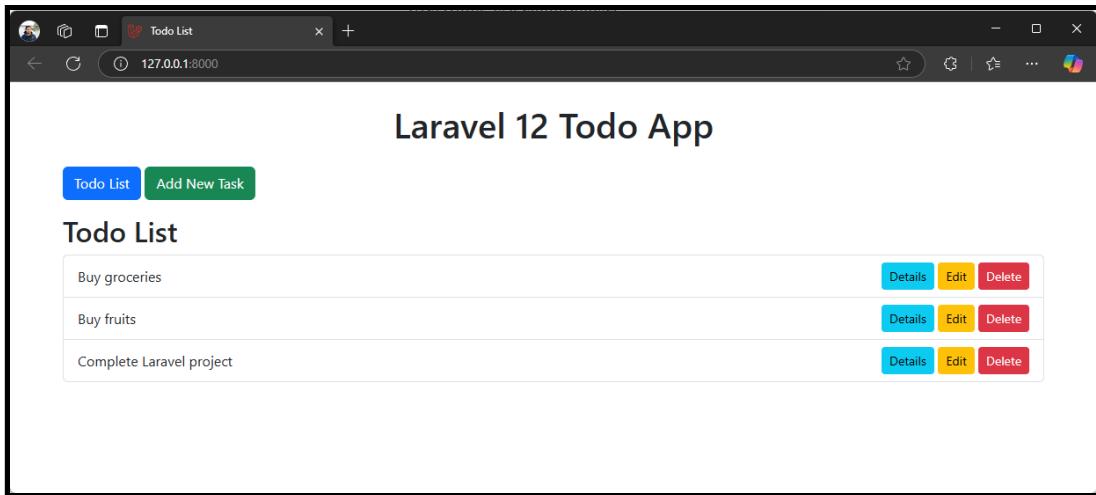


Figure 10.1: Todo List.

10.5.6 Test Features

We can now test the following features:

- Open <http://127.0.0.1:8000> to view the Todo list
- Click **Add New Task** to create a new task

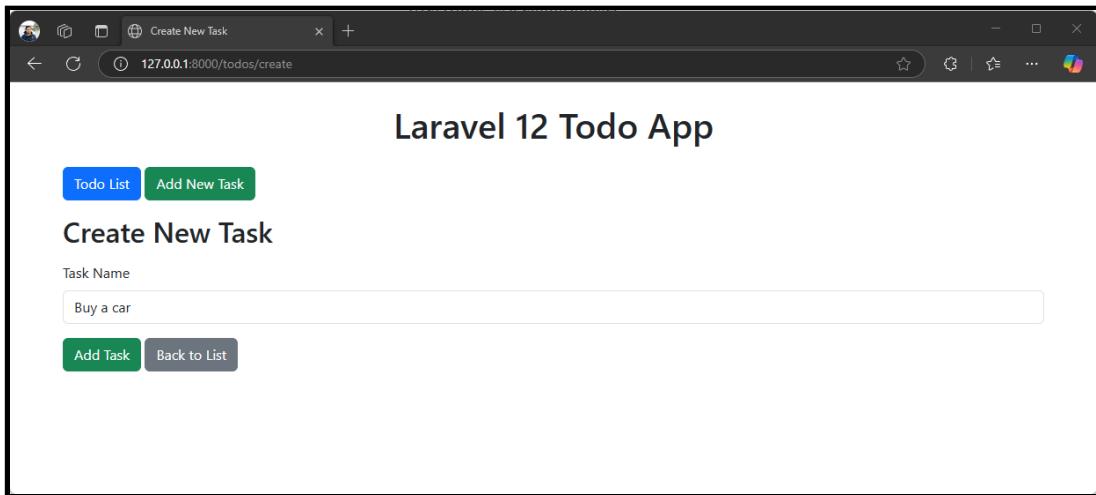


Figure 10.2: Add new task.

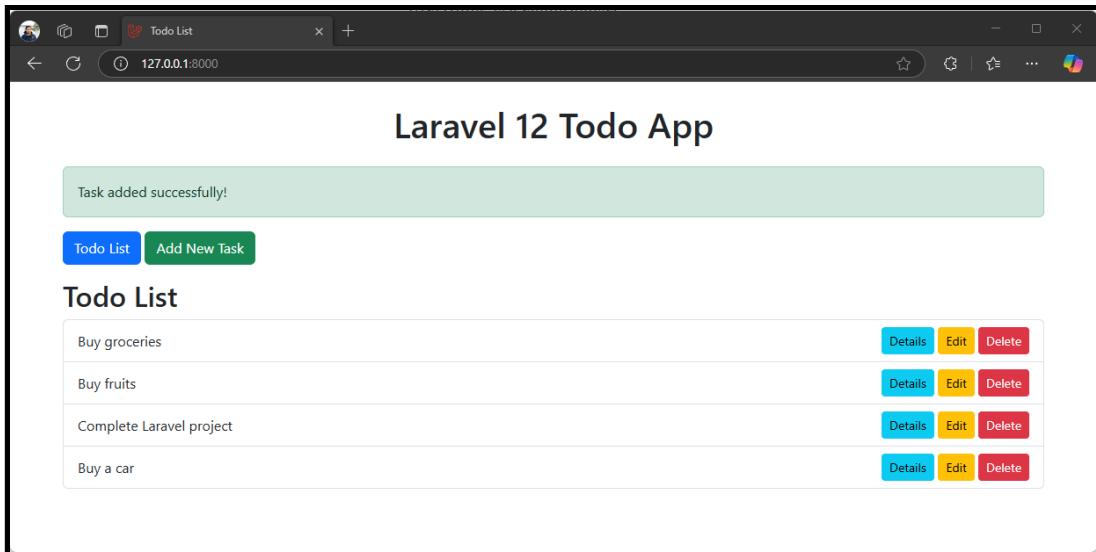


Figure 10.3: A new list.

- Click **Details** to update a task

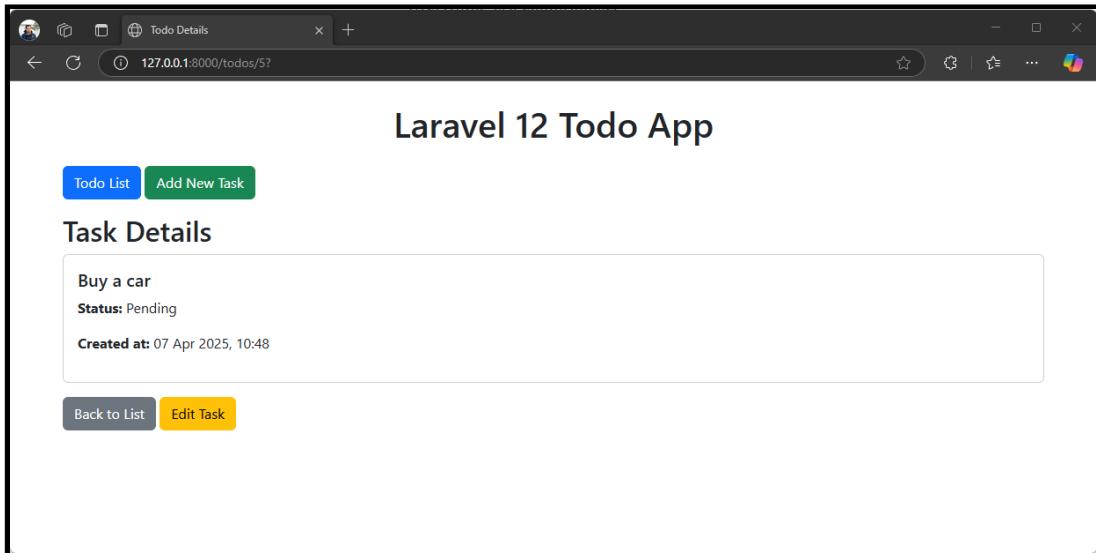


Figure 10.4: A detail of task.

- Click **Edit** to update a task

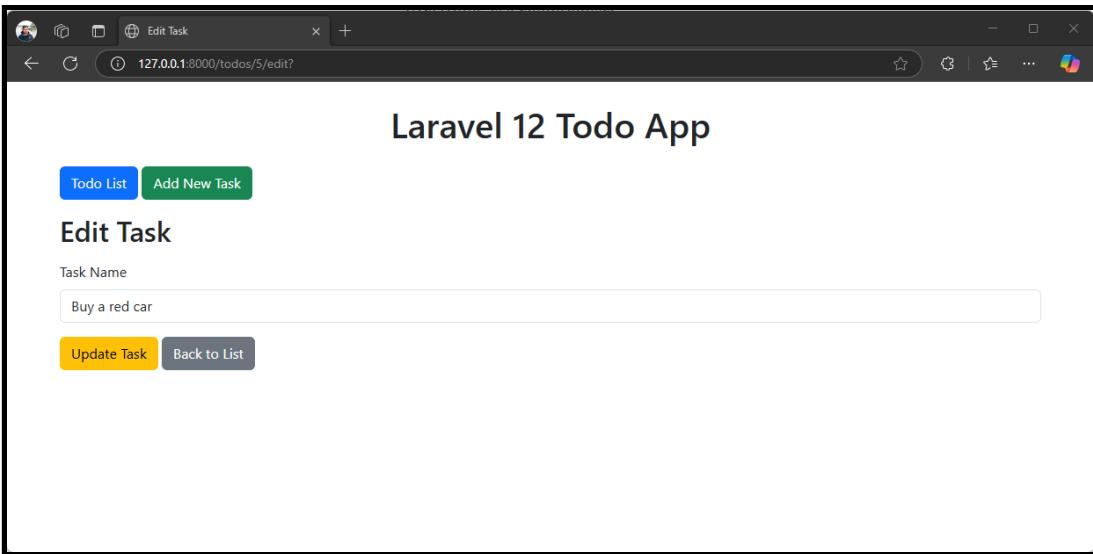


Figure 10.5: Edit task.

- Click **Delete** to remove a task

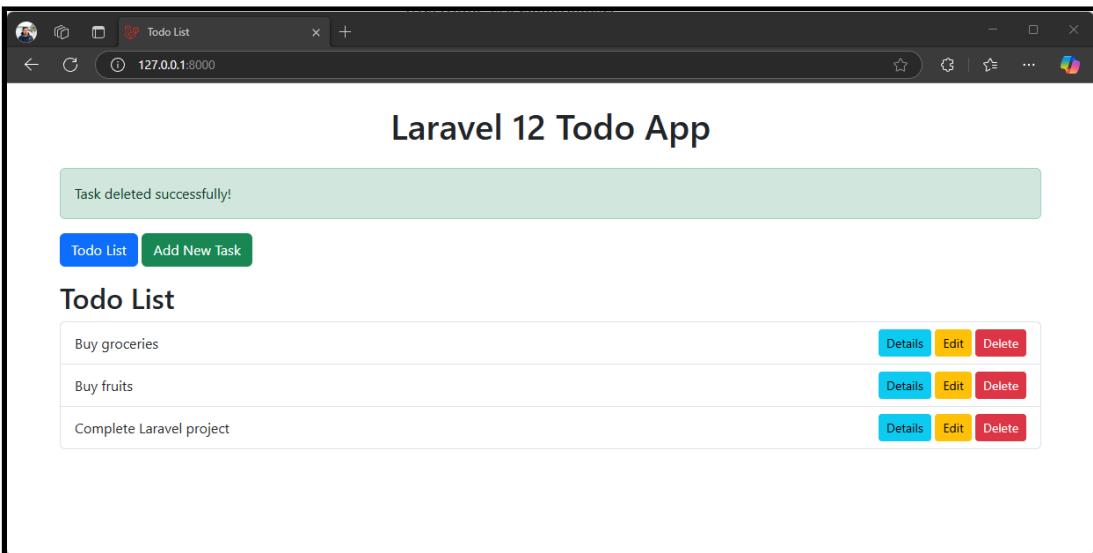


Figure 10.6: Delete task.

That's it! You've successfully built a simple Todo web app using Laravel 12, Eloquent ORM, and SQLite.

10.5.7 Summary

In this hands-on lab, you have successfully:

- Created a new Laravel 12 project with SQLite as the database
- Configured the database connection in the `.env` file

- Created a migration for the `todos` table
- Created a seeder to populate the `todos` table with initial data
- Created a model for the `todos` table
- Created a controller to handle CRUD operations for the `todos` table
- Defined web routes for the Todo app
- Created Blade views for the Todo app using Bootstrap for styling
- Tested the application by adding, editing, and deleting todos

10.6 Exercise 29: Building a Simple Todo Web App with Laravel 12, Eloquent ORM, and MySQL

10.6.1 Description

In this hands-on lab, we will build a **simple CRUD Todo web application** using **Laravel 12, Eloquent ORM, and MySQL**. The app will allow users to **create, read, update, and delete** todo tasks.

Laravel 12 **uses SQLite by default**, so we will configure it to use **MySQL**. We will also create a **migration and seeder** to populate initial data and use **Bootstrap** for styling.

This lab is extension of the previous lab, where we built a Todo web app using **SQLite**. We will now switch to **MySQL** and add more features to the app.

10.6.2 Objectives

By the end of this lab, you will:

- Set up a **Laravel 12** web app with **MySQL** as the database
- Use **Eloquent ORM** for **CRUD operations**
- Create a **migration and seeder** for the `todos` table
- Build a **TodoController** to handle CRUD logic
- Create **Blade** views with **Bootstrap styling**

10.6.3 Prerequisites

Before starting, ensure you have:

- **PHP 8.2+** and Laravel cli installed
- **MySQL 8.4+** installed and running
- Visual Studio Code or any code editor
- Basic knowledge of Laravel

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

10.6.4 MySQL Driver Notes

We need to ensure the MySQL driver is installed and enabled in your PHP configuration.

- **Windows:** Ensure mysql extensions are enabled in `php.ini`.

```
| extension=mysqli  
| extension=pdo_mysql
```

- **Linux/macOS:** Install MySQL and PHP MySQL extension:

```
| sudo apt install mysql-server php8.x-mysql # Ubuntu/Debian  
| sudo dnf install mysql-server php-mysql      # Fedora/CentOS  
| brew install mysql                         # macOS (Homebrew)
```

10.6.5 Steps

Here are the steps to complete this lab:

10.6.5.1 Step 1: Create a New Laravel 12 Project

Open a terminal and create a new Laravel project:

```
| laravel new todo-mysql  
| cd todo-mysql  
| code .
```

While creating the project, you may be prompted to select a database server and perform migrations. You select **MySQL**. You can skip a migration step for now because we will create our own migration later with MySQL.

```

Command Prompt - laravel n × + ▾
26/110 [=====→] 23%
37/110 [=====→-----] 33%
47/110 [=====→-----] 42%
59/110 [=====→-----] 53%
66/110 [=====→-----] 60%
80/110 [=====→-----] 72%
90/110 [=====→-----] 81%
100/110 [=====→-----] 90%
110/110 [=====→-----] 100%
66 package suggestions were added by new dependencies, use `composer suggest` to see details.
Generating optimized autoload files
  88 packages you are using are looking for funding.
  Use the `composer fund` command to find out more!
  No security vulnerability advisories found.
  > php -r "file_exists('.env') || copy('.env.example', '.env');"

INFO Application key set successfully.

Which database will your application use? [SQLite]:
[sqlite] SQLite
[mysql] MySQL
[mariadb] MariaDB
[pgsql] PostgreSQL (Missing PDO extension)
[sqlsrv] SQL Server (Missing PDO extension)
> mysql

Default database updated. Would you like to run the default database migrations? (yes/no) [yes]:
> no

```

Figure 10.7: Create a new Laravel project with MySQL.

You should see a new folder named `todo-mysql` created. Open the project in your favorite code editor, such as Visual Studio Code.

10.6.5.2 Step 2: Configure MySQL Database

Laravel 12 **uses SQLite by default**, so we need to change it to **MySQL**.

1. Open **MySQL CLI**, or **MySQL Workbench** or **DBeaver** and create a database:

```
| CREATE DATABASE tododb;
```

You may assign a user to the database. For example, if you are using MySQL Workbench/DBeaver, you can create a new user and assign it to the `tododb` database. Don't use `root` user for security reasons.

2. Open `.env` and update the database settings:

```
| DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=tododb
DB_USERNAME=username
DB_PASSWORD=yourpassword
```

Change `username` and `yourpassword` to your MySQL username and password.

3. Clear the config cache:

```
| php artisan config:clear
```

10.6.5.3 Step 3: Create a Migration for the todos Table

We will create a migration file to define the `todos` table structure.

Run the migration command:

```
| php artisan make:migration create.todos_table
```

Open the generated file in `database/migrations/YYYY_MM_DD_create.todos_table.php` and update it:

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    public function up()
    {
        Schema::create('todos', function (Blueprint $table) {
            $table->id();
            $table->string('task');
            $table->boolean('completed')->default(false);
            $table->timestamps();
        });
    }

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

This code defines the `todos` table with the following columns:

- `id`: Primary key
- `task`: String column for the task name
- `completed`: Boolean column to indicate if the task is completed

After editing the migration file, we need to run the migration to create the `todos` table in the SQLite database.

Run the migration:

```
| php artisan migrate
```

You should see a message indicating that the migration was successful. You can check the `tododb` database to confirm that the `todos` table has been created.

10.6.5.4 Step 4: Create a Seeder for Dummy Data

We will create a seeder to populate the `todos` table with some initial data.

Run this command to create a seeder:

```
| php artisan make:seeder TodoSeeder
```

This will create a new seeder file in the `database/seeders` directory. Open the generated file in `database/seeders/TodoSeeder.php` and update it:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Carbon\Carbon;

class TodoSeeder extends Seeder
{
    public function run()
    {
        DB::table('todos')->insert([
            [
                'task' => 'Buy groceries',
                'completed' => false,
                'created_at' => Carbon::now(),
                'updated_at' => Carbon::now()
            ],
            [
                'task' => 'Buy fruits',
                'completed' => false,
                'created_at' => Carbon::now(),
                'updated_at' => Carbon::now()
            ],
            [
                'task' => 'Complete Laravel project',
                'completed' => true,
                'created_at' => Carbon::now(),
                'updated_at' => Carbon::now()
            ],
        ]);
    }
}
```

This code defines a seeder that inserts three dummy tasks into the `todos` table.

Seed the database:

```
| php artisan db:seed --class=TodoSeeder
```

Now you can see the `todos` table with dummy data in the MySQL database, `tododb`.

10.6.5.5 Step 5: Create the Todo Model

We will create a model for the `todos` table to interact with it using Eloquent ORM.

Run:

```
| php artisan make:model Todo
```

This will create a new model file in the `app/Models` directory. Open the generated file in `app/Models/Todo.php` and update it:

```
<?php

namespace App\Models;

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

class Todo extends Model
{
    use HasFactory;

    protected $fillable = ['task', 'completed'];
}
```

Save the file. This code defines the `Todo` model and specifies that the `task` and `completed` attributes are mass assignable.

10.6.5.6 Step 6: Create a TodoController for CRUD Operations

We will create a controller to handle CRUD operations for the `todos` table.

Run:

```
| php artisan make:controller TodoController
```

This will create a new controller file in the `app/Http/Controllers` directory. Open the generated file in `app/Http/Controllers/TodoController.php` and update it:

```
<?php

namespace App\Http\Controllers;

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

class TodoController extends Controller
{
    public function index()
    {
        $todos = Todo::all();
        return view('todos.index', compact('todos'));
    }

    public function create()
    {
        return view('todos.create');
    }
}
```

```

public function store(Request $request)
{
    $request->validate(['task' => 'required|string']);
    Todo::create(['task' => $request->task]);
    return redirect()->route('todos.index')->with('success', 'Task added successfully!');
}

public function show(Todo $todo)
{
    return view('todos.show', compact('todo'));
}

public function edit(Todo $todo)
{
    return view('todos.edit', compact('todo'));
}

public function update(Request $request, Todo $todo)
{
    $request->validate(['task' => 'required|string']);
    $todo->update(['task' => $request->task]);
    return redirect()->route('todos.index')->with('success', 'Task updated successfully!');
}

public function destroy(Todo $todo)
{
    $todo->delete();
    return redirect()->route('todos.index')->with('success', 'Task deleted successfully!');
}
}

```

Explanation of the methods:

- `index()`: Fetches all todos and returns the index view.
- `create()`: Returns the create view for adding a new todo.
- `store()`: Validates and stores a new todo in the database.
- `show()`: Displays a single todo.
- `edit()`: Returns the edit view for updating an existing todo.
- `update()`: Validates and updates an existing todo in the database.
- `destroy()`: Deletes a todo from the database.

Save the file. This code defines a controller with methods for handling CRUD operations for the `todos` table.

10.6.5.7 Step 7: Define Web Routes

We need to define routes for the Todo app in the `routes/web.php` file.

Edit `routes/web.php`:

```

use Illuminate\Support\Facades\Route;
use App\Http\Controllers\TodoController;

```

```
Route::get('/', [TodoController::class, 'index'])->name('todos.index');
Route::get('/todos/create', [TodoController::class, 'create'])->name('todos.create');
Route::post('/todos', [TodoController::class, 'store'])->name('todos.store');
Route::get('/todos/{todo}', [TodoController::class, 'show'])->name('todos.show');
Route::get('/todos/{todo}/edit', [TodoController::class, 'edit'])->name('todos.edit');
Route::patch('/todos/{todo}', [TodoController::class, 'update'])->name('todos.update');
Route::delete('/todos/{todo}', [TodoController::class, 'destroy'])->name('todos.destroy');
```

This code defines the routes for the Todo app, mapping URLs to the corresponding controller methods.

Please comment out the default route to avoid conflicts.

```
// Route::get('/', function () {
//     return view('welcome');
//});
```

Save the file. This code defines the routes for the Todo app, mapping URLs to the corresponding controller methods.

10.6.5.8 Step 8: Create Blade Views with Bootstrap

In this step, we will create some Blade views for the Todo app using **Bootstrap** for styling.

First, we create a **layout view** that will be used by all other views. This layout will include the Bootstrap CSS and a navigation bar.

Layout View

Create `layouts` folder in `resources/views` and create a new file `resources/views/layouts/app.blade.php`. This file will serve as the base template for all views.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@yield('title', 'Todo App')</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body class="container mt-4">

    <h1 class="text-center mb-4">Laravel 12 Todo App</h1>

    @if(session('success'))
        <div class="alert alert-success">{{ session('success') }}</div>
    @endif

    <nav class="mb-3">
        <a href="{{ route('todos.index') }}" class="btn btn-primary">Todo List</a>
        <a href="{{ route('todos.create') }}" class="btn btn-success">Add New Task</a>
    </nav>
```

```
| @yield('content')  
|</body>  
|</html>
```

This code defines a layout with a title, Bootstrap CSS, and a navigation bar with links to the Todo list and create new task pages.

Todo Pages

Now we will create the views for the Todo app. Create a new folder `todos` in `resources/views` and create the following files:

- `index.blade.php`: This view will display the list of todos.
- `create.blade.php`: This view will display a form to add a new todo.
- `edit.blade.php`: This view will display a form to edit an existing todo.
- `show.blade.php`: This view will display a single todo.
- To delete a todo, we will use the `index.blade.php` view.

Create a new file `resources/views/todos/index.blade.php`. This view will **display a list of todos** include links to create, edit, update, and delete todos.

Add the following code:

```
| @extends('layouts.app')  
|  
| @section('title', 'Todo List')  
|  
| @section('content')  
|     <h2>Todo List</h2>  
|  
|     <ul class="list-group">  
|         @foreach($todos as $todo)  
|             <li class="list-group-item d-flex justify-content-between align-items-center">  
|                 {{ $todo->task }}  
|                 <div>  
|                     <form action="{{ route('todos.show', $todo->id) }}" method="GET" class="d-  
|                     inline">  
|                         <button type="submit" class="btn btn-info btn-sm">Details</button>  
|                     </form>  
|                     <form action="{{ route('todos.edit', $todo->id) }}" method="GET" class="d-  
|                     inline">  
|                         <button type="submit" class="btn btn-warning btn-sm">Edit</button>  
|                     </form>  
|                     <form action="{{ route('todos.destroy', $todo->id) }}" method="POST"  
|                     class="d-inline">  
|                         @csrf @method('DELETE')  
|                         <button class="btn btn-danger btn-sm">Delete</button>  
|                     </form>  
|                 </div>  
|             </li>  
|         @endforeach  
|     </ul>  
| @endsection
```

This view **displays a form to add a new todo**. Create a new file `resources/views/todos/create.blade.php` and add the following code:

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

@section('title', 'Create New Task')

@section('content')
    <h2>Create New Task</h2>

    <form action="{{ route('todos.store') }}" method="POST" class="mt-3">
        @csrf
        <div class="mb-3">
            <label for="task" class="form-label">Task Name</label>
            <input type="text" name="task" id="task" class="form-control" required>
        </div>
        <button type="submit" class="btn btn-success">Add Task</button>
        <a href="{{ route('todos.index') }}" class="btn btn-secondary">Back to List</a>
    </form>
@endsection
```

This view **displays a form to edit an existing todo**. We create a new file `resources/views/todos/edit.blade.php` and add the following code:

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

@section('title', 'Edit Task')

@section('content')
    <h2>Edit Task</h2>

    <form action="{{ route('todos.update', $todo->id) }}" method="POST" class="mt-3">
        @csrf @method('PATCH')
        <div class="mb-3">
            <label for="task" class="form-label">Task Name</label>
            <input type="text" name="task" id="task" class="form-control" value="{{ $todo->task }}"
} required>
        </div>
        <button type="submit" class="btn btn-warning">Update Task</button>
        <a href="{{ route('todos.index') }}" class="btn btn-secondary">Back to List</a>
    </form>
@endsection
```

The **delete button** is inside `index.blade.php` and uses a form:

```
<form action="{{ route('todos.destroy', $todo->id) }}" method="POST" style="display:inline;">
    @csrf @method('DELETE')
    <button class="btn btn-sm btn-danger">Delete</button>
</form>
```

Finally, we create a new file `resources/views/todos/show.blade.php` and add the following code:

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

@section('title', 'Task Details')
@section('content')
    <h2>Task Details</h2>

    <div class="card mt-3">
        <div class="card-body">
            <h5 class="card-title">{{ $todo->task }}</h5>
```

```
|     </p>
|     <p class="card-text">Status: {{ $todo->completed ? 'Completed' : 'Not Completed' }}</p>
|     <a href="{{ route('todos.edit', $todo->id) }}" class="btn btn-warning">Edit</a>
|     <a href="{{ route('todos.index') }}" class="btn btn-secondary">Back to List</a>
|   </div>
| </div>
@endsection
```

Save all the files. This code defines the views for the Todo app, including the layout and individual views for listing, creating, and editing todos.

10.6.5.9 Testing the Application

After completing all the steps, we can now test the application.

Run:

```
| php artisan serve
```

Visit:

```
| http://127.0.0.1:8000
```

You should see the Todo list page with initial data populated from the seeder. You can add, edit, and delete todos using the buttons provided.

You perform the following actions:

- Click **Add New Task** to create a new task
- Click **Details** to view a task
- Click **Edit** to update a task
- Click **Delete** to remove a task
- Click **Back to List** to return to the Todo list

This similar to the previous lab, but now we are using **MySQL** as the database.

10.6.6 Summary

In this hands-on lab, you have successfully:

- Created a new Laravel 12 project with MySQL as the database
- Configured the database connection in the `.env` file
- Created a migration for the `todos` table
- Created a seeder to populate the `todos` table with initial data
- Created a model for the `todos` table
- Created a controller to handle CRUD operations for the `todos` table
- Defined web routes for the Todo app

- Created Blade views for the Todo app using Bootstrap for styling
- Tested the application by adding, editing, and deleting todos
- Successfully built a simple Todo web app using Laravel 12, Eloquent ORM, and MySQL

10.7 Exercise 30: Eloquent ORM Relationships: One-to-One, One-to-Many, Many-to-Many

10.7.1 Description

In this hands-on lab, we will build a **Laravel 12 web application** that demonstrates different **Eloquent relationships** with **SQLite as the database**.

We will define a **complex schema** showcasing:

- **One-to-One** (User → Profile)
- **One-to-Many** (User → Posts)
- **Many-to-Many** (Posts ↔ Tags)

We will create **migrations**, **models**, **seeders**, **controllers**, and **views** to interact with these relationships. The frontend will use **Bootstrap** for styling.

10.7.2 Objectives

By the end of this lab, you will:

- Set up a **Laravel 12 web app** with **SQLite**
- Create **One-to-One**, **One-to-Many**, and **Many-to-Many relationships** using **Eloquent ORM**
- Populate the database using **seeders**
- Implement **controllers** to fetch related data
- Build **views with Bootstrap** for display

10.7.3 Prerequisites

Before starting, ensure you have:

- **PHP 8.2+**, Composer, and Laravel 12 installed
- **SQLite installed** (default in Laravel 12)
- A **code editor (VS Code, PHPStorm, etc.)**
- **Basic Laravel and Eloquent ORM knowledge**

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

*Make sure you have the **SQLite** extension enabled in your `php.ini` file.*

10.7.4 Steps

Here are the steps to complete this lab:

10.7.4.1 Step 1: Create a New Laravel 12 Project

Open a terminal and create a new Laravel project:

```
| laravel new complex-relationships  
| cd complex-relationships  
| code .
```

You should see a new folder named `complex-relationships` created. Open the project in your favorite code editor, such as Visual Studio Code.

10.7.4.2 Step 2: Configure SQLite Database

Laravel 12 **uses SQLite by default**, so we need to configure it to use **SQLite**.

Open `.env` and update the database settings. Ensure the following lines are present:

```
| DB_CONNECTION=sqlite
```

You don't need to specify `DB_DATABASE` for SQLite, as it will use the default SQLite database file. Unless you want to specify a custom path, you can leave it as is.

10.7.4.3 Step 3: Create Migrations for the Database Schema

We will create migrations for the following tables:

- `users`: Stores user information. We will modify the default migration.
- `profiles`: Stores user profiles (One-to-One relationship with users)
- `posts`: Stores blog posts (One-to-Many relationship with users)
- `tags`: Stores tags for posts (Many-to-Many relationship with posts)
- `post_tag`: Pivot table for Many-to-Many relationship between posts and tags

Run the following command to create migrations for the tables:

```
| php artisan make:migration create_profiles_table  
| php artisan make:migration create_posts_table  
| php artisan make:migration create_tags_table  
| php artisan make:migration create_post_tag_table
```

Laravel 12 already has a default migration for the `users` table. You can see it in `database/migrations/YYYY_MM_DD_create_users_table.php`.

```
use Illuminate\Database\Migrations\Migration;  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Support\Facades\Schema;  
  
return new class extends Migration {  
    public function up(): void  
    {  
        Schema::create('users', function (Blueprint $table) {  
            $table->id();  
            $table->string('name');  
            $table->string('email')->unique();  
            $table->timestamp('email_verified_at')->nullable();  
            $table->string('password');  
            $table->rememberToken();  
            $table->timestamps();  
        });  
    }  
};
```

You can see `users` table has the following columns:

- `id`: Primary key
- `name`: String column for the user's name
- `email`: String column for the user's email (unique)
- `email_verified_at`: Timestamp column for email verification
- `password`: String column for the user's password
- `remember_token`: String column for the remember token
- `created_at`: Timestamp column for when the user was created
- `updated_at`: Timestamp column for when the user was last updated
- `deleted_at`: Timestamp column for when the user was deleted (soft delete)

We will add the `profile` and `posts` relationships in the `User` model later. Edit `database/migrations/YYYY_MM_DD_create_profiles_table.php`:

```
return new class extends Migration {  
    public function up()  
    {  
        Schema::create('profiles', function (Blueprint $table) {  
            $table->id();  
            $table->unsignedBigInteger('user_id')->unique();  
            $table->text('bio')->nullable();  
            $table->string('website')->nullable();  
            $table->timestamps();  
        });  
    }  
};
```

```
        $table->foreign('user_id')->references('id')->on('users')->onDelete('cascade');
    });

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

We modified the `profiles` table to include a foreign key reference to the `users` table. This establishes a **One-to-One relationship** between users and profiles. The `user_id` column is marked as unique to ensure that each user can have only one profile.

Edit `database/migrations/YYYY_MM_DD_create_posts_table.php`:

```
return new class extends Migration {
    public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->unsignedBigInteger('user_id');
            $table->string('title');
            $table->text('content');
            $table->timestamps();

            $table->foreign('user_id')->references('id')->on('users')->onDelete('cascade');
        });
    }

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

This code defines the `posts` table with a foreign key reference to the `users` table. This establishes a **One-to-Many relationship** between users and posts. The `user_id` column is used to associate each post with a specific user.

Edit `database/migrations/YYYY_MM_DD_create_tags_table.php`:

```
return new class extends Migration {
    public function up()
    {
        Schema::create('tags', function (Blueprint $table) {
            $table->id();
            $table->string('name')->unique();
            $table->timestamps();
        });
    }

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

This code defines the `tags` table with a unique `name` column. This establishes a **Many-to-Many relationship** between posts and tags. The `name` column is used to store the name of each tag.

Finally, we need to create a pivot table for the **Many-to-Many relationship** between posts and tags. This table will be named `post_tag` and will contain two foreign keys: `post_id` and `tag_id`.

Edit `database/migrations/YYYY_MM_DD_create_post_tag_table.php`:

```
return new class extends Migration {
    public function up()
    {
        Schema::create('post_tag', function (Blueprint $table) {
            $table->id();
            $table->unsignedBigInteger('post_id');
            $table->unsignedBigInteger('tag_id');
            $table->timestamps();

            $table->foreign('post_id')->references('id')->on('posts')->onDelete('cascade');
            $table->foreign('tag_id')->references('id')->on('tags')->onDelete('cascade');
        });
    }

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

This code defines the `post_tag` table with two foreign key references: `post_id` and `tag_id`. This establishes a **Many-to-Many relationship** between posts and tags. The `post_id` column is used to associate each tag with a specific post, and the `tag_id` column is used to associate each post with a specific tag.

After editing the migration files, we need to run the migrations to create the tables in the SQLite database. Make sure to run the following command to create the tables in the SQLite database:

```
| php artisan migrate
```

You should see a message indicating that the migrations were successful. You can check the SQLite database to confirm that the tables have been created.

If you open the SQLite database file in DBeaver, you should see table relations like this:

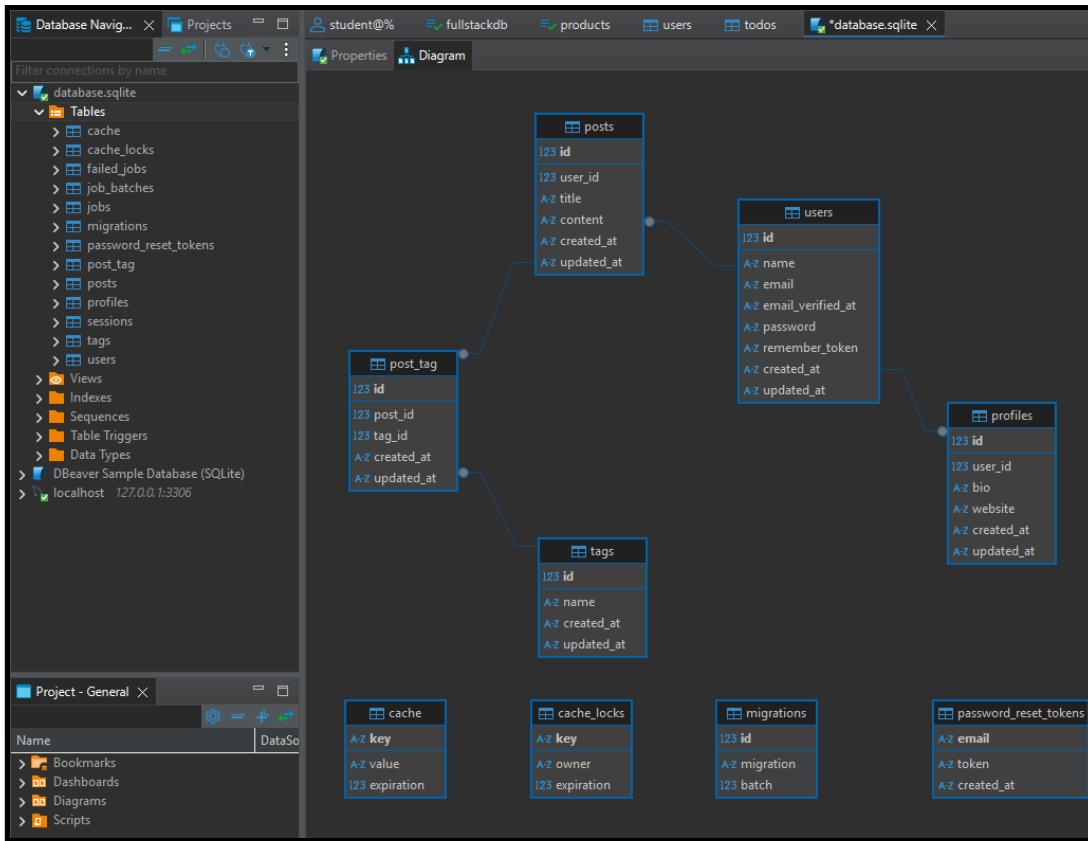


Figure 10.8: SQLite database tables.

You can see the `users`, `profiles`, `posts`, `tags`, and `post_tag` tables created with the specified columns and relationships.

10.7.4.4 Step 4: Define Eloquent Models

We will create models for the following tables:

- `User`: Represents the `users` table. We use the default model created by Laravel.
- `Profile`: Represents the `profiles` table
- `Post`: Represents the `posts` table
- `Tag`: Represents the `tags` table
- `PostTag`: Represents the `post_tag` pivot table

Perform the following commands to create the models:

```
php artisan make:model Profile
php artisan make:model Post
php artisan make:model Tag
```

We don't need to create a model for the `post_tag` table, as we will use the `Post` and `Tag` models to interact with it.

Modify the `User` model to include the relationships with `Profile`, `Post`, and `Tag`. Open `app/Models/User.php` and update it:

```
use App\Models\Profile;
use App\Models\Post;

class User extends Model
{
    ...

    public function profile()
    {
        return $this->hasOne(Profile::class);
    }

    public function posts()
    {
        return $this->hasMany(Post::class);
    }

    ...
}
```

Keep the default `User` model created by Laravel. This code defines the relationships between the `User` model and the `Profile` and `Post` models.

Edit `app/Models/Profile.php`:

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

use app\Models\User;

class Profile extends Model
{
    use HasFactory;

    protected $fillable = ['user_id', 'bio', 'website'];

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

This code defines the `Profile` model and specifies that the `user_id`, `bio`, and `website` attributes are mass assignable. The `user()` method defines the **inverse of the One-to-One relationship** with the `User` model.

Edit `app/Models/Post.php`:

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use App\Models\User;
use App\Models\Tag;

class Post extends Model
```

```

{
    use HasFactory;

    protected $fillable = ['user_id', 'title', 'content'];

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

    public function tags()
    {
        return $this->belongsToMany(Tag::class);
    }
}

```

This code defines the `Post` model and specifies that the `user_id`, `title`, and `content` attributes are mass assignable. The `user()` method defines the **inverse of the One-to-Many relationship** with the `User` model. The `tags()` method defines the **Many-to-Many relationship** with the `Tag` model.

Edit `app/Models/Tag.php`:

```

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use App\Models\Post;

class Tag extends Model
{
    use HasFactory;

    protected $fillable = ['name'];

    public function posts()
    {
        return $this->belongsToMany(Post::class);
    }
}

```

This code defines the `Tag` model and specifies that the `name` attribute is mass assignable. The `posts()` method defines the **Many-to-Many relationship** with the `Post` model.

Save all the files. This code defines the models and their relationships for the `users`, `profiles`, `posts`, and `tags` tables.

10.7.4.5 Step 5: Create Seeders

We will create seeders to populate the `users`, `profiles`, `posts`, and `tags` tables with initial data.

If `database/seeders/DatabaseSeeder.php` file already exists, you can skip this step. If not, run the following command to create a seeder:

```
| php artisan make:seeder DatabaseSeeder
```

This will create a new seeder file in the `database/seeders` directory. Open the generated file in `database/seeders/DatabaseSeeder.php` and update it on `run()` method:

```
use App\Models\User;
use App\Models\Profile;
use App\Models\Post;
use App\Models\Tag;

...

public function run(): void
{
    User::factory(10)->create();

    // create a profile for each user
    foreach (User::all() as $user) {
        $user->profile()->create([
            'bio' => 'This is a bio for user ' . $user->id,
            'website' => 'https://ilmudata.id/user/' . $user->id,
        ]);
    }

    // create posts for each user
    foreach (User::all() as $user) {
        $user->posts()->create([
            'title' => 'Post Title for user ' . $user->id,
            'content' => 'This is the content of the post for user ' . $user->id,
        ]);
    }

    // create tags and associate them with posts
    foreach (Post::all() as $post) {
        $tag = Tag::create(['name' => 'Tag for post ' . $post->id]);
        $post->tags()->attach($tag->id);
    }
}
```

Seed the database:

```
| php artisan db:seed
```

This code creates a user, a profile for that user, a post for that user, and a tag for that post. It also attaches the tag to the post using the `attach()` method.

You can check the SQLite database to confirm that the tables have been populated with initial data.

You can open the SQLite database file in DBeaver and check the `users`, `profiles`, `posts`, and `tags` tables to confirm that the data has been populated.

10.7.4.6 Step 6: Create Controllers

We will create controllers to handle the logic for displaying users, profiles, posts, and tags.

Run the following commands to generate controllers:

```
| php artisan make:controller UserController  
| php artisan make:controller PostController
```

You should see two new controller files created in the `app/Http\Controllers` directory: `UserController.php` and `PostController.php`.

Edit `app/Http\Controllers/UserController.php`:

```
| <?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
use App\Models\User;  
  
class UserController extends Controller  
{  
    public function index()  
    {  
        $users = User::with('profile', 'posts')->get();  
        return view('users.index', compact('users'));  
    }  
  
    public function show(User $user)  
    {  
        return view('users.show', compact('user'));  
    }  
}
```

This code defines the `UserController` with methods to fetch users and their related data. The `index()` method fetches all users with their profiles and posts, while the `show()` method fetches a single user with their profile and posts.

Next, we will create the `PostController` to handle the logic for displaying posts and their related tags.

Edit `app/Http\Controllers/PostController.php`:

```
| <?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
use App\Models\Post;  
  
class PostController extends Controller  
{  
    public function index()  
    {  
        $posts = Post::with('user', 'tags')->get();  
    }  
}
```

```

        return view('posts.index', compact('posts'));
    }

    public function show(Post $post)
    {
        return view('posts.show', compact('post'));
    }
}

```

This code defines the `PostController` with methods to fetch posts and their related data. The `index()` method fetches all posts with their users and tags, while the `show()` method fetches a single post with its user and tags.

10.7.4.7 Step 7: Define Web Routes

We need to define routes for the User and Post controllers in the `routes/web.php` file. Open `routes/web.php` and update it:

```

use Illuminate\Support\Facades\Route;
use App\Http\Controllers\UserController;
use App\Http\Controllers\PostController;

Route::get('/users', [UserController::class, 'index'])->name('users.index');
Route::get('/users/{user}', [UserController::class, 'show'])->name('users.show');

Route::get('/posts', [PostController::class, 'index'])->name('posts.index');
Route::get('/posts/{post}', [PostController::class, 'show'])->name('posts.show');

```

This code defines the routes for the User and Post controllers. Here is a summary of the routes:

- `/users`: Displays a list of users and their profiles
- `/users/{user}`: Displays a single user and their profile
- `/posts`: Displays a list of posts and their authors
- `/posts/{post}`: Displays a single post and its author

10.7.4.8 Step 8: Create Views Using Bootstrap

We will create views for the User and Post controllers using Bootstrap for styling.

Create a new folder `layouts` in `resources/views` and create a new file `resources/views/app.blade.php`. This file will serve as the base template for all views.

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@yield('title')</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>

```

```

<body class="container mt-4">
    <h1 class="text-center mb-4">Laravel 12 Complex Relationships</h1>
    @if(session('success'))
        <div class="alert alert-success">{{ session('success') }}</div>
    @endif
    <nav class="mb-4">
        <a href="{{ route('users.index') }}" class="btn btn-primary">Users</a>
        <a href="{{ route('posts.index') }}" class="btn btn-secondary">Posts</a>
    </nav>
    @yield('content')
</body>
</html>

```

Create a new folder `users` in `resources/views` and create the following files:

- `index.blade.php`: This view will display a list of users and their profiles.
- `show.blade.php`: This view will display a single user and their profile. Create a new folder `posts` in `resources/views` and create the following files:
 - `index.blade.php`: This view will display a list of posts and their authors.
 - `show.blade.php`: This view will display a single post and its author.

Create `resources/views/users/index.blade.php` and add the following code:

```

@extends('layouts.app')

@section('title', 'Users')

@section('content')
    <h2>Users List</h2>

    <ul class="list-group">
        @foreach($users as $user)
            <li class="list-group-item">
                <a href="{{ route('users.show', $user->id) }}>{{ $user->name }}</a> ({{ $user->email }})
            </li>
        @endforeach
    </ul>
@endsection

```

This code defines the `index` view for displaying a list of users and their profiles. Each user is displayed as a link to their profile.

Create `resources/views/users/show.blade.php` and add the following code:

```

@extends('layouts.app')

@section('title', 'User Profile')

@section('content')
    <h2>{{ $user->name }}'s Profile</h2>
    <p>Email: {{ $user->email }}</p>

    <h3>Profile Details</h3>
    <p>Bio: {{ $user->profile->bio ?? 'No bio available' }}</p>
    <p>Website: <a href="{{ $user->profile->website ?? '#' }}>{{ $user->profile->website ?? '#' }}</a></p>

```

```

    'No website' }}></a></p>

    <h3>Posts</h3>
    <ul class="list-group">
        @foreach($user->posts as $post)
            <li class="list-group-item">
                <a href="{{ route('posts.show', $post->id) }}>{{ $post->title }}</a>
            </li>
        @endforeach
    </ul>
@endsection

```

This code defines the `show` view for displaying a single user and their profile. It shows the user's name, email, profile details, and a list of posts created by the user.

Create `resources/views/posts/index.blade.php` and add the following code:

```

@extends('layouts.app')

@section('title', 'Posts')

@section('content')
    <h2>All Posts</h2>

    <ul class="list-group">
        @foreach($posts as $post)
            <li class="list-group-item">
                <a href="{{ route('posts.show', $post->id) }}>{{ $post->title }}</a> by {{ $post->user->name }}
            </li>
        @endforeach
    </ul>
@endsection

```

This code defines the `index` view for displaying a list of posts and their authors. Each post is displayed as a link to its details, along with the author's name.

Create `resources/views/posts/show.blade.php` and add the following code:

```

@extends('layouts.app')

@section('title', 'Post Details')

@section('content')
    <h2>{{ $post->title }}</h2>
    <p><strong>Author:</strong> {{ $post->user->name }}</p>
    <p>{{ $post->content }}</p>

    <h3>Tags</h3>
    <ul class="list-group">
        @foreach($post->tags as $tag)
            <li class="list-group-item">{{ $tag->name }}</li>
        @endforeach
    </ul>

    <a href="{{ route('posts.index') }}" class="btn btn-secondary mt-3">Back to Posts</a>
@endsection

```

Save all the files. This code defines the `show` view for displaying a single post and its author. It shows the post's title, author, content, and a list of tags associated with the post.

10.7.4.9 Testing the Application

After completing all the steps, we can now test the application. Run the following command to start the Laravel development server:

```
| php artisan serve
```

Visit:

```
| http://127.0.0.1:8000/users
```

or

```
| http://127.0.0.1:8000/posts
```

Test Features:

- Click **Users** to view the list of users and their profiles

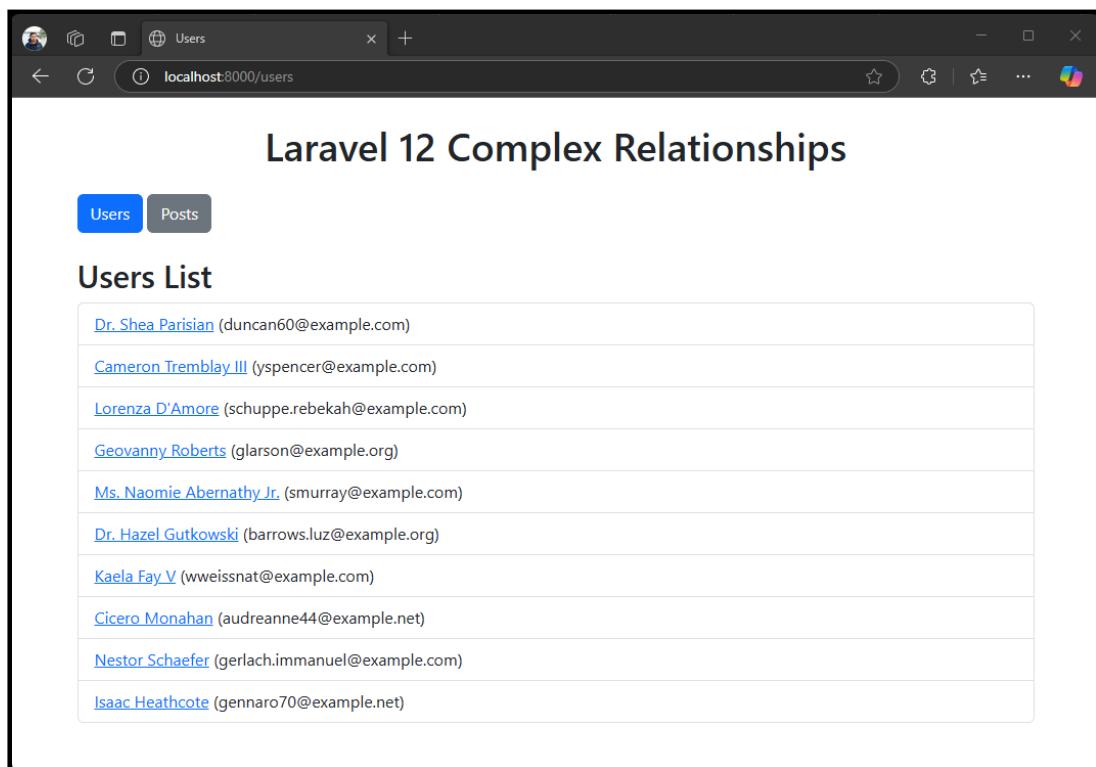


Figure 10.9: Users list.

- Click on a user to view their profile and posts

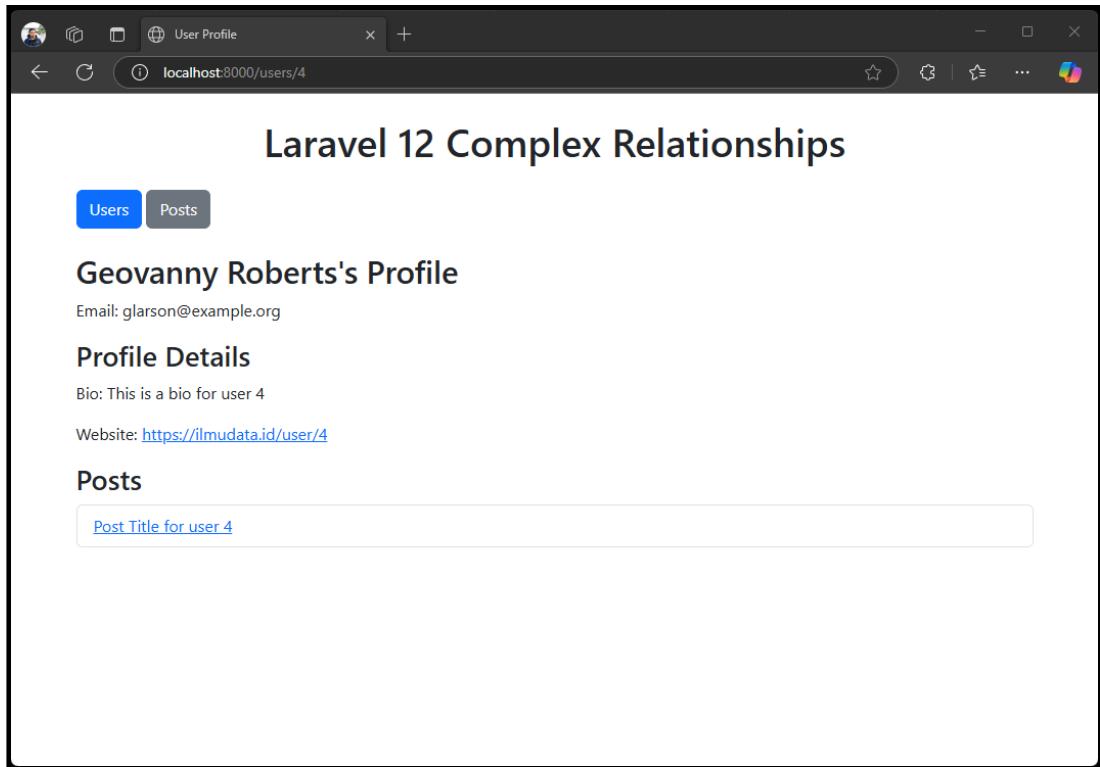


Figure 10.10: User profile.

- Click **Posts** to view the list of posts and their authors

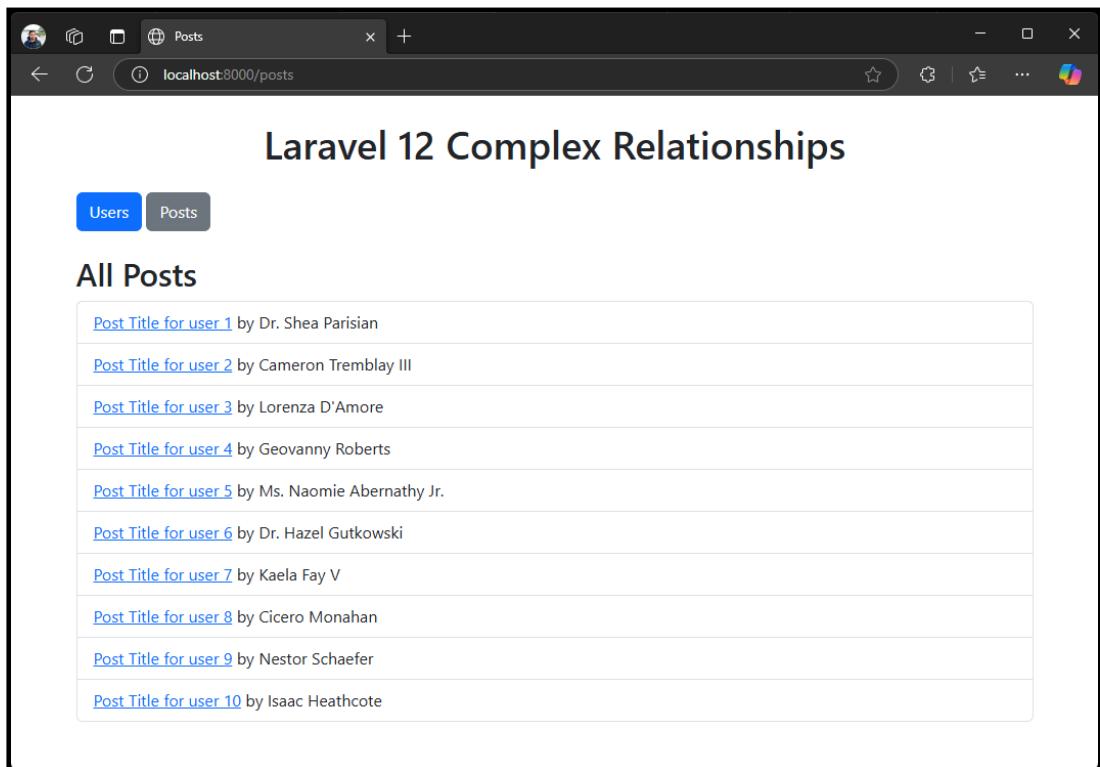


Figure 10.11: Posts list.

- Click on a post to view its details and tags

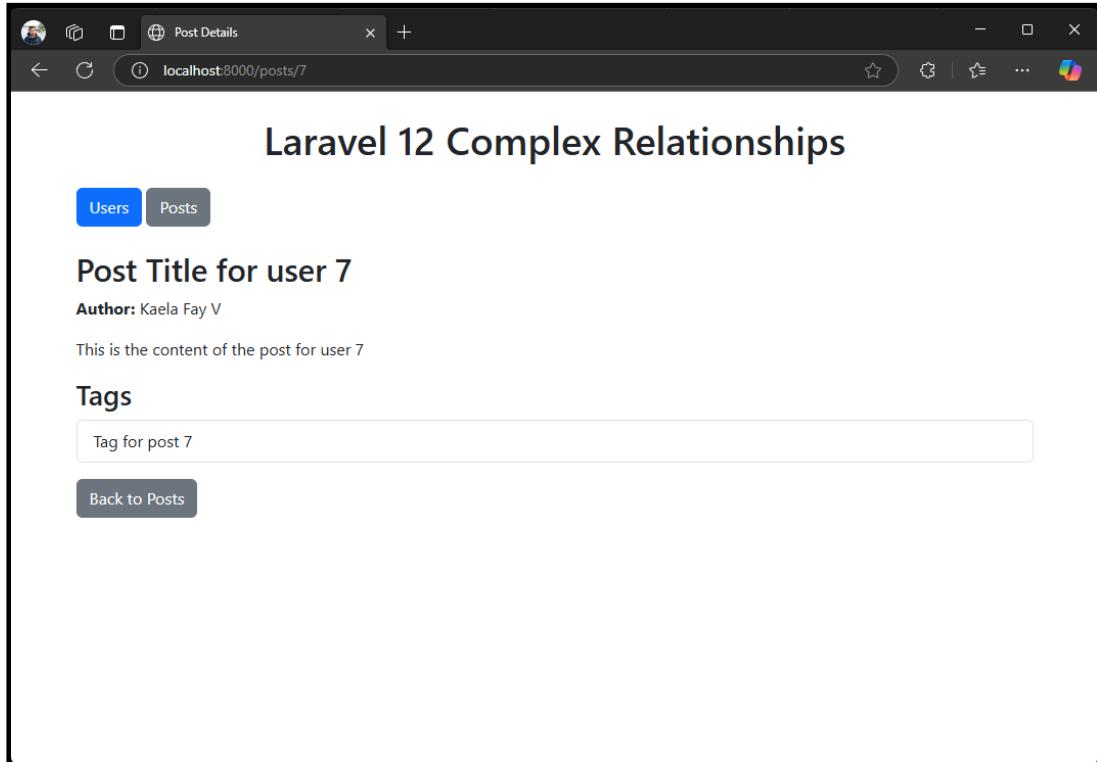


Figure 10.12: Post details.

10.7.5 Summary

Successfully built a **Laravel 12** web app showcasing **complex relationships** using **Eloquent ORM**:

- One-to-One (User ↔ Profile)**
- One-to-Many (User ↔ Posts)**
- Many-to-Many (Posts ↔ Tags)**
- Created controllers and views**
- Implemented Bootstrap styling**

This lab demonstrated how to define and use complex relationships in Laravel 12 using Eloquent ORM. You learned how to create migrations, models, seeders, controllers, and views to interact with these relationships. The frontend was styled using Bootstrap for a better user experience.

10.8 Exercise 31: Pagination with Eloquent ORM (SQLite)

10.8.1 Description

In this lab, you will learn how to implement pagination using Laravel 12's Eloquent ORM and SQLite. You'll create a simple `Product` model, seed the database with dummy data, and use Laravel's built-in pagination methods to paginate the product list. The pagination links are automatically styled with **Tailwind CSS**, which Laravel uses by default.

10.8.2 Objectives

- Set up a Laravel project using SQLite.
- Create a `Product` model, migration, and seeder.
- Use Eloquent ORM to fetch paginated data.
- Display pagination links using Laravel's built-in paginator (Tailwind CSS-compatible).
- Build a simple interface using Blade and Tailwind.

10.8.3 Prerequisites

- Laravel 12 installed.
- SQLite installed or enabled.
- Basic knowledge of Laravel routes, controllers, and Blade templates.

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

10.8.4 Steps

Here are the steps to complete this lab:

10.8.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project using the following command:

```
| laravel new productpagination  
| cd productpagination  
| code .
```

You should see a new folder named `productpagination` created. Open the project in your favorite code editor, such as Visual Studio Code.

10.8.4.2 Step 2: Configure SQLite Database

By default, Laravel uses SQLite as the database. While creating a new project, it creates a SQLite database file in the `database` directory. You can check the `.env` file to see the default database configuration.

If you missed the SQLite database file, you can create it manually. Open the `.env` file and ensure the following lines are present:

```
| DB_CONNECTION=sqlite
```

(Laravel will use `database/database.sqlite` by default.)

10.8.4.3 Step 3: Create Product Model and Migration

We will create a `Product` model and migration using the following command:

```
| php artisan make:model Product -m
```

This command creates a new model file in `app/Models/Product.php` and a migration file in `database/migrations`.

Update the migration `database/migrations/xxxx_create_products_table.php`:

```
public function up(): void
{
    Schema::create('products', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->decimal('price', 10, 2);
        $table->timestamps();
    });
}
```

Save the file. This code defines the `products` table with the following columns:

- `id`: Primary key
- `name`: String column for the product name
- `price`: Decimal column for the product price
- `created_at`: Timestamp column for when the product was created
- `updated_at`: Timestamp column for when the product was last updated
- `deleted_at`: Timestamp column for when the product was deleted (soft delete)

Now, we need to run the migration to create the `products` table in the SQLite database. Run the following command:

```
| php artisan migrate
```

You should see a message indicating that the migration was successful. You can check the SQLite database to confirm that the `products` table has been created.

10.8.4.4 Step 4: Create a Seeder for Dummy Data

We will create a seeder to populate the `products` table with dummy data. Run the following command:

```
| php artisan make:seeder ProductSeeder
```

This command creates a new seeder file in `database/seeders/ProductSeeder.php`. Open the generated file in `database/seeders/ProductSeeder.php` and update it:

```
namespace Database\Seeders;

use Illuminate\Database\Seeder;
use App\Models\Product;

class ProductSeeder extends Seeder
{
    public function run(): void
    {
        Product::factory()->count(50)->create();
    }
}
```

This code defines the `ProductSeeder` class and uses a factory to create 50 dummy products. We will create the factory in the next step.

We modify `Product.php` to use the `HasFactory` trait. Open the generated file in `app/Models/Product.php` and update it:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;
class Product extends Model
{
    use HasFactory;

    protected $fillable = ['name', 'price'];
}
```

Next, we need to create a factory for the `Product` model. Run the following command:

```
| php artisan make:factory ProductFactory --model=Product
```

This command creates a new factory file in `database/factories/ProductFactory.php`. Open the generated file in `database/factories/ProductFactory.php` and update it:

```
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
class ProductFactory extends Factory
```

```
| {
|     public function definition(): array
|     {
|         return [
|             'name' => fake()->word(),
|             'price' => fake()->randomFloat(2, 10, 1000),
|         ];
|     }
| }
```

This code defines the `ProductFactory` class and specifies the attributes for the `Product` model. The `name` attribute is generated using the `word()` method, and the `price` attribute is generated using the `randomFloat()` method.

Now, we modify the `DatabaseSeeder.php` file to call the `ProductSeeder` class. Open the generated file in `database/seeders/DatabaseSeeder.php` and update it:

```
| public function run(): void
| {
|     $this->call([
|         ProductSeeder::class,
|     ]);
| }
```

This code calls the `ProductSeeder` class to seed the database with dummy data. Now, we need to run the seeder to populate the `products` table with dummy data. Run the following command:

```
| php artisan db:seed
```

You should see a message indicating that the seeder was successful. You can check the SQLite database to confirm that the `products` table has been populated with dummy data.

		id	name	price	created_at	updated_at
>	cache	1	harum	766.92	2025-04-10 10:13:44	2025-04-10 10:13:44
>	cache_locks	2	mollitia	836.98	2025-04-10 10:13:44	2025-04-10 10:13:44
>	failed_jobs	3	dolor	660.26	2025-04-10 10:13:44	2025-04-10 10:13:44
>	job_batches	4	autem	930.87	2025-04-10 10:13:44	2025-04-10 10:13:44
>	jobs	5	numquam	610.61	2025-04-10 10:13:44	2025-04-10 10:13:44
>	migrations	6	quisquam	700.74	2025-04-10 10:13:44	2025-04-10 10:13:44
>	password_reset_tokens	7	commodi	274.88	2025-04-10 10:13:44	2025-04-10 10:13:44
>	products	8	cum	306.98	2025-04-10 10:13:44	2025-04-10 10:13:44
>	sessions	9	at	200.87	2025-04-10 10:13:44	2025-04-10 10:13:44
>	sqlite_sequence	10	voluptatibus	299.76	2025-04-10 10:13:44	2025-04-10 10:13:44
>	users	11	et	189.84	2025-04-10 10:13:44	2025-04-10 10:13:44
		12	inventore	928.61	2025-04-10 10:13:44	2025-04-10 10:13:44
		13	quae	473.86	2025-04-10 10:13:44	2025-04-10 10:13:44
		14	cumque	669.03	2025-04-10 10:13:44	2025-04-10 10:13:44
		15	quae	366.73	2025-04-10 10:13:44	2025-04-10 10:13:44
		16	est	220.9	2025-04-10 10:13:44	2025-04-10 10:13:44
		17	sint	714.98	2025-04-10 10:13:44	2025-04-10 10:13:44
		18	voluptas	104	2025-04-10 10:13:44	2025-04-10 10:13:44
		19	dolore	361.33	2025-04-10 10:13:44	2025-04-10 10:13:44
		20	nisi	471.95	2025-04-10 10:13:44	2025-04-10 10:13:44
		21	veniam	191.42	2025-04-10 10:13:44	2025-04-10 10:13:44
		22	aperiam	763.97	2025-04-10 10:13:44	2025-04-10 10:13:44
		23	dolorem	761.86	2025-04-10 10:13:44	2025-04-10 10:13:44

Figure 10.13: SQLite database tables.

10.8.4.5 Step 5: Create Controller for Pagination

We will create a controller to handle the logic for displaying the paginated product list. Run the following command:

```
| php artisan make:controller ProductController
```

This command creates a new controller file in `app/Http/Controllers/ProductController.php`. Open the generated file in `app/Http/Controllers/ProductController.php` and update it:

```
namespace App\Http\Controllers;

use App\Models\Product;

class ProductController extends Controller
{
    public function index()
    {
        $products = Product::orderBy('id', 'desc')->paginate(10);
        return view('products.index', compact('products'));
    }
}
```

This code defines the `ProductController` class and the `index()` method. The `index()` method fetches the products from the database, orders them by `id` in descending order, and

paginates them with 10 products per page. The paginated products are passed to the view.

10.8.4.6 Step 6: Define Route

We need to define a route for the `ProductController` in the `routes/web.php` file. Open the generated file in `routes/web.php` and update it:

```
| use App\Http\Controllers\ProductController;  
|  
| Route::get('/products', [ProductController::class, 'index'])->name('products.index');
```

This code defines the route for the `ProductController`. The `/products` route will display the paginated product list.

10.8.4.7 Step 7: Create View for Product List with Pagination

We will create a view to display the paginated product list. Create a new folder `products` in `resources/views` and create a new file `resources/views/products/index.blade.php`.

Open the generated file in `resources/views/products/index.blade.php` and update it:

```
<!DOCTYPE html>  
<html>  
<head>  
    <title>Paginated Products</title>  
    <script src="https://cdn.tailwindcss.com"></script>  
</head>  
<body class="max-w-4xl mx-auto py-10">  
    <h1 class="text-2xl font-bold mb-5">Product List (Paginated)</h1>  
  
    <table class="table-auto w-full border-collapse border border-gray-300 mb-6">  
        <thead>  
            <tr class="bg-gray-200">  
                <th class="border px-4 py-2">#</th>  
                <th class="border px-4 py-2">Name</th>  
                <th class="border px-4 py-2">Price</th>  
            </tr>  
        </thead>  
        <tbody>  
            @foreach ($products as $product)  
            <tr>  
                <td class="border px-4 py-2">{{ $product->id }}</td>  
                <td class="border px-4 py-2">{{ $product->name }}</td>  
                <td class="border px-4 py-2">${{ number_format($product->price, 2) }}</td>  
            </tr>  
            @endforeach  
        </tbody>  
    </table>  
  
    <div>  
        {{ $products->links() }}  
    </div>  
</body>  
</html>
```

This code defines the view for displaying the paginated product list. It uses **Tailwind CSS** for styling the table and pagination links. The `{{ $products->links() }}` method generates the pagination links automatically.

The `links()` method generates the pagination links based on the current page and the total number of pages. It uses **Tailwind CSS** classes for styling the pagination links.

10.8.4.8 Step 8: Run and Test

After completing all the steps, we can now test the application. Run the following command to start the Laravel development server:

```
| php artisan serve
```

Visit:

<http://localhost:8000/products>

You should see a paginated table of products styled with **Tailwind CSS** and working pagination links.

#	Name	Price
50	eum	\$203.25
49	et	\$259.57
48	sequi	\$364.71
47	nam	\$278.12
46	aliquid	\$579.77
45	labore	\$785.84
44	quam	\$868.45
43	quis	\$633.15
42	temporibus	\$658.16
41	autem	\$502.50

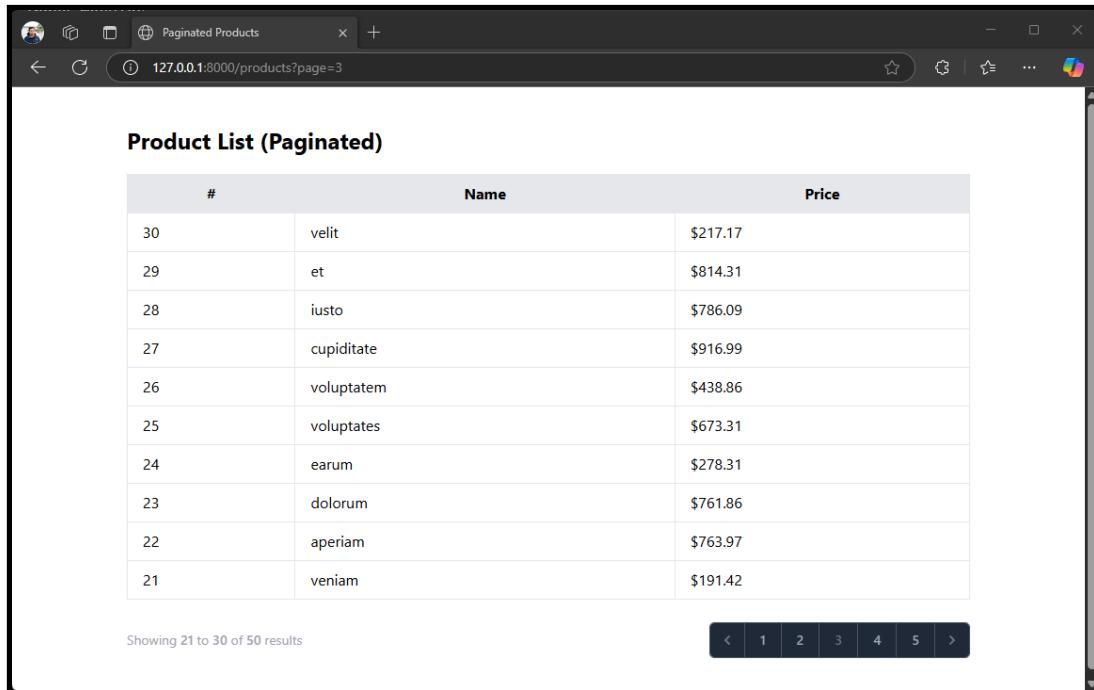
Showing 1 to 10 of 50 results

< | 1 | 2 | 3 | 4 | 5 | >

Figure 10.14: Paginated Products.

Try to navigate through the pages using the pagination links at the bottom of the table. You should see different products on each page.

You can also modify the number of products per page by changing the `paginate(10)` method in the `ProductController` to a different number.



A screenshot of a web browser window titled "Paginated Products". The URL in the address bar is "127.0.0.1:8000/products?page=3". The page displays a table titled "Product List (Paginated)" with columns "#", "Name", and "Price". The data shows 10 products from page 3 of 5. At the bottom, there is a navigation bar with links for <, 1, 2, 3, 4, 5, >.

#	Name	Price
30	velit	\$217.17
29	et	\$814.31
28	iusto	\$786.09
27	cupiditate	\$916.99
26	voluptatem	\$438.86
25	voluptates	\$673.31
24	earum	\$278.31
23	dolorum	\$761.86
22	aperiam	\$763.97
21	veniam	\$191.42

Showing 21 to 30 of 50 results

< 1 2 3 4 5 >

Figure 10.15: Paginated Products on page 3.

10.8.5 Summary

In this lab, you learned how to:

- Use **Eloquent ORM** to fetch paginated records from an **SQLite** database.
- Create and seed dummy data using **factories and seeders**.
- Implement pagination using Laravel's built-in paginator and **Tailwind CSS-compatible UI**.
- Build a clean view for displaying paginated data using **Blade** and **Tailwind**.

This lab demonstrates how to effectively use Laravel's features to create a simple yet powerful paginated product list. You can extend this example by adding more features, such as filtering, searching, and sorting products.

10.9 Conclusion

We have successfully built a simple Todo web app using **Laravel 12**, **Eloquent ORM**, and **MySQL**. We have also built a complex web app showcasing different Eloquent relationships using **SQLite**.

We have learned how to create migrations, models, seeders, controllers, and views to interact with the database. We have also learned how to use Bootstrap for styling the views.

We have also learned how to use Eloquent ORM to define relationships between models and how to use those relationships to fetch related data.

We have also learned how to use Laravel's built-in features such as routing, validation, and authentication to build a complete web application.

OceanofPDF.com

11 Query Builder and Repository Pattern

11.1 Introduction to Query Builder

When working with databases in Laravel, developers are provided with multiple approaches to interact with data. One of the most powerful and flexible tools Laravel offers is the **Query Builder**. Unlike raw SQL queries that are tightly coupled to the database syntax, the Query Builder provides a fluent and expressive interface for constructing and executing SQL statements in a programmatic way.

Laravel's Query Builder is built on top of PDO and supports all major database systems supported by Laravel, including MySQL, PostgreSQL, SQLite, and SQL Server. With its fluent interface, developers can chain methods together to construct complex queries that are still readable and maintainable. This is especially useful in scenarios where you want fine-grained control over SQL queries but still benefit from the safety and structure that Laravel provides.

One key advantage of using the Query Builder is its ability to prevent SQL injection attacks through automatic parameter binding. When you use methods like `where()` or `insert()`, Laravel ensures that all inputs are properly escaped and bound, reducing the security risks that often come with raw SQL queries.

Another benefit of the Query Builder is that it sits comfortably between using raw SQL and Laravel's Eloquent ORM. While Eloquent provides a full-featured, model-centric approach to working with data, it may sometimes be too heavy or abstract for simple queries or performance-critical code. Query Builder gives developers a leaner alternative that can be used for custom reporting, complex joins, or when avoiding the overhead of Eloquent models is preferred.

In this chapter, we will explore how to use Query Builder for common database operations such as selecting, inserting, updating, and deleting records. We will also demonstrate how Query Builder can be used effectively in combination with the Repository Pattern to create clean, reusable, and testable code that separates business logic from data access logic.

11.2 Basic Syntax and Usage

Laravel's Query Builder is designed to be intuitive and readable. It allows you to construct SQL queries using method chaining, which results in clean and fluent syntax. At its core, Query Builder starts with the `DB::table()` method, where you specify the table you want to query. From there, you can chain additional methods to build up your query logic.

To begin using the Query Builder, you first need to import the `DB` facade:

```
| use Illuminate\Support\Facades\DB;
```

11.2.1 Selecting Records

The `get()` method is used to retrieve all matching records as a collection:

```
| $users = DB::table('users')->get();
```

To select specific columns, use the `select()` method:

```
| $users = DB::table('users')
|     ->select('name', 'email')
|     ->get();
```

If you need only a single record, use the `first()` method:

```
| $user = DB::table('users')->where('id', 1)->first();
```

11.2.2 Filtering with Where Clauses

The `where()` method is used to add conditions to the query. It can be chained multiple times to apply multiple conditions:

```
| $activeUsers = DB::table('users')
|     ->where('active', 1)
|     ->where('role', 'admin')
|     ->get();
```

You can also use comparison operators:

```
| $users = DB::table('users')
|     ->where('age', '>', 25)
|     ->get();
```

11.2.3 Ordering, Limiting, and Offsetting

To sort the result, use `orderBy()`:

```
| $users = DB::table('users')
|     ->orderBy('created_at', 'desc')
|     ->get();
```

To limit the number of results, use `limit()`:

```
| $users = DB::table('users')->limit(10)->get();
```

To skip a number of records (for pagination), use `offset()`:

```
| $users = DB::table('users')->offset(10)->limit(10)->get();
```

11.2.4 Aggregates

Query Builder supports SQL aggregate functions like `count`, `sum`, `avg`, `min`, and `max`:

```
| $totalUsers = DB::table('users')->count();
| $averageAge = DB::table('users')->avg('age');
| $maxSalary = DB::table('employees')->max('salary');
```

11.2.5 Joins

You can perform inner joins using the `join()` method:

```
| $orders = DB::table('orders')
|     ->join('users', 'orders.user_id', '=', 'users.id')
|     ->select('orders.*', 'users.name')
|     ->get();
```

Laravel also supports left joins via `leftJoin()`, cross joins with `crossJoin()`, and even advanced join conditions using closures.

11.3 Insert, Update, and Delete

In addition to retrieving records, Laravel's Query Builder makes it simple to insert, update, and delete data in the database. These operations follow a consistent and expressive syntax, which improves code readability and maintainability.

11.3.1 Inserting Records

To insert a new record into a table, use the `insert()` method. This method accepts an associative array where the keys are the column names and the values are the values to insert.

```
| DB::table('users')->insert([
|   'name' => 'Agus Kurniawan',
|   'email' => 'agus@example.com',
|   'active' => 1,
|   'created_at' => now(),
|   'updated_at' => now()
| ]);
```

If you need to retrieve the ID of the newly inserted record, use `insertGetId()`:

```
| $id = DB::table('users')->insertGetId([
|   'name' => 'Dewi Rahmawati',
|   'email' => 'dewi@example.com',
|   'active' => 1,
|   'created_at' => now(),
|   'updated_at' => now()
| ]);
```

11.3.2 Updating Records

To update existing records, use the `update()` method along with a `where()` clause to specify which records to update.

```
| DB::table('users')
|   ->where('id', 1)
|   ->update([
|     'active' => 0,
|     'updated_at' => now()
| ]);
```

⚠ Important: Always use `where()` to avoid updating every record in the table.

11.3.3 Deleting Records

The `delete()` method removes records that match the given condition:

```
| DB::table('users')->where('id', 1)->delete();
```

To delete all records from a table (use with caution), omit the `where()` clause:

```
| DB::table('users')->delete();
```

To truncate the entire table and reset auto-incrementing IDs:

```
| DB::table('users')->truncate();
```

11.4 Query Builder and Raw Expressions

While Laravel's Query Builder covers most use cases, sometimes you need to use raw SQL for complex queries or special SQL expressions. Laravel provides `DB::raw()` for such scenarios, allowing you to write raw expressions within your query.

11.4.1 Selecting with Raw Expressions

```
| $users = DB::table('users')
|     ->select(DB::raw('count(*) as total_users'))
|     ->get();
```

You can also alias fields or perform calculations:

```
| $results = DB::table('orders')
|     ->select(DB::raw('SUM(total) as total_sales'))
|     ->where('status', 'completed')
|     ->get();
```

11.4.2 Using Raw in Where Clauses

```
| $users = DB::table('users')
|     ->whereRaw('YEAR(created_at) = ?', [2024])
|     ->get();
```

11.4.3 Order By with Raw

```
| $users = DB::table('users')
|     ->orderByRaw('LENGTH(name) ASC')
|     ->get();
```

While raw expressions offer flexibility, they should be used cautiously to avoid SQL injection. Always pass dynamic values as bindings rather than injecting them directly into raw strings.

11.5 Introduction to the Repository Pattern

As your Laravel application grows, separating business logic from data access becomes increasingly important. This is where the **Repository Pattern** comes in. The Repository Pattern provides a clean abstraction between your application's business logic and the underlying data source, whether it's a database, API, or something else.

Instead of writing queries directly in your controllers or services, you define a repository class that acts as a mediator between your models or query builders and

the rest of your application. This leads to more maintainable, testable, and reusable code.

For example, imagine you have a controller that needs to fetch active users. Without the repository pattern, you might write a query directly in the controller:

```
| $users = DB::table('users')->where('active', 1)->get();
```

With a repository, this logic is encapsulated in a method:

```
| $users = $userRepository->getActiveUsers();
```

This separation means you can easily swap out the underlying implementation (e.g., switch from Query Builder to Eloquent or an API) without changing the controller. It also improves unit testing, as you can mock repository interfaces instead of depending on the actual database.

11.6 Exercise 32: Performing CRUD with Query Builder in Laravel 12

11.6.1 Description

In this lab, we will build a simple **User Management Web Application** using Laravel 12. Instead of using Eloquent, we will use **Laravel's Query Builder** to perform CRUD (Create, Read, Update, Delete) operations. The UI will be styled using **Bootstrap 5** for a better visual experience.

11.6.2 Objectives

- Understand how to use Laravel Query Builder to interact with the database.
- Build a web-based CRUD interface using Laravel and Blade templates.
- Apply Bootstrap to create a clean and responsive UI.

11.6.3 Prerequisites

Before you start, make sure you have the following:

- Laravel 12 installed globally
- A local or remote MySQL/MariaDB server or SQLite database
- PHP 8.2 or higher

- Composer
- Web browser and any code editor (e.g., VS Code)

We will use **SQLite** for simplicity, but you can adapt the code for MySQL or any other supported database.

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

11.6.4 Steps

Here are the steps to create a simple CRUD application using Laravel 12 Query Builder.

11.6.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project using the following command:

```
| laravel new laravel12-crud-querybuilder  
| cd laravel12-crud-querybuilder  
| code .
```

You should see the Laravel project structure in your code editor.

11.6.4.2 Step 2: Configure the Database

By default, Laravel uses SQLite for local development. You can see this in the `.env` file.

```
| DB_CONNECTION=sqlite
```

If you missed this, you can create a SQLite database file by running:

```
| php artisan migrate
```

11.6.4.3 Step 3: Create the `cars` Table

We will create a new migration for the `cars` table. This table will store information about cars.

```
| php artisan make:migration create_cars_table
```

This will create a new migration file in the `database/migrations` directory. Open the newly created migration file and modify it to create the `cars` table. The file will be named something like `xxxx_xx_xx_xxxxxxx_create_cars_table.php`.

Modify the generated file:

```
| Schema::create('cars', function (Blueprint $table) {
|     $table->id();
|     $table->string('brand');
|     $table->string('model');
|     $table->integer('year');
|     $table->timestamps();
| });
| }
```

Save the file and run the migration to create the `cars` table:

```
| php artisan migrate
```

Verify that the table has been created by checking the database file or using a database management tool.

11.6.4.4 Step 4: Create the Controller

We will create a controller to handle the CRUD operations for the `cars` table.

```
| php artisan make:controller CarController
```

This will create a new controller file in the `app/Http/Controllers` directory. Open the newly created file and add the following methods.

Edit `app/Http/Controllers/CarController.php`:

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class CarController extends Controller
{
    public function index()
    {
        $cars = DB::table('cars')->get();
        return view('cars.index', compact('cars'));
    }

    public function create()
    {
        return view('cars.create');
    }
}
```

```

public function store(Request $request)
{
    DB::table('cars')->insert([
        'brand' => $request->brand,
        'model' => $request->model,
        'year' => $request->year,
        'created_at' => now(),
        'updated_at' => now(),
    ]);
    return redirect()->route('cars.index')->with('success', 'Car added!');
}

public function edit($id)
{
    $car = DB::table('cars')->where('id', $id)->first();
    return view('cars.edit', compact('car'));
}

public function update(Request $request, $id)
{
    DB::table('cars')->where('id', $id)->update([
        'brand' => $request->brand,
        'model' => $request->model,
        'year' => $request->year,
        'updated_at' => now(),
    ]);
    return redirect()->route('cars.index')->with('success', 'Car updated!');
}

public function destroy($id)
{
    DB::table('cars')->where('id', $id)->delete();
    return redirect()->route('cars.index')->with('success', 'Car deleted!');
}
}

```

Save the file. This controller includes methods for listing, creating, editing, updating, and deleting cars.

11.6.4.5 Step 5: Define Routes

We need to define routes for our CRUD operations. Open the `routes/web.php` file and add the following routes:

```

use App\Http\Controllers\CarController;

Route::get('/', [CarController::class, 'index'])->name('cars.index');
Route::get('/create', [CarController::class, 'create'])->name('cars.create');
Route::post('/store', [CarController::class, 'store'])->name('cars.store');
Route::get('/edit/{id}', [CarController::class, 'edit'])->name('cars.edit');
Route::post('/update/{id}', [CarController::class, 'update'])->name('cars.update');
Route::get('/delete/{id}', [CarController::class, 'destroy'])->name('cars.destroy');

```

Comment the default route to avoid conflicts:

```
// Route::get('/', function () {
//     return view('welcome');
//});
```

Save the file. This will set up the routes for our CRUD operations.

11.6.4.6 Step 6: Create Views

We create layout and views for the CRUD operations. Create `layout.blade.php` in the `resources/views` directory. This will be our main layout file.

Codes for `resources/views/layout.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Laravel 12 - Car Management</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div class="container mt-5">
    @yield('content')
</div>
</body>
</html>
```

Next, create a new directory named `cars` in the `resources/views` directory. Inside the `cars` directory, create three files: - `index.blade.php` - `create.blade.php` - `edit.blade.php`

Let's add the code for each of these files.

Codes for `resources/views/cars/index.blade.php`:

```
@extends('layout')

@section('content')
<h2>Car List</h2>
<a href="{{ route('cars.create') }}" class="btn btn-primary mb-3">Add Car</a>

@if(session('success'))
<div class="alert alert-success">{{ session('success') }}</div>
@endif

<table class="table table-bordered">
    <tr>
        <th>ID</th><th>Brand</th><th>Model</th><th>Year</th><th>Actions</th>
    </tr>
    @foreach($cars as $car)
    <tr>
        <td>{{ $car->id }}</td>
        <td>{{ $car->brand }}</td>
        <td>{{ $car->model }}</td>
        <td>{{ $car->year }}</td>
        <td>
```

```

        <a href="{{ route('cars.edit', $car->id) }}" class="btn btn-warning btn-sm">Edit</a>
        <a href="{{ route('cars.destroy', $car->id) }}" class="btn btn-danger btn-sm">Delete</a>
    </td>
</tr>
@endforeach
</table>
@endsection

```

Codes for resources/views/cars/create.blade.php:

```

@extends('layout')

@section('content')
<h2>Add Car</h2>
<form method="POST" action="{{ route('cars.store') }}">
    @csrf
    <div class="mb-3">
        <label>Brand</label>
        <input type="text" name="brand" class="form-control" required />
    </div>
    <div class="mb-3">
        <label>Model</label>
        <input type="text" name="model" class="form-control" required />
    </div>
    <div class="mb-3">
        <label>Year</label>
        <input type="number" name="year" class="form-control" required />
    </div>
    <button type="submit" class="btn btn-success">Save</button>
    <a href="{{ route('cars.index') }}" class="btn btn-secondary">Back</a>
</form>
@endsection

```

Codes for resources/views/cars/edit.blade.php:

```

@extends('layout')

@section('content')
<h2>Edit Car</h2>
<form method="POST" action="{{ route('cars.update', $car->id) }}">
    @csrf
    <div class="mb-3">
        <label>Brand</label>
        <input type="text" name="brand" value="{{ $car->brand }}" class="form-control" required />
    </div>
    <div class="mb-3">
        <label>Model</label>
        <input type="text" name="model" value="{{ $car->model }}" class="form-control" required />
    </div>
    <div class="mb-3">
        <label>Year</label>
        <input type="number" name="year" value="{{ $car->year }}" class="form-control" required />
    </div>
    <button type="submit" class="btn btn-primary">Update</button>
    <a href="{{ route('cars.index') }}" class="btn btn-secondary">Back</a>
</form>
@endsection

```

These views will provide a simple interface for listing, adding, editing, and deleting cars.

Save all the files.

11.6.4.7 Step 7: Run the Application

After creating the views, you can run the application using the built-in PHP server:

```
| php artisan serve
```

Open your browser and navigate to <http://localhost:8000>.

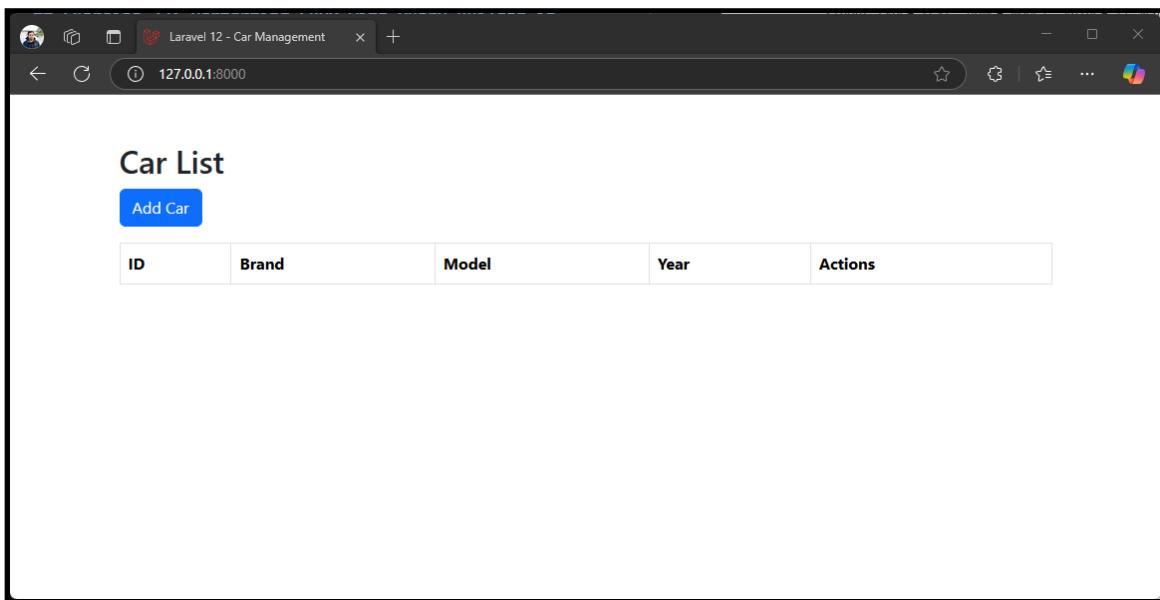


Figure 11.1: Laravel 12 Car Management

You should see the car management interface. You can add, edit, and delete cars using the buttons provided.

You can perform to test the CRUD operations:

- **Add a new car** by clicking the “Add Car” button.
- Enter the car details and click “Save”.

A screenshot of a web browser window titled "Laravel 12 - Car Management". The URL in the address bar is "127.0.0.1:8000/create". The page has a white background and features a form titled "Add Car". The form contains three input fields: "Brand" with the value "Black Car", "Model" with the value "B101", and "Year" with the value "2022". Below the inputs are two buttons: a green "Save" button and a grey "Back" button.

Figure 11.2: Laravel 12 Car Management

- **View the car list** to see the newly added car.

A screenshot of a web browser window titled "Laravel 12 - Car Management". The URL in the address bar is "127.0.0.1:8000". The page has a white background and features a header "Car List" with a blue "Add Car" button. Below the header is a green message box containing the text "Car added!". A table follows, showing a single row of data:

ID	Brand	Model	Year	Actions
1	Black Car	B101	2022	Edit Delete

Figure 11.3: Laravel 12 Car Management

- **Edit an existing car** by clicking the “Edit” button next to the car.
- **Delete a car** by clicking the “Delete” button next to the car.

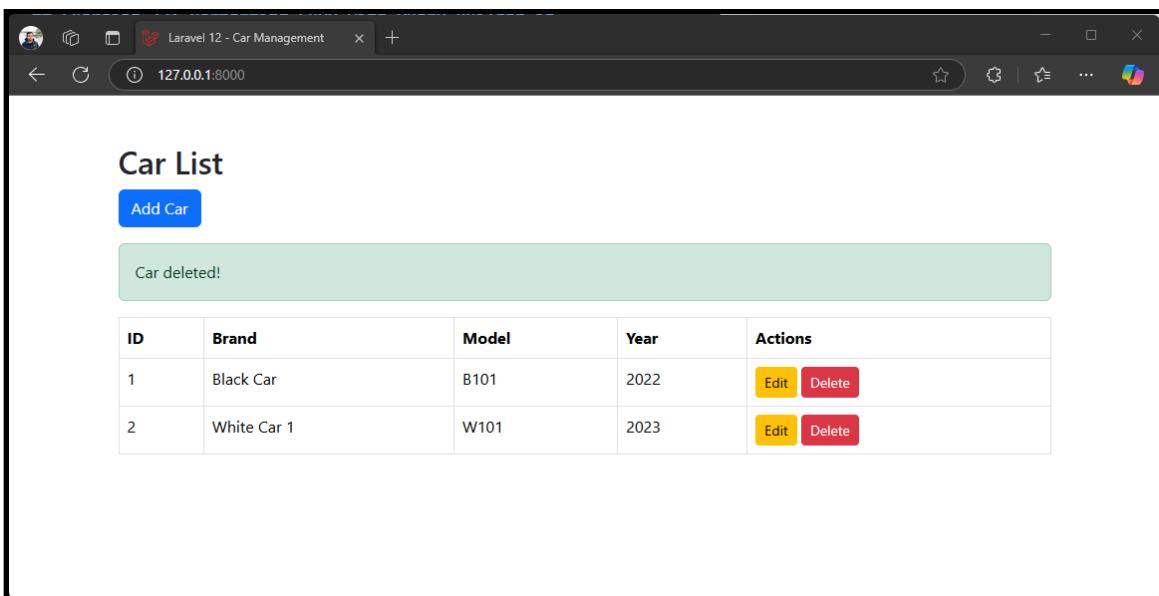


Figure 11.4: Laravel 12 Car Management

11.6.5 Summary

In this exercise, we built a **Car Management Web Application** using **Laravel 12** and **Query Builder**. You learned how to:

- Set up a new custom table (`cars`)
- Perform **CRUD operations** using `DB::table()`
- Create a Bootstrap-styled UI with Laravel Blade
- Organize logic using a dedicated controller

This approach is ideal for scenarios where lightweight, custom data access logic is needed without relying on Eloquent ORM.

11.7 Exercise 33: Query Builder for Joins, Aggregate, and Filtering

11.7.1 Description

In this lab, we will build a Laravel 12 web application that demonstrates how to use **Query Builder** to perform complex queries using **joins**, **aggregate functions**, and **filtering conditions** across related tables. We'll work with three related tables: `users`, `orders`, and `products`. The application will show user order summaries with filtering options. The UI will be styled using **Bootstrap 5**.

11.7.2 Objectives

- Understand how to perform SQL joins using Laravel's Query Builder.
- Use aggregate functions like `SUM`, `COUNT`, and `AVG` to summarize data.
- Apply filtering using `where()`, `whereBetween()`, and `whereNull()`.
- Display data in a Bootstrap-based view.

11.7.3 Prerequisites

- Laravel 12 installed
- MySQL or MariaDB database
- PHP 8.2+
- Composer
- Code editor (e.g., VS Code)

We will use **MySQL** for this lab, but you can adapt the code for SQLite or any other supported database.

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

11.7.4 Steps

Here are the steps to create a simple report application using Laravel 12 Query Builder with joins and aggregate functions.

11.7.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project using the following command:

```
| laravel new laravel12-querybuilder-joins  
| cd laravel12-querybuilder-joins  
| code .
```

Accept the default options for the installation including the database connection.

You should see the Laravel project structure in your code editor.

11.7.4.2 Step 2: Configure the `.env` File

By default, Laravel uses SQLite for local development. You can see this in the `.env` file.

```
| DB_CONNECTION=mysql
```

You also see database file `database.sqlite` in the `database` directory. If you missed this, you can create a SQLite database file by performing the following command:

```
| php artisan migrate
```

11.7.4.3 Step 3: Create Migrations for Users, Products, and Orders

We will create migrations for two tables: `products`, and `orders`. These tables will store information about products, and orders respectively.

Create the migrations:

```
| php artisan make:migration create_products_table
| php artisan make:migration create_orders_table
```

Laravel 12 already has a `users` table by default. You can use that table for this exercise.

Open the newly created migration files in the `database/migrations` directory and modify them to create the `products` and `orders` tables. The files will be named something like `xxxx_xx_xx_xxxxxx_create_products_table.php` and `xxxx_xx_xx_xxxxxx_create_orders_table.php`.

Open `xxxx_xx_xx_xxxxxx_create_products_table.php` and modify it:

```
| Schema::create('products', function (Blueprint $table) {
|     $table->id();
|     $table->string('name');
|     $table->decimal('price', 8, 2);
|     $table->timestamps();
| });
| );
```

Open `xxxx_xx_xx_xxxxxx_create_orders_table.php` and modify it:

```
| Schema::create('orders', function (Blueprint $table) {
|     $table->id();
|     $table->foreignId('user_id')->constrained();
|     $table->foreignId('product_id')->constrained();
|     $table->integer('quantity');
|     $table->timestamps();
| });
| );
```

Save the files and run the migrations to create the `products` and `orders` tables:

```
| php artisan migrate
```

You can verify that the tables have been created by checking the database file or using a database management tool.

11.7.4.4 Step 4: Seed the Database

To populate the tables with sample data, we will create a seeder for the `users`, `products`, and `orders` tables.

```
| php artisan make:seeder SampleSeeder
```

This will create a new seeder file in the `database/seeders` directory. Open the newly created file and add the following code to seed the tables. Edit `database/seeders/SampleSeeder.php`:

```
use Illuminate\Support\Facades\DB;

public function run(): void
{
    DB::table('users')->insert([
        ['name' => 'William', 'email' => 'william@example.com', 'password' => bcrypt('password123')],
        ['name' => 'Nia', 'email' => 'nia@example.com', 'password' => bcrypt('password123')]
    ]);

    DB::table('products')->insert([
        ['name' => 'Laptop', 'price' => 1000],
        ['name' => 'Phone', 'price' => 600],
    ]);

    DB::table('orders')->insert([
        ['user_id' => 1, 'product_id' => 1, 'quantity' => 2],
        ['user_id' => 1, 'product_id' => 2, 'quantity' => 1],
        ['user_id' => 2, 'product_id' => 1, 'quantity' => 1],
    ]);
}
```

Save the file. This seeder will insert sample data into the `users`, `products`, and `orders` tables.

Run the seeder:

```
| php artisan db:seed --class=SampleSeeder
```

You can verify that the tables have been populated by checking the database file or using a database management tool.

11.7.4.5 Step 5: Create Controller

We will create a controller to handle the report generation.

```
| php artisan make:controller ReportController
```

You can find the newly created controller file in the `app/Http/Controllers` directory. Open the newly created file and add the following code to generate the report.

Edit `app/Http/Controllers/ReportController.php`:

```
namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;

class ReportController extends Controller
{
    public function index()
    {
        $reports = DB::table('orders')
            ->join('users', 'orders.user_id', '=', 'users.id')
            ->join('products', 'orders.product_id', '=', 'products.id')
            ->select(
                'users.name as user_name',
                DB::raw('SUM(orders.quantity * products.price) as total_spent'),
                DB::raw('COUNT(orders.id) as total_orders')
            )
            ->groupBy('users.name')
            ->orderByDesc('total_spent')
            ->get();

        return view('reports.index', compact('reports'));
    }
}
```

Save the file. This controller includes a method to generate a report that shows the total amount spent by each user and the total number of orders placed.

11.7.4.6 Step 6: Add Routes

We need to define a route for the report generation. Open the `routes/web.php` file and add the following route:

```
use App\Http\Controllers\ReportController;

Route::get('/', [ReportController::class, 'index'])->name('reports.index');
```

Comment the default route to avoid conflicts:

```
| // Route::get('/', function () {  
| //     return view('welcome');  
| // });
```

save the file. This will set up the route for the report generation.

11.7.4.7 Step 7: Create View with Bootstrap

We will create a view to display the report. Create a new directory named `reports` in the `resources/views` directory. Inside the `reports` directory, create a file named `index.blade.php`.

Codes for `resources/views/reports/index.blade.php`:

```
<!DOCTYPE html>  
<html>  
<head>  
    <title>Order Report</title>  
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">  
</head>  
<body>  
<div class="container mt-5">  
    <h2>User Order Summary</h2>  
    <table class="table table-bordered">  
        <thead>  
            <tr>  
                <th>User</th>  
                <th>Total Orders</th>  
                <th>Total Spent ($)</th>  
            </tr>  
        </thead>  
        <tbody>  
            @foreach($reports as $report)  
            <tr>  
                <td>{{ $report->user_name }}</td>  
                <td>{{ $report->total_orders }}</td>  
                <td>{{ number_format($report->total_spent, 2) }}</td>  
            </tr>  
            @endforeach  
        </tbody>  
    </table>  
</div>  
</body>  
</html>
```

Save the file. This view will display the report in a Bootstrap-styled table.

11.7.4.8 Step 8: Run the App

After creating the view, you can run the application using the built-in PHP server:

```
| php artisan serve
```

Open your browser and go to <http://localhost:8000> to see the report.

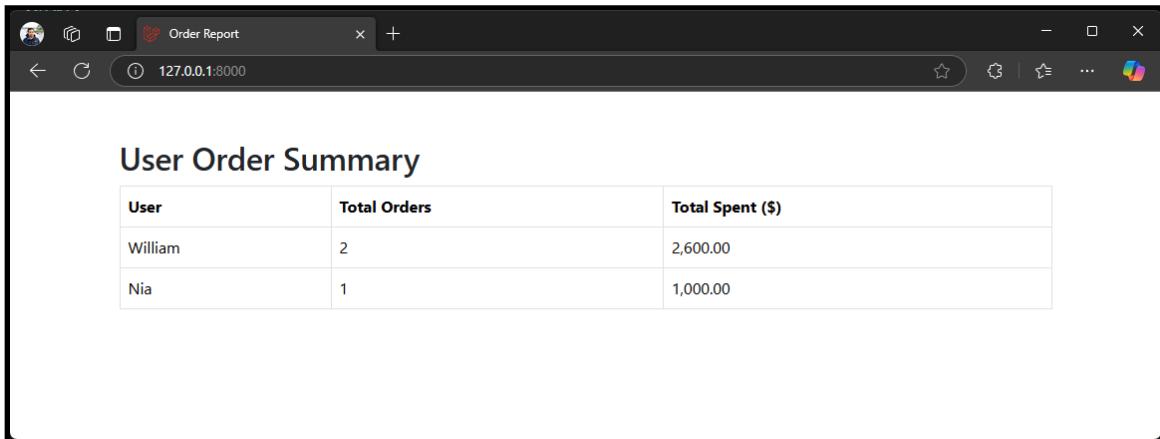


Figure 11.5: Laravel 12 Order Report.

11.7.5 Summary

In this lab, you learned how to:

- Use **Laravel Query Builder** for SQL joins across related tables.
- Apply **aggregate functions** such as `SUM` and `COUNT`.
- Group and sort the results.
- Create a **Bootstrap-styled report** to display the results.

This lab showcases how Laravel's Query Builder can be used for analytical reports and dashboards in real-world applications.

11.8 Exercise 34: Query Builder with Raw Expressions and Secure Input Parameters

11.8.1 Description

In this lab, we will build a Laravel 12 web application that demonstrates how to use **raw expressions** in **Query Builder** using `DB::raw()`, `whereRaw()`, and `orderByRaw()`. You will also learn how to **securely accept and sanitize user input** to prevent SQL injection when using raw expressions. The interface will include a filter form styled with **Bootstrap 5**.

11.8.2 Objectives

- Use `DB::raw()` for custom column calculations.
- Filter and sort data with `whereRaw()` and `orderByRaw()`.
- Safely bind input parameters to avoid SQL injection.
- Display query results using a Bootstrap-based view.

11.8.3 Prerequisites

- Laravel 12 installed
- MySQL or compatible database
- PHP 8.2+
- Composer
- Code editor (e.g., VS Code)

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

11.8.4 Steps

Here are the steps to create a simple report application using Laravel 12 Query Builder with raw expressions.

11.8.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project using the following command:

```
| laravel new laravel12-querybuilder-raw  
| cd laravel12-querybuilder-raw  
| code .
```

Accept the default options for the installation including the database connection. We use SQLite for this lab, but you can adapt the code for MySQL or any other supported database.

You should see the Laravel project structure in your code editor.

11.8.4.2 Step 2: Configure `.env` with Database Connection

Since we are using default SQLite database, you can skip this step. If you want to use MySQL, you can configure the `.env` file with your database connection.

Make sure to set the following variables in your `.env` file:

```
| DB_CONNECTION=sqlite
```

11.8.4.3 Step 3: Create a Migration for `employees`

We will create a migration for the `employees` table. This table will store information about employees.

Create the migration:

```
| php artisan make:migration create_employees_table
```

This will create a new migration file in the `database/migrations` directory. Open the newly created migration file and modify it to create the `employees` table. The file will be named something like `xxxx_xx_xx_xxxxxx_create_employees_table.php`. Open the migration file and modify it:

```
| Schema::create('employees', function (Blueprint $table) {
|     $table->id();
|     $table->string('name');
|     $table->decimal('salary', 10, 2);
|     $table->integer('years_of_service');
|     $table->timestamps();
| });
| }
```

Save the file and run the migration to create the `employees` table:

```
| php artisan migrate
```

You can verify that the table has been created by checking the database file or using a database management tool.

11.8.4.4 Step 4:Seed the Database

We will create a seeder for the `employees` table to populate it with sample data.

Create the seeder:

```
| php artisan make:seeder EmployeeSeeder
```

This will create a new seeder file in the `database/seeders` directory. Open the newly created file and add the following code to seed the `employees` table.

Edit `database/seeders/EmployeeSeeder.php`:

```
use Illuminate\Support\Facades\DB;

public function run(): void
{
    DB::table('employees')->insert([
        ['name' => 'Hiroshi', 'salary' => 5500, 'years_of_service' => 3],
        ['name' => 'Yuki', 'salary' => 7200, 'years_of_service' => 4],
        ['name' => 'Liam', 'salary' => 9500, 'years_of_service' => 6],
        ['name' => 'Emma', 'salary' => 11000, 'years_of_service' => 9],
    ]);
}
```

Save the file. This seeder will insert sample data into the `employees` table.

Modify the `database/seeders/DatabaseSeeder.php` file to call the `EmployeeSeeder`:

```
// In DatabaseSeeder.php
class DatabaseSeeder extends Seeder
{
    public function run(): void
    {
        $this->call(EmployeeSeeder::class);
    }
}
```

Now run the seeder to populate the `employees` table:

```
| php artisan db:seed
```

You can verify that the table has been populated by checking the database file or using a database management tool.

11.8.4.5 Step 5: Create Controller

We will create a controller to handle the report generation.

```
| php artisan make:controller EmployeeReportController
```

You can find the newly created controller file in the `app/Http/Controllers` directory. Open the newly created file and add the following code to generate the report.

Edit `app/Http/Controllers/EmployeeReportController.php`:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class EmployeeReportController extends Controller
{
    public function index(Request $request)
```

```

    {
        $minYears = $request->input('min_years', 0);
        $sort = $request->input('sort', 'bonus DESC');

        // Sanitize allowed sort options
        $allowedSorts = ['bonus DESC', 'bonus ASC', 'name ASC', 'name DESC'];
        if (!in_array($sort, $allowedSorts)) {
            $sort = 'bonus DESC';
        }

        $employees = DB::table('employees')
            ->select(
                'name',
                'salary',
                'years_of_service',
                DB::raw('salary * years_of_service * 0.1 AS bonus')
            )
            ->whereRaw('years_of_service >= ?', [$minYears])
            ->orderByRaw($sort)
            ->get();

        return view('reports.employees', compact('employees', 'minYears', 'sort'));
    }
}

```

Save the file. This controller includes a method to generate a report that shows the total bonus for each employee based on their salary and years of service.

11.8.4.6 Step 6: Define Route

We need to define a route for the report generation. Open the `routes/web.php` file and add the following route:

In `routes/web.php`:

```

use App\Http\Controllers\EmployeeReportController;

Route::get('/', [EmployeeReportController::class, 'index'])->name('employees.report');

```

Comment the default route to avoid conflicts:

```

// Route::get('/', function () {
//     return view('welcome');
// });

```

Save the file. This will set up the route for the report generation.

11.8.4.7 Step 7: Create View with Bootstrap

We will create a view to display the report. Create a new directory named `reports` in the `resources/views` directory. Inside the `reports` directory, create a file named `employees.blade.php`.

Codes for `resources/views/reports/employees.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Employee Bonus Report</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div class="container mt-5">
    <h2>Employee Bonus Report</h2>

    <form method="GET" class="row g-3 mb-4">
        <div class="col-md-4">
            <label for="min_years" class="form-label">Minimum Years of Service</label>
            <input type="number" name="min_years" value="{{ $minYears }}" class="form-control" />
        </div>
        <div class="col-md-4">
            <label for="sort" class="form-label">Sort By</label>
            <select name="sort" class="form-select">
                <option value="bonus DESC" {{ $sort == 'bonus DESC' ? 'selected' : '' }}>Bonus High to Low</option>
                <option value="bonus ASC" {{ $sort == 'bonus ASC' ? 'selected' : '' }}>Bonus Low to High</option>
                <option value="name ASC" {{ $sort == 'name ASC' ? 'selected' : '' }}>Name A-Z</option>
                <option value="name DESC" {{ $sort == 'name DESC' ? 'selected' : '' }}>Name Z-A</option>
            </select>
        </div>
        <div class="col-md-4 d-flex align-items-end">
            <button type="submit" class="btn btn-primary">Filter</button>
        </div>
    </form>

    <table class="table table-bordered">
        <thead>
            <tr>
                <th>Name</th>
                <th>Salary ($)</th>
                <th>Years of Service</th>
                <th>Bonus ($)</th>
            </tr>
        </thead>
        <tbody>
            @foreach($employees as $emp)
            <tr>
                <td>{{ $emp->name }}</td>
                <td>{{ number_format($emp->salary, 2) }}</td>
                <td>{{ $emp->years_of_service }}</td>
                <td>{{ number_format($emp->bonus, 2) }}</td>
            </tr>
            @endforeach
        </tbody>
    </table>
</div>
```

```
| </body>
| </html>
```

Save the file. This view will display the report in a Bootstrap-styled table with a filter form.

11.8.4.8 Step 8: Run the Application

After creating the view, you can run the application using the built-in PHP server:

```
| php artisan serve
```

Navigate to <http://localhost:8000>.

You should see the employee bonus report with a filter form.

Name	Salary (\$)	Years of Service	Bonus (\$)
Emma	11,000.00	9	9,900.00
Liam	9,500.00	6	5,700.00
Yuki	7,200.00	4	2,880.00
Hiroshi	5,500.00	3	1,650.00

Figure 11.6: Laravel 12 Employee Bonus Report.

Try to sort the report by selecting different options from the dropdown and filtering by minimum years of service. Then, click the **Filter** button.

Name	Salary (\$)	Years of Service	Bonus (\$)
Hiroshi	5,500.00	3	1,650.00
Yuki	7,200.00	4	2,880.00
Liam	9,500.00	6	5,700.00
Emma	11,000.00	9	9,900.00

Figure 11.7: Laravel 12 Employee Bonus Report.

11.8.5 Summary

In this lab, you learned how to:

- Use `DB::raw()` for custom column expressions.
- Filter results using `whereRaw()` with bound parameters.
- Sort results dynamically with `orderByRaw()` and safe user input.
- Build a secure and interactive Bootstrap-based report page.

This approach is ideal when working with custom calculations and advanced SQL logic, while still maintaining security and clean code practices.

11.9 Exercise 35: Implementing Repository Pattern using Query Builder

11.9.1 Description

In this lab, we will implement the **Repository Pattern** in a Laravel 12 web application using **Query Builder** instead of Eloquent. We will use a scenario of managing a list of books, allowing users to view all books from the repository injected into a controller. The UI will be styled using **Bootstrap 5**.

11.9.2 Objectives

- Understand the purpose and structure of the Repository Pattern.
- Create a repository using Query Builder.
- Bind repository interfaces to implementations via the service container.
- Inject the repository into a controller and display data using Blade and Bootstrap.

11.9.3 Prerequisites

- Laravel 12 installed
- MySQL database
- PHP 8.2+
- Composer
- Code editor (e.g., VS Code)

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

11.9.4 Steps

Here are the steps to implement the Repository Pattern using Query Builder in Laravel 12.

11.9.4.1 Step 1: Create a New Laravel Project

We will create a new Laravel project using the following command:

```
| laravel new laravel12-repository-pattern  
| cd laravel12-repository-pattern  
| code .
```

Accept the default options for the installation including the database connection.

You should see the Laravel project structure in your code editor.

11.9.4.2 Step 2: Configure Database in `.env`

Since we use SQLite by default, you can skip this step. If you want to use MySQL, you can configure the `.env` file with your database connection based on your local setup.

11.9.4.3 Step 3: Create a Migration for `books` Table

We will create a migration for the `books` table. This table will store information about books.

Create the migration:

```
| php artisan make:migration create_books_table
```

This will create a new migration file in the `database/migrations` directory. Open the newly created migration file and modify it to create the `books` table. The file will be named something like `xxxx_xx_xx_xxxxxxx_create_books_table.php`. Open the migration file and modify it:

```
| Schema::create('books', function (Blueprint $table) {
|     $table->id();
|     $table->string('title');
|     $table->string('author');
|     $table->integer('year');
|     $table->timestamps();
| });
| );
```

Save the file and run the migration to create the `books` table:

```
| php artisan migrate
```

You can verify that the table has been created by checking the database file or using a database management tool.

11.9.4.4 Step 4: Seed the Books Table

We will create a seeder for the `books` table to populate it with sample data. Create the seeder:

```
| php artisan make:seeder BookSeeder
```

This will create a new seeder file in the `database/seeders` directory. Open the newly created file and add the following code to seed the `books` table.

Edit `database/seeders/BookSeeder.php`:

```
| use Illuminate\Support\Facades\DB;
|
| public function run(): void
| {
|     DB::table('books')->insert([
|         ['title' => 'Clean Code', 'author' => 'Robert C. Martin', 'year' => 2008],
|     ]);
| }
```

```
        ['title' => 'The Pragmatic Programmer', 'author' => 'Andy Hunt', 'year' => 1999],
        ['title' => 'Design Patterns', 'author' => 'Erich Gamma', 'year' => 1994],
    ]);
}
```

Modify the `database/seeders/DatabaseSeeder.php` file to call the `BookSeeder`:

```
class DatabaseSeeder extends Seeder
{
    public function run(): void
    {
        $this->call(BookSeeder::class);
    }
}
```

Save the file. This seeder will insert sample data into the `books` table.

Run the seeder to populate the `books` table:

```
| php artisan db:seed
```

This will populate the `books` table with sample data. You can verify that the table has been populated by checking the database file or using a database management tool.

11.9.4.5 Step 5: Create a Repository Interface

We will create an interface for the repository. This interface will define the methods that our repository will implement.

Create a directory `app/Repositories` and then create a file for the interface, `BookRepositoryInterface.php`.

Codes for `app/Repositories/BookRepositoryInterface.php`:

```
<?php
namespace App\Repositories;

interface BookRepositoryInterface
{
    public function getAllBooks();
}
```

Save the file. This interface defines a method `getAllBooks()` that will be implemented in the repository class.

11.9.4.6 Step 6: Create a Repository Class Using Query Builder

Create a file for the repository class, `BookRepository.php` in the same directory.

Modify the file to implement the interface and use Query Builder to fetch data.

Codes for `app/Repositories/BookRepository.php`:

```
<?php
namespace App\Repositories;

use Illuminate\Support\Facades\DB;

class BookRepository implements BookRepositoryInterface
{
    public function getAllBooks()
    {
        return DB::table('books')->orderBy('year', 'desc')->get();
    }
}
```

Save the file. This class implements the `getAllBooks()` method to fetch all books from the database using Query Builder.

11.9.4.7 Step 7: Bind the Interface to the Implementation

Open `app/Providers/AppServiceProvider.php` and update the `register()` method:

```
use App\Repositories\BookRepositoryInterface;
use App\Repositories\BookRepository;

public function register(): void
{
    $this->app->bind(BookRepositoryInterface::class, BookRepository::class);
}
```

11.9.4.8 Step 8: Create the Controller

We will create a controller to handle the book listing. Create the controller:

```
| php artisan make:controller BookController
```

This will create a new controller file in the `app/Http/Controllers` directory. Open the newly created file and add the following code to inject the repository. Edit `app/Http/Controllers/BookController.php`:

```
namespace App\Http\Controllers;

use App\Repositories\BookRepositoryInterface;

class BookController extends Controller
```

```

{
    protected $bookRepo;

    public function __construct(BookRepositoryInterface $bookRepo)
    {
        $this->bookRepo = $bookRepo;
    }

    public function index()
    {
        $books = $this->bookRepo->getAllBooks();
        return view('books.index', compact('books'));
    }
}

```

Save the file. This controller injects the `BookRepositoryInterface` and uses it to fetch all books in the `index()` method.

11.9.4.9 Step 9: Define Route

We need to define a route for the book listing. Open the `routes/web.php` file and add the following route:

```

use App\Http\Controllers\BookController;

Route::get('/', [BookController::class, 'index'])->name('books.index');

```

Comment the default route to avoid conflicts:

```

// Route::get('/', function () {
//     return view('welcome');
// });

```

Save the file. This will set up the route for the book listing.

11.9.4.10 Step 10: Create the Blade View

We will create a view to display the list of books. Create a new directory named `books` in the `resources/views` directory. Inside the `books` directory, create a file named `index.blade.php`.

Codes for `resources/views/books/index.blade.php`:

```

<!DOCTYPE html>
<html>
<head>
    <title>Book Repository</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>

```

```

<body>
<div class="container mt-5">
    <h2>Book List</h2>
    <table class="table table-bordered">
        <thead>
            <tr>
                <th>Title</th><th>Author</th><th>Year</th>
            </tr>
        </thead>
        <tbody>
            @foreach($books as $book)
            <tr>
                <td>{{ $book->title }}</td>
                <td>{{ $book->author }}</td>
                <td>{{ $book->year }}</td>
            </tr>
            @endforeach
        </tbody>
    </table>
</div>
</body>
</html>

```

11.9.4.11 Step 11: Run the Application

After creating the view, you can run the application using the built-in PHP server:

```
| php artisan serve
```

Open your browser at <http://localhost:8000> and view the book list powered by a repository using Query Builder.

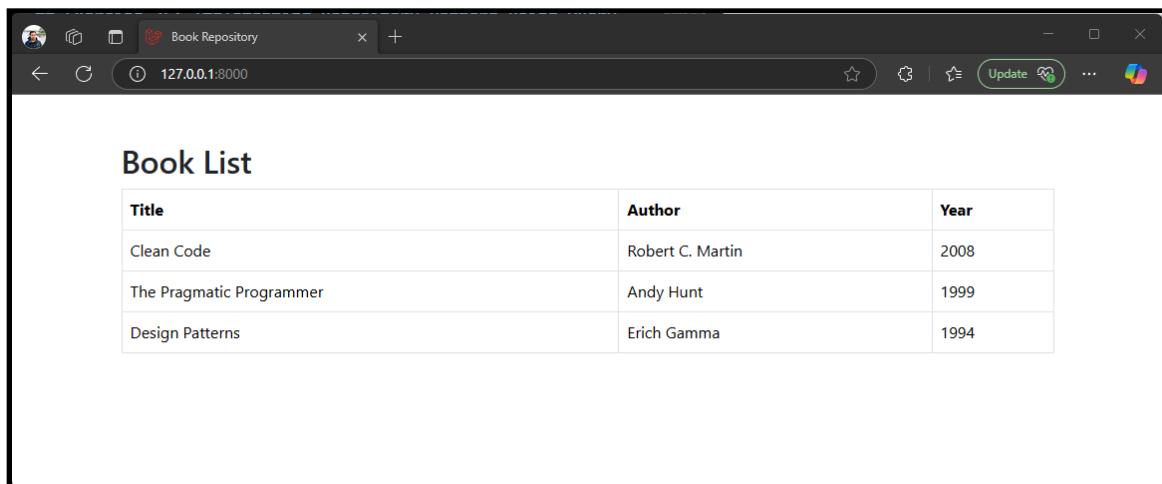


Figure 11.8: Laravel 12 Book Repository.

11.9.5 Summary

In this hands-on lab, you learned how to:

- Implement the **Repository Pattern** in Laravel 12 using **Query Builder**.
- Create an interface and concrete class for data access.
- Bind the interface to the implementation using Laravel's service container.
- Inject the repository into a controller and display the data in a **Bootstrap**-styled view.

This approach separates business logic from data access, improving your application's testability and maintainability.

11.10 Conclusion

In this chapter, we explored the **Query Builder** in Laravel 12, focusing on its capabilities for performing complex queries, joins, and aggregate functions. We also implemented the **Repository Pattern** to demonstrate how to structure your code for better maintainability and testability.

We built several hands-on labs, including:

- A **Car Management Application** using Query Builder for CRUD operations.
- A **Report Application** using joins and aggregate functions to summarize data.
- A **Repository Pattern** implementation to separate data access logic from business logic.

These exercises provided practical experience with Laravel's Query Builder, showcasing its flexibility and power in building robust applications. You also learned how to create a clean and user-friendly interface using **Bootstrap 5**.

You can now apply these concepts to your own projects, leveraging the power of Laravel's Query Builder to create efficient and maintainable applications.

12 NoSQL Databases and Laravel 12

12.1 Introduction to NoSQL Databases

NoSQL (Not Only SQL) databases represent a class of database systems that provide a flexible approach to data storage and retrieval, particularly well-suited for unstructured and semi-structured data. Unlike traditional relational databases, NoSQL databases do not enforce a fixed schema, making them ideal for applications where data structures evolve over time. They are designed to scale horizontally and support high availability and distributed architectures. Common types of NoSQL databases include document stores like MongoDB, key-value stores like Redis, column-family stores such as Apache Cassandra, and graph databases like Neo4j. This flexibility makes NoSQL a powerful option in modern web development, especially when working with big data, real-time applications, or microservices.

12.2 When to Use NoSQL vs SQL

Choosing between NoSQL and SQL databases depends largely on the requirements of your application. SQL databases, like MySQL and PostgreSQL, are well-suited for structured data and complex queries involving relationships. They enforce ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring data integrity. NoSQL databases, in contrast, offer schema flexibility, faster read/write operations at scale, and are often better equipped for handling large volumes of varied data formats. When your application involves frequent changes in data structure, needs to store hierarchical or nested data, or requires high scalability, NoSQL becomes a compelling choice. A comparison table can help you evaluate these trade-offs in your specific use case.

12.3 Installing MongoDB for Laravel

MongoDB is a popular open-source document database, ideal for NoSQL use cases. To use MongoDB with Laravel, you can either install it locally or connect to a managed service like MongoDB Atlas. Laravel does not natively support MongoDB, but this limitation can be addressed using a third-party package such as `mongodb/laravel-mongodb`. This package extends Laravel's Eloquent ORM to work

seamlessly with MongoDB. You can install it via Composer by running the command:

```
| composer require mongodb/laravel-mongodb
```

Once installed, you can begin configuring your Laravel application to use MongoDB as its database backend.

12.4 Configuring MongoDB in Laravel

After installing the MongoDB package, the next step is to configure the Laravel project to use MongoDB. This is done by updating the `.env` file to set the MongoDB connection parameters such as host, port, and database name. For example:

```
| DB_CONNECTION=mongodb  
| MONGODB_URI=  
| MONGODB_DATABASE=
```

You must also register the MongoDB connection inside the `config/database.php` file. Add a new connection array with the driver set to `mongodb` and map the environment variables accordingly. With this setup, Laravel is now ready to use MongoDB as its primary or secondary database connection.

12.5 Creating a MongoDB Model

To work with MongoDB collections in Laravel, you need to define models that extend the MongoDB-compatible Eloquent model provided by the package. Instead of using the base `Illuminate\Database\Eloquent\Model`, your model should extend `MongoDB\Laravel\Eloquent\Model`. For instance, a `Product` model might look like this:

```
| use MongoDB\Laravel\Eloquent\Model;  
  
| class Product extends Model  
{  
|     protected $connection = 'mongodb';  
|     protected $collection = 'products';  
|     protected $fillable = ['name', 'price', 'category'];  
| }
```

This configuration tells Laravel to use the MongoDB connection and map the model to the `products` collection. The `$fillable` array defines which fields can be mass-assigned, similar to standard Eloquent behavior.

12.6 CRUD Operations with MongoDB

Once the model is set up, you can perform CRUD operations in a manner familiar to any Laravel developer using Eloquent. To create a new product, you can use the `create()` method. Reading all records from the collection can be achieved with `Product::all()`. Updating a document involves retrieving it with `find()` and then calling `save()` after modifying its fields. Deleting can be done using the `destroy()` method. Here is an example of each:

```
// Create
Product::create(['name' => 'SSD 1TB', 'price' => 149, 'category' => 'storage']);

// Read
$products = Product::all();

// Update
$product = Product::find($id);
$product->price = 129;
$product->save();

// Delete
Product::destroy($id);
```

This seamless integration makes working with MongoDB intuitive for developers already comfortable with Laravel's Eloquent ORM.

12.7 Using MongoDB Aggregation and Queries

MongoDB offers powerful querying capabilities, including support for advanced conditions and aggregation pipelines. With Laravel's MongoDB package, many of these features are accessible through familiar query builder syntax. You can use operators such as `$gte`, `$lte`, `$in`, and `$or` to perform advanced searches. For example:

```
// Price greater than 100
Product::where('price', '>', 100)->get();

// Category in a list
Product::whereIn('category', ['storage', 'memory'])->get();
```

These queries allow you to filter and retrieve data efficiently without needing to write raw MongoDB queries.

12.8 Real-World Use Case: Product Inventory

To illustrate the practical application of MongoDB with Laravel, consider a product inventory system. Using MongoDB's schema flexibility, you can store products with varying attributes in a single collection. You can build a controller that handles storing, updating, and listing products. These operations are mapped to routes and views using Laravel's MVC pattern. For example, a form can submit data to a controller method that calls `Product::create()`, and a list view can iterate over `Product::all()` to display available products. This approach demonstrates how MongoDB can power real-world applications with dynamic data models.

12.9 MongoDB and Eloquent Relationships (Limitations)

MongoDB does not support joins in the same way as SQL databases. Instead, it encourages data modeling approaches such as embedding related documents or manually referencing them using IDs. While the Laravel MongoDB package offers basic support for relationships like `hasMany` and `belongsTo`, these are not as robust as traditional Eloquent relationships. For one-to-many relationships, you can store an array of embedded documents or a list of referenced IDs. It is important to design your data model carefully, balancing between embedding for performance and referencing for flexibility.

12.10 Exercise 36: CRUD Web App with MongoDB in Laravel 12

12.10.1 Description

In this hands-on lab, we will build a web application using Laravel 12 that performs Create, Read, Update, and Delete (CRUD) operations with MongoDB. We'll use the `jenssegers/laravel-mongodb` package as the MongoDB driver. The app will manage a collection of products, each having a name, price, and stock.

12.10.2 Objectives

- Install and configure MongoDB in a Laravel 12 application.
- Use `jenssegers/laravel-mongodb` to interact with MongoDB.
- Create a Product model and controller.
- Build Bootstrap-styled views to manage products (CRUD).

12.10.3 Prerequisites

- PHP ≥ 8.2
- Composer
- MongoDB installed locally or MongoDB Atlas account
- Laravel 12 installed
- Basic Laravel knowledge

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

12.10.4 Steps

Here are the steps to create a CRUD web application using MongoDB in Laravel 12.

12.10.4.1 Step 1: Install and Configure MongoDB

If you haven't installed MongoDB yet, you can follow the official MongoDB installation guide, <https://docs.mongodb.com/manual/installation> for your operating system. Alternatively, you can use a cloud service like MongoDB Atlas <https://www.mongodb.com/cloud/atlas> to create a free cluster.

You can also use Docker to run MongoDB. To start a MongoDB container with authentication enabled, use the following command:

```
| docker run --name mongodb -d -p 27017:27017 -e MONGO_INITDB_ROOT_USERNAME=admin -e MONGO_II
```

This command sets up MongoDB with a root user (`admin`) and password (`pass12345`). You can access MongoDB at `mongodb://admin:pass12345@localhost:27017`.

To manage MongoDB, we recommend to use MongoDB Compass, a GUI client for MongoDB. You can download it from <https://www.mongodb.com/try/download/compass>.

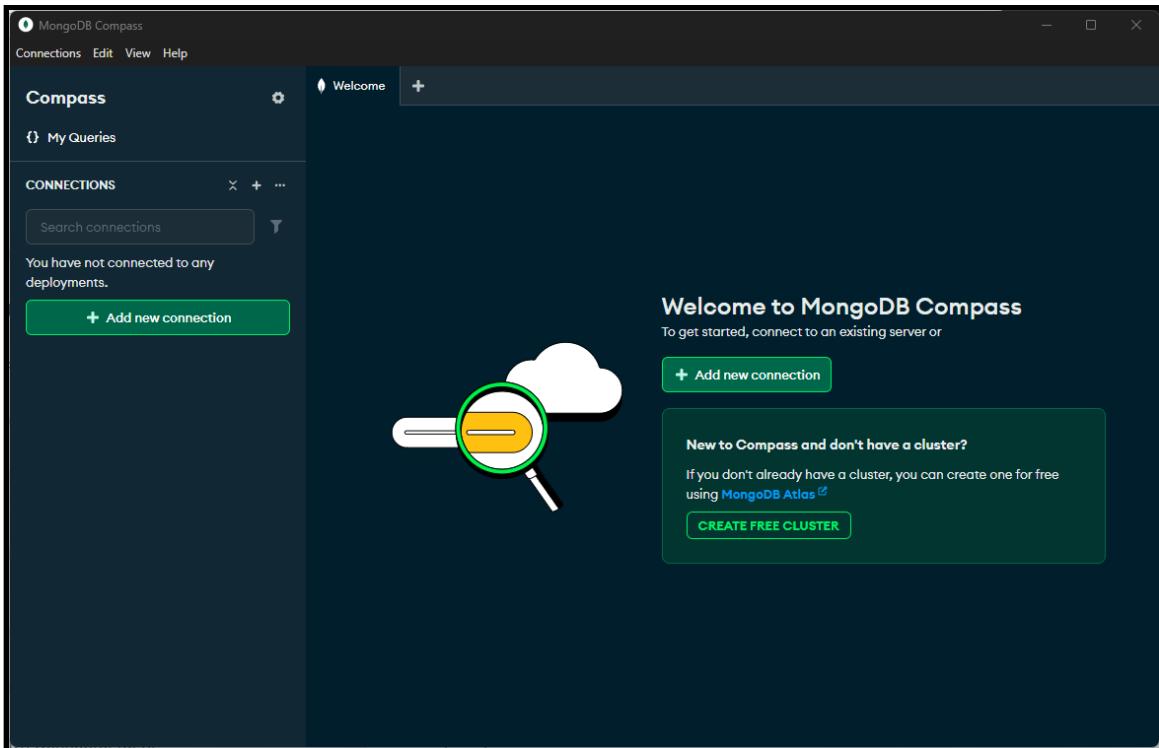


Figure 12.1: MongoDB Compass.

If you want to create additional users, you can connect to the MongoDB instance and configure them as follows:

1. Open MongoDB Compass and connect to your MongoDB instance using the connection string `mongodb://admin:pass12345@localhost:27017/admin`.
2. Click `Create database` in tab to create a new database and collection.
 - o Database name: `productdb`
 - o Collection name: `products`

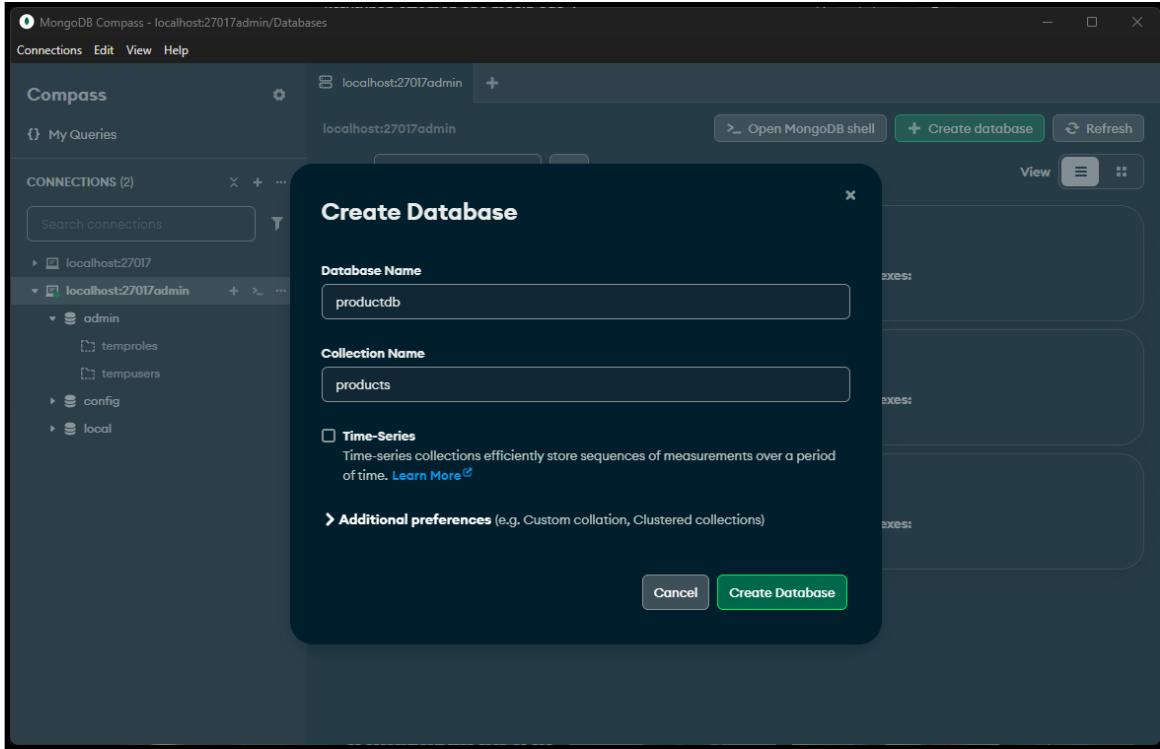


Figure 12.2: Create database and collection in MongoDB Compass.

3. After creating the database, select `productdb` from the left sidebar.
4. Click `Open MongoDB Shell` in the top right corner to open the shell.
5. We create a new user with read/write access to the `productdb` database. In the shell, run the following commands:

```
db.createUser({  
  user: "appuser",  
  pwd: "pass12345",  
  roles: [  
    {  
      role: "readWrite",  
      db: "productdb"  
    }  
  ]  
})
```

The screenshot shows the MongoDB Compass interface. The title bar says "MongoDB Compass - localhost:27017/admin/Shell". The top menu has "Connections", "Edit", "View", and "Help". Below the menu is a "Compass" section with "My Queries" and a search bar. The "CONNECTIONS (2)" section shows two connections: "localhost:27017" and "localhost:27017/admin". The "localhost:27017/admin" connection is expanded, showing the "admin", "config", "local", "productdb", and "temp" databases. The "local" database contains "repset.election", "repset.minvalid", and "startup_log". The "productdb" database contains "products". On the right side of the interface, there is a mongo shell terminal window with the following command and output:

```
>_MONGOSH
> use productdb
< switched to db productdb
> db.createUser({user: "appuser", pwd:"pass12345", roles: [{role: "readWrite", db: "productdb"}]})
< { ok: 1 }
productdb>
```

Figure 12.3: Create user in MongoDB Compass.

This command creates a new user `appuser` with password `pass12345` and grants read/write access to the `productdb` database.

6. Test the connection using the new user credentials.
7. Create a new connection in MongoDB Compass with the following details:
 - o Connection String: `mongodb://appuser:pass12345@localhost:27017/productdb`
 - o Click `Connect` to test the connection.

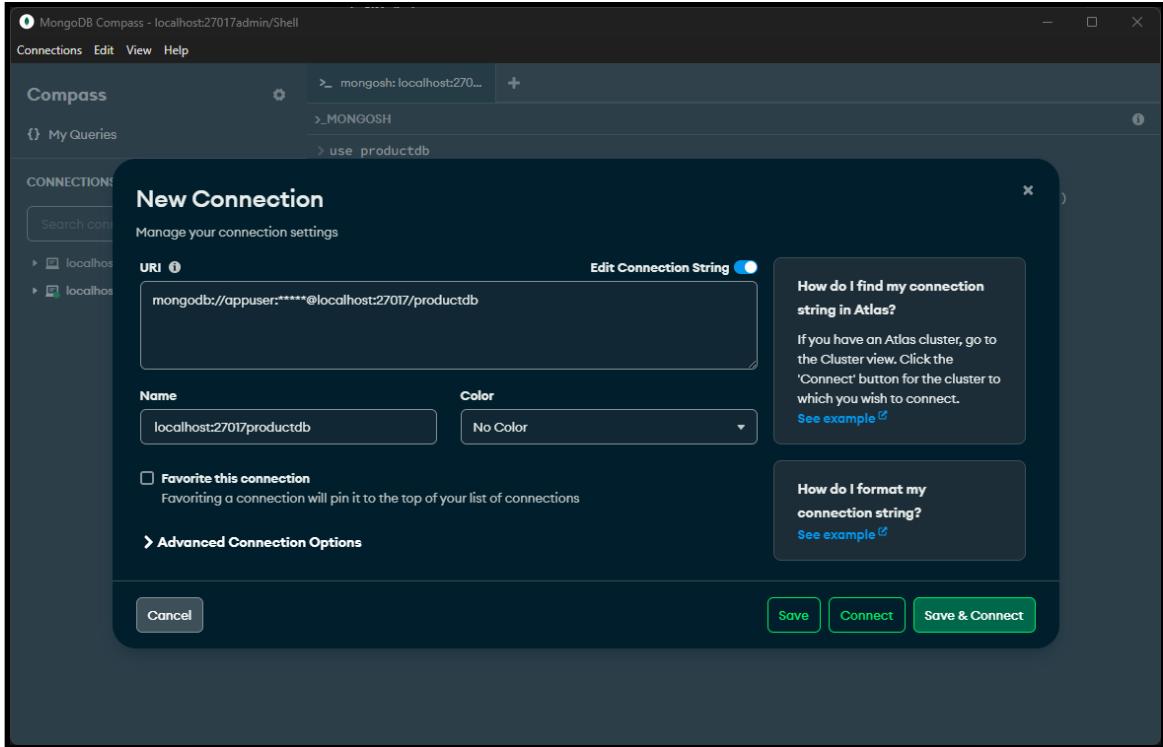


Figure 12.4: Connect to MongoDB with new user.

7. if successful, you should see the `productdb` database and the `products` collection in the left sidebar.

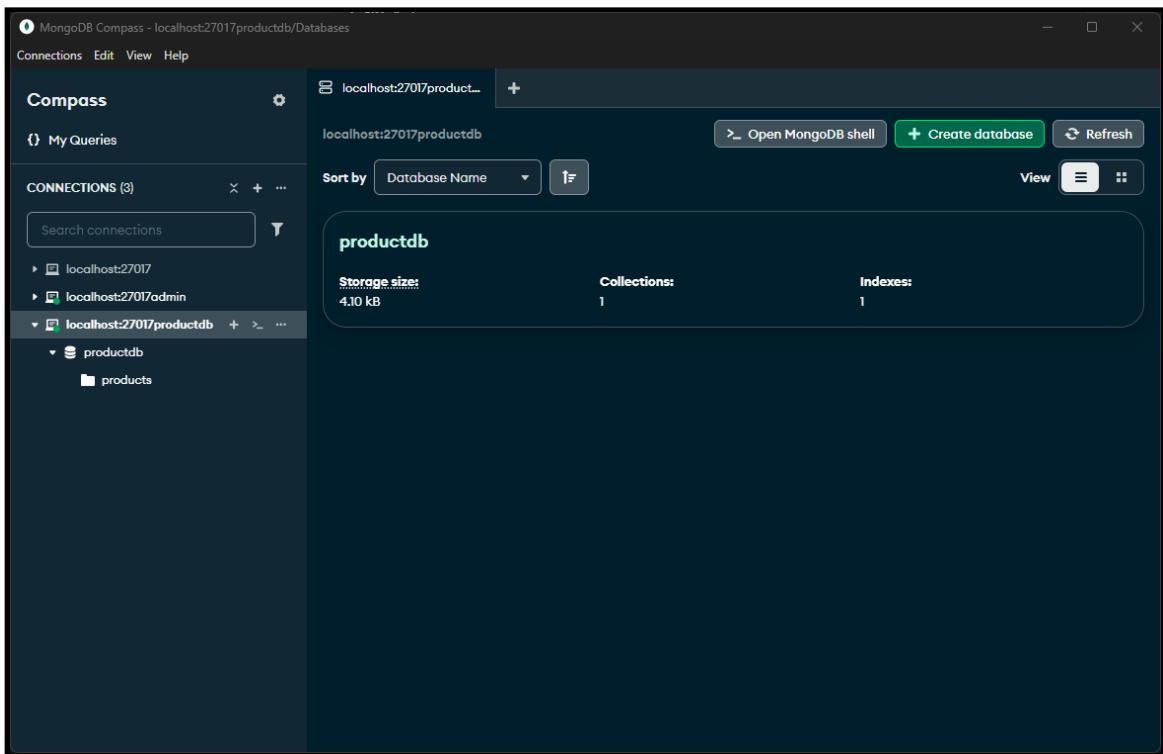


Figure 12.5: MongoDB Compass with new user connection.

Now we have a MongoDB database and collection set up, along with a user that has read/write access to the `productdb` database. You can use this connection string in your Laravel application to interact with MongoDB.

You can now connect to MongoDB using the new user credentials:

```
| mongodb://appuser:pass12345@localhost:27017/productdb
```

12.10.4.2 Step 2: Install MongoDB Driver for Laravel

To work with MongoDB in Laravel, we should install MongoDB driver for Laravel.

Linux, Unix, and macOS users may run the following command to install the extension:

```
| sudo pecl install mongodb
```

Finally, add the following line to the `php.ini` file for each environment that will need to use the extension:

```
| extension=mongodb.so
```

For macOS users, you can use Homebrew to install the MongoDB extension:

```
| brew tap shivammathur/php
| brew install shivammathur/extensions/mongodb@8.4
```

For Windows users, you can download the appropriate DLL file from the PECL repository, <https://pecl.php.net/package/mongodb> and place it in your PHP extensions directory.

For instance, we use `mongodb` pecl version 2.0.0 for PHP 8.4. You can download the DLL file from the following link:

<https://pecl.php.net/package/mongodb/2.0.0/windows>

Download DLL based on your PHP version and architecture (x86 or x64). Place the downloaded DLL file in your PHP extensions directory, usually located at `C:\xampp\php\ext` **Or** `C:\php\phpX.X.X\ext`.

Then, add the following line to your `php.ini` file:

```
| extension=php_mongodb
```

12.10.4.3 Step 3: Create a Laravel Project

We will create a new Laravel project using the Laravel installer. If you don't have the Laravel installer, you can install it globally using Composer:

```
| laravel new laravel-mongo-crud  
| cd laravel-mongo-crud  
| code .
```

Accept the default options to create a new Laravel project. This will create a new directory called `laravel-mongo-crud` with a fresh Laravel installation.

You should see your program structure to Visual Studio Code.

12.10.4.4 Step 4: Install the MongoDB Package

We will use the `mongodb/laravel-mongodb` package to connect Laravel with MongoDB. `mongodb/laravel-mongodb` package, formerly named `jenssegers/mongodb`. This package is now owned and maintained by MongoDB, Inc. and is compatible with Laravel 10.x and later.

This package provides an Eloquent model and query builder for MongoDB.

To install the package, run the following command in your terminal:

```
| composer require mongodb/laravel-mongodb
```

You should see the following output:

```
D:\GitHub\laraveludemy\laravel12book\codes\laravel-mongo-crud>composer require mongodb/laravel-mongodb
./composer.json has been updated
Running composer update mongodb/laravel-mongodb
Loading composer repositories with package information
Updating dependencies
Nothing to modify in lock file
Writing lock file
Installing dependencies from lock file (including require-dev)
Nothing to install, update or remove
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi

  INFO  Discovering packages.

  laravel/pail ..... DONE
  laravel/sail ..... DONE
  laravel/tinker ..... DONE
  mongodb/laravel-mongodb ..... DONE
  nesbot/carbon ..... DONE
  nunomaduro/collision ..... DONE
  nunomaduro/termwind ..... DONE
  pestphp/pest-plugin-laravel ..... DONE

87 packages you are using are looking for funding.
Use the 'composer fund' command to find out more!
> @php artisan vendor:publish --tag=laravel-assets --ansi --force

  INFO  No publishable resources for tag [laravel-assets].
```

No security vulnerability advisories found.
Using version ^5.3 for mongodb/laravel-mongodb

D:\GitHub\laraveludemy\laravel12book\codes\laravel-mongo-crud>

Figure 12.6: Install mongodb/laravel-mongodb package.

Make sure you already have the MongoDB PHP driver installed. You can check this by running `php -m` in your terminal and looking for `mongodb` in the list of installed extensions.

12.10.4.5 Step 5: Configure MongoDB Connection

We need to configure the MongoDB connection in Laravel. Open the `.env` file in the root directory of your Laravel project and update the following lines:

```
DB_CONNECTION=mongodb
MONGODB_URI="mongodb://appuser:pass12345@127.0.0.1:27017/productdb"
MONGODB_DATABASE="productdb"
```

Change the `DB_DATABASE`, `MONGODB_URI`, and `MONGODB_DATABASE` values to match your MongoDB database name, username, and password. You can copy `MONGODB_URI` from the MongoDB Compass connection string.

We also need to update the `config/database.php` file to add the MongoDB connection. Open the `config/database.php` file and add the following code to the `connections` array:

```
'mongodb' => [
    'driver'    => 'mongodb',
    'dsn'       => env('MONGODB_URI', 'mongodb://appuser:pass12345@127.0.0.1:27017/productdb'),
    'database'  => env('MONGODB_DATABASE', 'productdb'),
],
```

Set default connection to `mongodb` in the `default` key:

```
'default' => env('DB_CONNECTION', 'mongodb'),
```

This tells Laravel to use the MongoDB connection when interacting with the database.

We need to activate MongoDB provider in the `bootstrap/providers.php` file. Open the `bootstrap/providers.php` file and add the following line to the `MongoDB\Laravel\MongoDBServiceProvider::class` inside return array:

```
<?php

return [
    ...

    MongoDB\Laravel\MongoDBServiceProvider::class,
];
```

Save all the files and close them.

We need to refresh the configuration cache to apply the changes. Run the following command in your terminal:

```
php artisan config:cache
```

This command will clear the configuration cache and regenerate it with the new settings.

12.10.4.6 Step 6: Create the Product Model

We will create a `Product` model to interact with the `products` collection in MongoDB. Run the following command to create the model:

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

This will create a new file called `Product.php` in the `app/Models` directory. Open the `Product.php` file and update it as follows:

```
<?php

namespace App\Models;

use MongoDB\Laravel\Eloquent\Model;

class Product extends Model
{
    protected $connection = 'mongodb';
    protected $collection = 'products';

    protected $fillable = ['name', 'price', 'stock'];
}
```

Our model is now set up to use the `mongodb` connection and interact with the `products` collection. The `$fillable` property specifies which fields can be mass-assigned when creating or updating a product.

Save the file and close it.

12.10.4.7 Step 7: Create the Product Controller

We will create a `ProductController` to handle the CRUD operations for the `products` collection. Run the following command to create the controller:

```
| php artisan make:controller ProductController --resource
```

This will create a new file called `ProductController.php` in the `app/Http/Controllers` directory. Open the `ProductController.php` file and update it as follows:

```
<?php

namespace App\Http\Controllers;

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

class ProductController extends Controller
{
    public function index()
    {
        $products = Product::all();
        return view('products.index', compact('products'));
    }

    public function create()
    {
        return view('products.create');
    }

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

```

    $request->validate([
        'name' => 'required',
        'price' => 'required|numeric',
        'stock' => 'required|integer'
    ]);

    Product::create($request->all());
    return redirect()->route('products.index')->with('success', 'Product created!');
}

public function edit(Product $product)
{
    return view('products.edit', compact('product'));
}

public function update(Request $request, Product $product)
{
    $request->validate([
        'name' => 'required',
        'price' => 'required|numeric',
        'stock' => 'required|integer'
    ]);

    $product->update($request->all());
    return redirect()->route('products.index')->with('success', 'Product updated!');
}

public function destroy(Product $product)
{
    $product->delete();
    return redirect()->route('products.index')->with('success', 'Product deleted!');
}
}

```

Save the file and close it.

12.10.4.8 Step 8: Define Routes

We need to define the routes for the `ProductController` in the `routes/web.php` file. Open the `routes/web.php` file and add the following code:

```

use App\Http\Controllers\ProductController;

Route::resource('products', ProductController::class);

```

This will create the following routes for the `ProductController`:

- GET /products - **index**
- GET /products/create - **create**
- POST /products - **store**
- GET /products/{product} - **show**

- GET /products/{product}/edit - **edit**
- PUT/PATCH /products/{product} - **update**
- DELETE /products/{product} - **destroy**

Save the file and close it.

12.10.4.9 Step 9: Create Blade Views with Bootstrap

We will create Blade views to display the product list, create a new product, and edit an existing product. We will use Bootstrap for styling.

Create a layout `resources/views/layout.blade.php` file and add the following code:

```
<!DOCTYPE html>
<html>
<head>
    <title>Product CRUD</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div class="container mt-4">
    @yield('content')
</div>
</body>
</html>
```

We will create a `products` directory in the `resources/views` directory. Inside the `products` directory, create the following Blade views:

- `index.blade.php` - for displaying the product list
- `create.blade.php` - for creating a new product
- `edit.blade.php` - for editing an existing product

Codes for index view `resources/views/products/index.blade.php`:

```
@extends('layout')

@section('content')
<h2>Product List</h2>
<a href="{{ route('products.create') }}" class="btn btn-primary mb-3">Add Product</a>
@if(session('success'))
    <div class="alert alert-success">{{ session('success') }}</div>
@endif
<table class="table table-bordered">
    <tr>
        <th>Name</th><th>Price</th><th>Stock</th><th>Action</th>
    </tr>
    @foreach ($products as $product)
    <tr>
        <td>{{ $product->name }}</td>
        <td>${{ $product->price }}</td>
        <td>{{ $product->stock }}</td>
```

```

        <td>
            <a href="{{ route('products.edit', $product->_id) }}" class="btn btn-warning
btn-sm">Edit</a>
            <form action="{{ route('products.destroy', $product->_id) }}" method="POST"
style="display:inline;">
                @csrf @method('DELETE')
                <button class="btn btn-danger btn-sm" onclick="return confirm('Are you
sure?')">Delete</button>
            </form>
        </td>
    </tr>
    @endforeach
</table>
@endsection

```

Codes for create view resources/views/products/create.blade.php:

```

@extends('layout')

@section('content')
<h2>Add New Product</h2>
<form method="POST" action="{{ route('products.store') }}">
    @csrf
    <div class="mb-3">
        <label>Name</label>
        <input name="name" class="form-control" required>
    </div>
    <div class="mb-3">
        <label>Price</label>
        <input name="price" type="number" step="0.01" class="form-control" required>
    </div>
    <div class="mb-3">
        <label>Stock</label>
        <input name="stock" type="number" class="form-control" required>
    </div>
    <button class="btn btn-success">Save</button>
</form>
@endsection

```

Codes for edit view resources/views/products/edit.blade.php:

```

@extends('layout')

@section('content')
<h2>Edit Product</h2>
<form method="POST" action="{{ route('products.update', $product->_id) }}>
    @csrf @method('PUT')
    <div class="mb-3">
        <label>Name</label>
        <input name="name" value="{{ $product->name }}" class="form-control" required>
    </div>
    <div class="mb-3">
        <label>Price</label>
        <input name="price" type="number" step="0.01" value="{{ $product->price }}"
class="form-control" required>
    </div>
    <div class="mb-3">
        <label>Stock</label>
        <input name="stock" type="number" value="{{ $product->stock }}" class="form-
control" required>
    </div>
    <button class="btn btn-primary">Update</button>
</form>

```

```
| </form>
| @endsection
```

Save all the files and close them.

12.10.4.10 Step 10: Run the Application

After creating the views, we can run the application using the built-in PHP server. Run the following command in your terminal:

```
| php artisan serve
```

Visit <http://localhost:8000/products> to use the app.

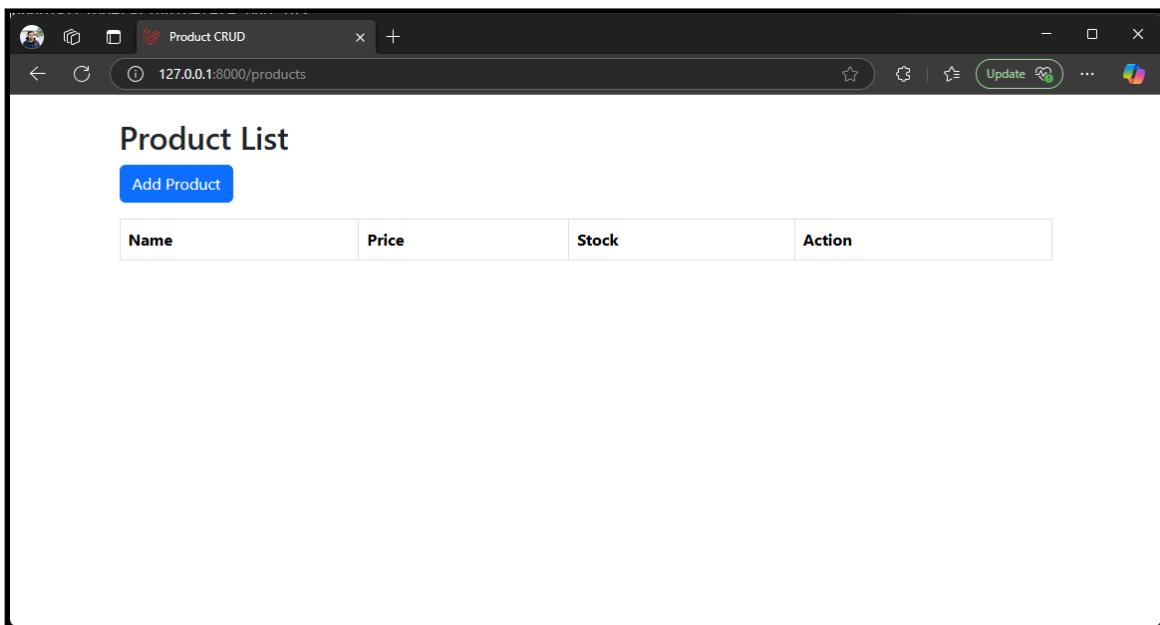


Figure 12.7: Product CRUD application.

You can add, edit, and delete products using the application. The data will be stored in the `products` collection in the `productdb` database in MongoDB.

Click the `Add Product` button to add a new product. Fill in the form and click `Save`. You should see the new product in the list.

Add New Product

Name
Product 1

Price
21.5

Stock
100

Save

Figure 12.8: Add new product.

Enter the product name, price, and stock. Click `Save` to add the product. You should see a success message and the new product in the list.

Product List

Add Product

Product created!

Name	Price	Stock	Action
Product 1	\$21.5	100	Edit Delete

Figure 12.9: Product list.

Try to add more products and edit or delete existing ones.

Name	Price	Stock	Action
Product 1	\$21.5	100	<button>Edit</button> <button>Delete</button>
Product 2	\$42.12	50	<button>Edit</button> <button>Delete</button>
Product 3	\$10.99	40	<button>Edit</button> <button>Delete</button>

Figure 12.10: Product list with more data.

To edit a product, click the `Edit` button next to the product. This will take you to the edit form. Make your changes and click `Update`. You should see a success message and the updated product in the list.

The screenshot shows an 'Edit Product' form. The URL in the browser is `127.0.0.1:8000/products/67fb8fa4087d030aea0a2f33/edit`. The form has three input fields: 'Name' containing 'Product 2-edited', 'Price' containing '9.99', and 'Stock' containing '9'. Below the inputs is a blue 'Update' button.

Figure 12.11: Edit product.

It shows the product details in the form. You can change the name, price, and stock. Click `Update` to save the changes. You should see a success message and the updated product in the list.

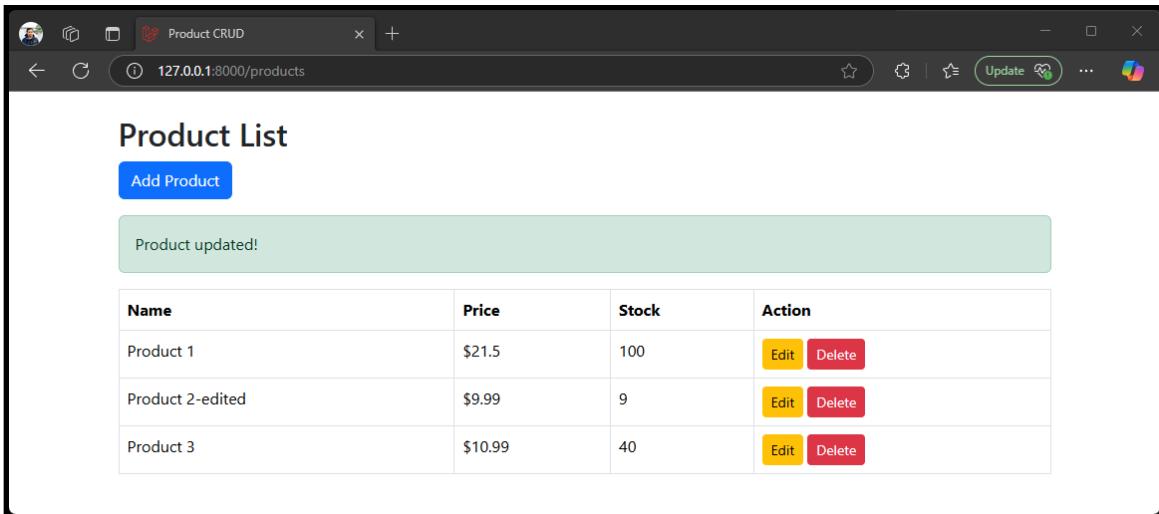


Figure 12.12: Product list after update.

To delete a product, click the `Delete` button next to the product. This will prompt you to confirm the deletion. Click `OK` to delete the product.

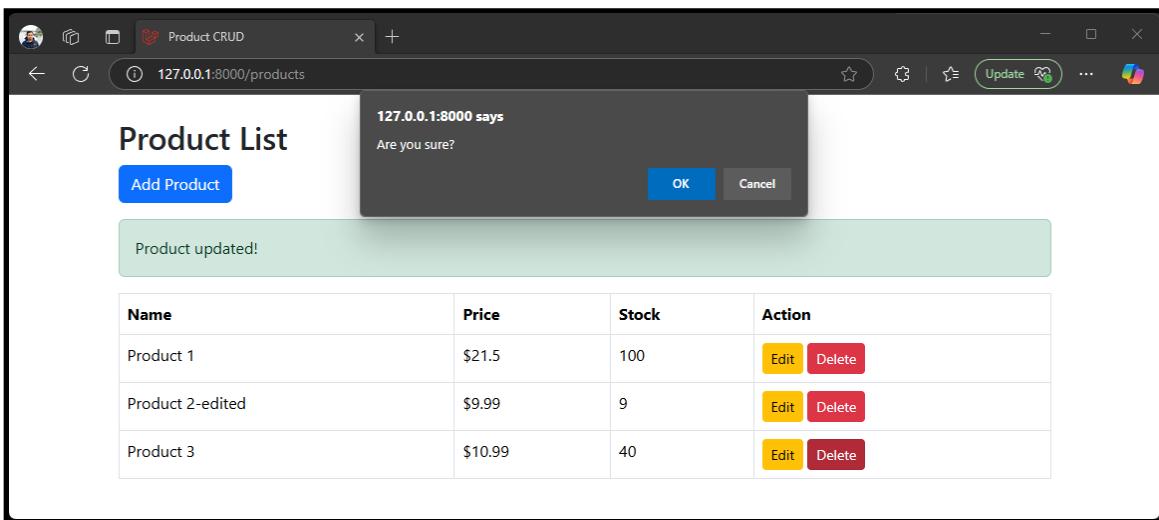


Figure 12.13: Delete product.

You should see a success message and the product removed from the list.

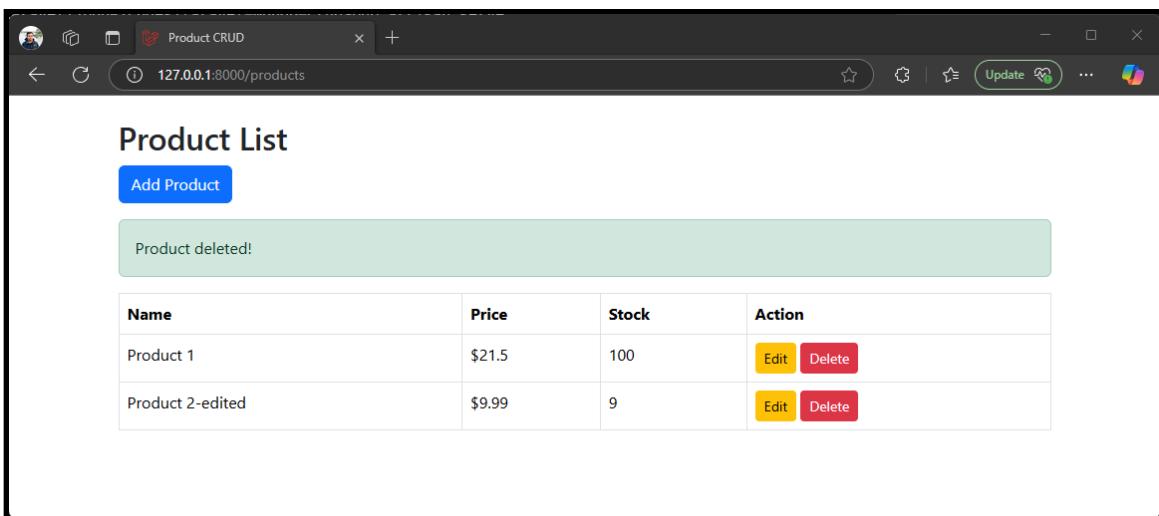


Figure 12.14: Product list after delete.

12.10.5 Summary

In this hands-on lab, we successfully created a Laravel 12 web application integrated with MongoDB using the `mongodb/laravel-mongodb` package. We built a CRUD interface for managing products and styled our views with Bootstrap. This lab demonstrated how easily Laravel can work with NoSQL databases while preserving its clean MVC structure.

12.11 Exercise 37: MongoDB Embedded Relationships with Laravel 12

12.11.1 Description

This hands-on lab demonstrates how to build a Laravel 12 web application using MongoDB with embedded documents. We'll create a `Blog` model where each blog post contains multiple embedded `Comment` objects. This mimics a one-to-many relationship using MongoDB's document model and is ideal for high-performance reads of nested data.

12.11.2 Objectives

- Set up MongoDB and integrate it with Laravel 12.
- Create embedded relationships (one-to-many) using MongoDB documents.
- Seed the database with sample data.
- Display embedded data using Bootstrap views.

12.11.3 Prerequisites

- PHP ≥ 8.2
- Composer
- MongoDB installed or MongoDB Atlas account
- Laravel 12 installed
- Basic Laravel knowledge
- Bootstrap CDN for styling

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

12.11.4 Steps

Here are the steps to create a Laravel 12 web application using MongoDB with embedded documents.

12.11.4.1 Step 1: MongoDB Installation and Configuration

We use the same MongoDB installation and configuration as in the previous lab. You can either install MongoDB locally or use a cloud service like MongoDB Atlas.

Please check exercise 36 for the installation steps.

Make sure you have done the following:

- Installed MongoDB locally or created a cluster in MongoDB Atlas.
- Created a database named `blogdb` and a collection named `blogs`.
- Created a user with read/write access to the `blogdb` database.

12.11.4.2 Step 2: Create Laravel Project

We will create a new Laravel project using the Laravel installer. Run the following command in your terminal:

```
| laravel new laravel-mongo-embed  
| cd laravel-mongo-embed  
| code .
```

You should see your project structure in Visual Studio Code.

12.11.4.3 Step 3: Install the MongoDB Package

We will use the `mongodb/laravel-mongodb` package to connect Laravel with MongoDB. `mongodb/laravel-mongodb` package, formerly named `jenssegers/mongodb`. This package is now owned and maintained by MongoDB, Inc. and is compatible with Laravel 10.x and later.

This package provides an Eloquent model and query builder for MongoDB.

To install the package, run the following command in your terminal:

```
| composer require mongodb/laravel-mongodb
```

Make sure you already have the MongoDB PHP driver installed. You can check this by running `php -m` in your terminal and looking for `mongodb` in the list of installed extensions.

12.11.4.4 Step 4: Configure MongoDB Connection

We need to configure the MongoDB connection in Laravel. Open the `.env` file in the root directory of your Laravel project and update the following lines:

```
| DB_CONNECTION=mongodb  
| MONGODB_URI="mongodb://appuser:pass12345@127.0.0.1:27017/blogdb"  
| MONGODB_DATABASE="blogdb"
```

Change the `DB_DATABASE`, `MONGODB_URI`, and `MONGODB_DATABASE` values to match your MongoDB database name, username, and password. You can copy `MONGODB_URI` from the MongoDB Compass connection string.

We also need to update the `config/database.php` file to add the MongoDB connection. Open the `config/database.php` file and add the following code to the `connections` array:

```
| 'mongodb' => [  
|   'driver'    => 'mongodb',  
|   'dsn'       => env('MONGODB_URI', 'mongodb://appuser:pass12345@127.0.0.1:27017/blogdb'),  
|   'database'  => env('MONGODB_DATABASE', 'blogdb'),  
| ],
```

Set default connection to `mongodb` in the `default` key:

```
| 'default' => env('DB_CONNECTION', 'mongodb'),
```

This tells Laravel to use the MongoDB connection when interacting with the database.

We need to activate MongoDB provider in the `bootstrap/providers.php` file. Open the `bootstrap/providers.php` file and add the following line to the `MongoDB\Laravel\MongoDBServiceProvider::class` inside return array:

```
| <?php  
|  
|     return [  
|     ...  
|  
|         MongoDB\Laravel\MongoDBServiceProvider::class,  
|     ];
```

Save all the files and close them.

We need to refresh the configuration cache to apply the changes. Run the following command in your terminal:

```
| php artisan config:cache
```

This command will clear the configuration cache and regenerate it with the new settings.

12.11.4.5 Step 5: Create Blog Model

We will create a `Blog` model to interact with the `blogs` collection in MongoDB. Run the following command to create the model:

```
| php artisan make:model Blog
```

Edit `app/Models/Blog.php`:

```
| <?php  
|  
| namespace App\Models;  
|  
| use MongoDB\Laravel\Eloquent\Model;  
|  
| class Blog extends Model  
| {  
|     protected $connection = 'mongodb';  
|     protected $collection = 'blogs';  
|  
|     protected $fillable = ['title', 'content', 'comments'];
```

```
|     protected $casts = [
|         'comments' => 'array'
|     ];
| }
```

Save the file and close it.

12.11.4.6 Step 6: Create Seeder with Embedded Comments

We will create a seeder to populate the `blogs` collection with sample data. Run the following command to create the seeder:

```
| php artisan make:seeder BlogSeeder
```

Edit `database/seeders/BlogSeeder.php`:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use App\Models\Blog;

class BlogSeeder extends Seeder
{
    public function run()
    {
        Blog::truncate();

        Blog::create([
            'title' => 'First Blog Post',
            'content' => 'This is a blog post with comments.',
            'comments' => [
                ['author' => 'Kim', 'comment' => 'Great post!'],
                ['author' => 'Susi', 'comment' => 'Thanks for the info.']
            ]
        ]);

        Blog::create([
            'title' => 'Second Blog Post',
            'content' => 'Another post with more comments.',
            'comments' => [
                ['author' => 'Zahra', 'comment' => 'Very helpful!']
            ]
        ]);

        Blog::create([
            'title' => 'Third Blog Post',
            'content' => 'Insights from Europe.',
            'comments' => [
                ['author' => 'Liam', 'comment' => 'Fantastic read!'],
                ['author' => 'Emma', 'comment' => 'Loved the perspective.']
            ]
        ]);
    }
}
```

```
|     Blog::create([
|         'title' => 'Fourth Blog Post',
|         'content' => 'Thoughts from Africa.',
|         'comments' => [
|             ['author' => 'Kwame', 'comment' => 'Very inspiring!'],
|             ['author' => 'Amina', 'comment' => 'Great insights, thank you!']
|         ]
|     ]);
| }
| }
```

Register the seeder in `DatabaseSeeder.php`:

```
| class DatabaseSeeder extends Seeder
| {
|     public function run(): void
|     {
|         $this->call(BlogSeeder::class);
|     }
| }
```

Run the seeder:

```
| php artisan db:seed
```

You can check the data in MongoDB Compass. Open the `blogdb` database and select the `blogs` collection. You should see two blog posts with embedded comments.

The screenshot shows the MongoDB Compass interface. On the left, the 'Connections' sidebar lists three connections: 'localhost:27017', 'localhost:27017admin', and 'localhost:27017productdb'. Under 'localhost:27017admin', the 'blogdb' database is selected, and its 'blogs' collection is highlighted. The main panel displays four documents in the 'Documents' tab. Each document is represented by a card with the following fields:

- _id:** ObjectId('67fba50fc9c5171682033282')
- title:** "First Blog Post"
- content:** "This is a blog post with comments."
- comments:** "[{"author": "Kim", "comment": "Great post!"}, {"author": "Susi", "comment": "..."}]"
- updated_at:** 2025-04-13T11:50:39.615+00:00
- created_at:** 2025-04-13T11:50:39.615+00:00

Below this, the second document is shown:

- _id:** ObjectId('67fba50fc9c5171682033283')
- title:** "Second Blog Post"
- content:** "Another post with more comments."
- comments:** "[{"author": "Zahra", "comment": "Very helpful!"}]"
- updated_at:** 2025-04-13T11:50:39.618+00:00
- created_at:** 2025-04-13T11:50:39.618+00:00

Below this, the third document is shown:

- _id:** ObjectId('67fba50fc9c5171682033284')
- title:** "Third Blog Post"
- content:** "Insights from Europe."
- comments:** "[{"author": "Liam", "comment": "Fantastic read!"}, {"author": "Emma", "comment": "..."}]"
- updated_at:** 2025-04-13T11:50:39.618+00:00
- created_at:** 2025-04-13T11:50:39.618+00:00

Below this, the fourth document is shown:

- _id:** ObjectId('67fba50fc9c5171682033285')
- title:** "Fourth Blog Post"
- content:** "Thoughts from Africa."
- comments:** "[{"author": "Kwame", "comment": "Very inspiring!"}, {"author": "Amina", "comment": "..."}]"
- updated_at:** 2025-04-13T11:50:39.619+00:00
- created_at:** 2025-04-13T11:50:39.619+00:00

Figure 12.15: MongoDB Compass with seeded data.

12.11.4.7 Step 7: Create BlogController

We will create a `BlogController` to handle the display of blog posts. Run the following command to create the controller:

```
| php artisan make:controller BlogController
```

Edit `app/Http/Controllers/BlogController.php`:

```
<?php

namespace App\Http\Controllers;

use App\Models\Blog;

class BlogController extends Controller
{
    public function index()
    {
        $blogs = Blog::all();
        return view('blogs.index', compact('blogs'));
    }
}
```

```
| } }
```

12.11.4.8 Step 8: Define Route

We define the route for the `BlogController` in the `routes/web.php` file. Open the `routes/web.php` file and add the following code:

```
use App\Http\Controllers\BlogController;  
Route::get('/blogs', [BlogController::class, 'index']);
```

Save the file and close it.

12.11.4.9 Step 9: Create View with Bootstrap

We create a layout file and a view to display the blog posts. Create `layout.blade.php` in the `resources/views` directory.

Codes for `resources/views/layout.blade.php`:

```
<!DOCTYPE html>  
<html>  
<head>  
    <title>Blogs</title>  
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">  
</head>  
<body>  
    <div class="container mt-4">  
        @yield('content')  
    </div>  
</body>  
</html>
```

Create a new directory called `blogs` in the `resources/views` directory. Inside the `blogs` directory, create a file called `index.blade.php`.

Codes for `resources/views/blogs/index.blade.php`:

```
@extends('layout')  
  
@section('content')  
<h2>Blog Posts</h2>  
@foreach($blogs as $blog)  
    <div class="card mb-4">  
        <div class="card-body">  
            <h4>{{ $blog->title }}</h4>  
            <p>{{ $blog->content }}</p>  
            <h6>Comments:</h6>  
            <ul class="list-group">  
                @foreach ($blog->comments as $comment)
```

```
        <li class="list-group-item">
            <strong>{{ $comment['author'] }}:</strong> {{ $comment['comment'] }}
    </li>
    @endforeach
</ul>
</div>
</div>
@endforeach
@endsection
```

Save all the files and close them.

12.11.4.10 Step 10: Run the Application

After creating the views, we can run the application using the built-in PHP server. Run the following command in your terminal:

```
| php artisan serve
```

Access the app at <http://localhost:8000/blogs> to view blog posts and their embedded comments.

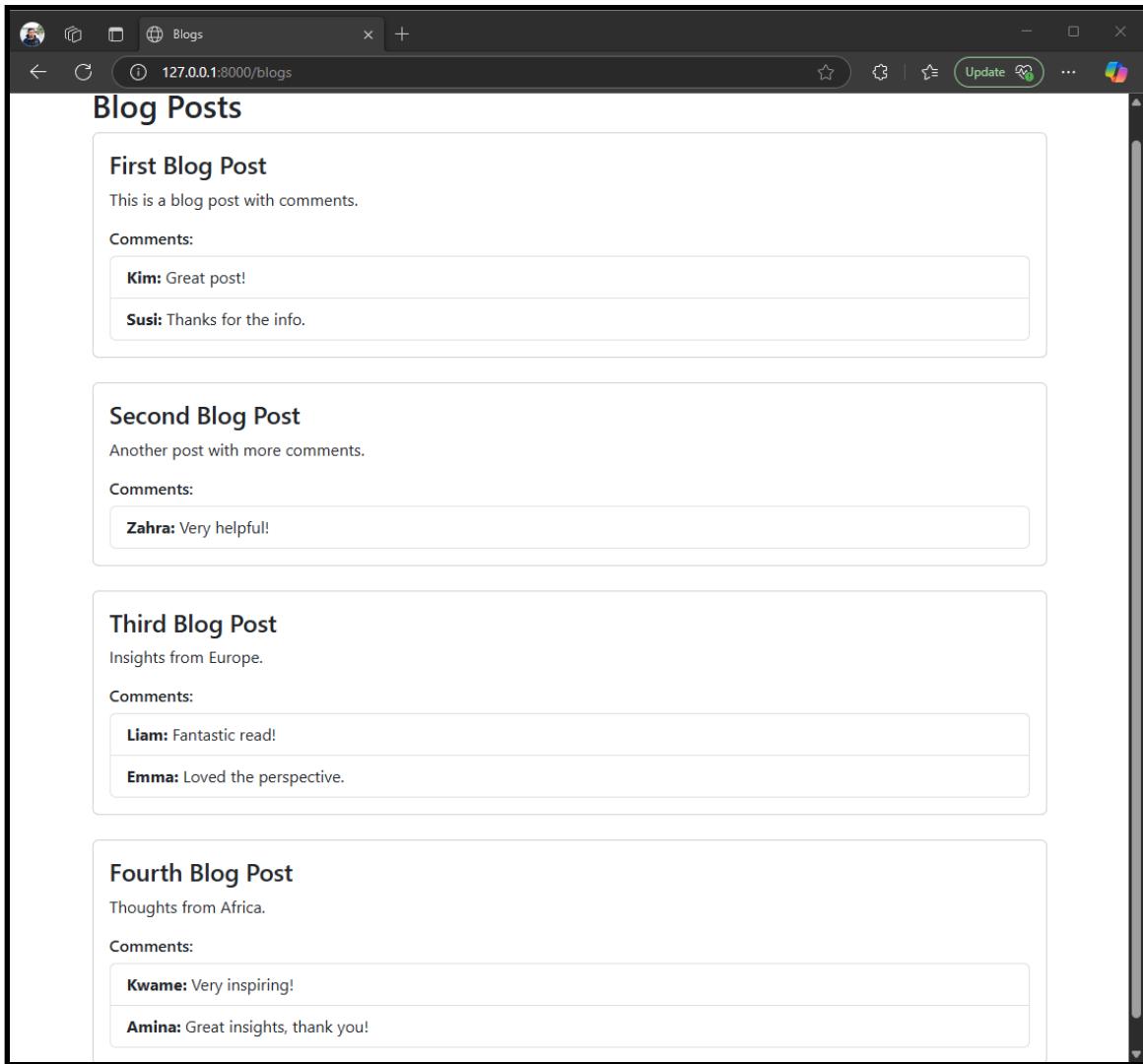


Figure 12.16: Blog posts with embedded comments.

12.11.5 Summary

In this lab, we created a Laravel 12 web app that uses MongoDB's embedded document feature to model a one-to-many relationship between blog posts and comments. We created a `Blog` model with embedded `comments`, seeded the database, and displayed the data in a Bootstrap-styled view. This approach shows how Laravel can work effectively with MongoDB's flexible data model for nested data structures.

12.12 Exercise 38: Data Pagination with MongoDB in Laravel 12

12.12.1 Description

In this hands-on lab, you'll build a Laravel 12 web application that displays a paginated list of documents from a MongoDB collection using the `jenssegers/laravel-mongodb` driver. The pagination will use Laravel's built-in paginator and render Tailwind-compatible HTML out of the box.

12.12.2 Objectives

- Set up Laravel 12 with MongoDB using the community driver
- Seed MongoDB with multiple data entries
- Create a model and controller to retrieve paginated data
- Display paginated results in a view using Tailwind CSS-compatible markup

12.12.3 Prerequisites

- PHP \geq 8.2
- Composer
- Laravel 12 installed
- MongoDB installed (or MongoDB Atlas account)
- Basic Laravel and Blade knowledge

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

12.12.4 Steps

12.12.4.1 Step 1: MongoDB Installation and Configuration

We use the same MongoDB installation and configuration as in the previous lab. You can either install MongoDB locally or use a cloud service like MongoDB Atlas.

Please check exercise 36 for the installation steps. Make sure you have done the following:

- Installed MongoDB locally or created a cluster in MongoDB Atlas.
- Created a database named `paginationdb` and a collection named `products`.
- Created a user with read/write access to the `paginationdb` database.

- Created a user with read/write access to the `paginationdb` database.

12.12.4.2 Step 2: Create Laravel Project

Create a new Laravel project using the Laravel installer. Run the following command in your terminal:

```
| laravel new laravel-mongo-pagination  
| cd laravel-mongo-pagination  
| code .
```

You should see your project structure in Visual Studio Code.

12.12.4.3 Step 3: Install the MongoDB Package

We will use the `mongodb/laravel-mongodb` package to connect Laravel with MongoDB. `mongodb/laravel-mongodb` package, formerly named `jenssegers/mongodb`. This package is now owned and maintained by MongoDB, Inc. and is compatible with Laravel 10.x and later.

This package provides an Eloquent model and query builder for MongoDB.

To install the package, run the following command in your terminal:

```
| composer require mongodb/laravel-mongodb
```

Make sure you already have the MongoDB PHP driver installed. You can check this by running `php -m` in your terminal and looking for `mongodb` in the list of installed extensions.

12.12.4.4 Step 4: Configure MongoDB Connection

We need to configure the MongoDB connection in Laravel. Open the `.env` file in the root directory of your Laravel project and update the following lines:

```
| DB_CONNECTION=mongodb  
| MONGODB_URI="mongodb://appuser:pass12345@127.0.0.1:27017/paginationdb"  
| MONGODB_DATABASE="paginationdb"
```

Change the `DB_DATABASE`, `MONGODB_URI`, and `MONGODB_DATABASE` values to match your MongoDB database name, username, and password. You can copy `MONGODB_URI` from

the MongoDB Compass connection string.

We also need to update the `config/database.php` file to add the MongoDB connection. Open the `config/database.php` file and add the following code to the `connections` array:

```
'mongodb' => [
    'driver' => 'mongodb',
    'dsn' => env('MONGODB_URI', 'mongodb://appuser:pass12345@127.0.0.1:27017/paginationdb'),
    'database' => env('MONGODB_DATABASE', 'paginationdb'),
],
```

Set default connection to `mongodb` in the `default` key:

```
'default' => env('DB_CONNECTION', 'paginationdb'),
```

This tells Laravel to use the MongoDB connection when interacting with the database.

We need to activate MongoDB provider in the `bootstrap/providers.php` file. Open the `bootstrap/providers.php` file and add the following line to the `MongoDB\Laravel\MongoDBServiceProvider::class` inside return array:

```
<?php
return [
    ...
    MongoDB\Laravel\MongoDBServiceProvider::class,
];
```

Save all the files and close them.

We need to refresh the configuration cache to apply the changes. Run the following command in your terminal:

```
php artisan config:cache
```

This command will clear the configuration cache and regenerate it with the new settings.

12.12.4.5 Step 5: Create Product Model

We will create a `Product` model to interact with the `products` collection in MongoDB. Run the following command to create the model:

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

Edit `app/Models/Product.php`:

```
<?php

namespace App\Models;

use MongoDB\Laravel\Eloquent\Model;

class Product extends Model
{
    protected $connection = 'mongodb';
    protected $collection = 'products';

    protected $fillable = ['name', 'price'];
}
```

Save the file and close it.

12.12.4.6 Step 6: Create Product Seeder

We will create a seeder to populate the `products` collection with sample data. Run the following command to create the seeder:

```
| php artisan make:seeder ProductSeeder
```

Edit `database/seeders/ProductSeeder.php`:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use App\Models\Product;

class ProductSeeder extends Seeder
{
    public function run()
    {
        Product::truncate();

        for ($i = 1; $i <= 100; $i++) {
            Product::create([
                'name' => "Product $i",
                'price' => rand(10, 500)
            ]);
        }
    }
}
```

Update `DatabaseSeeder.php` to call this seeder:

```
| class DatabaseSeeder extends Seeder
{
```

```

    public function run(): void
    {
        $this->call(ProductSeeder::class);
    }
}

```

Run the seeder:

```
| php artisan db:seed
```

You should verify the data in MongoDB Compass. Open the `paginationdb` database and select the `products` collection. You should see 100 products with random prices.

The screenshot shows the MongoDB Compass application interface. The left sidebar lists connections, with 'localhost:27017admin' expanded to show 'admin', 'blogdb', 'config', 'local', and 'paginationdb'. 'paginationdb' is selected, and its 'products' collection is shown in the main pane. The main pane displays four documents from the 'products' collection:

- `_id: ObjectId('67fbac2973aa912f6c0f2e12')`
`name : "Product 1"`
`price : 423`
`updated_at : 2025-04-13T12:20:57.992+00:00`
`created_at : 2025-04-13T12:20:57.992+00:00`
- `_id: ObjectId('67fbac2973aa912f6c0f2e13')`
`name : "Product 2"`
`price : 134`
`updated_at : 2025-04-13T12:20:57.994+00:00`
`created_at : 2025-04-13T12:20:57.994+00:00`
- `_id: ObjectId('67fbac2973aa912f6c0f2e14')`
`name : "Product 3"`
`price : 420`
`updated_at : 2025-04-13T12:20:57.995+00:00`
`created_at : 2025-04-13T12:20:57.995+00:00`
- `_id: ObjectId('67fbac2973aa912f6c0f2e15')`
`name : "Product 4"`
`price : 491`
`updated_at : 2025-04-13T12:20:57.996+00:00`
`created_at : 2025-04-13T12:20:57.996+00:00`

Figure 12.17: MongoDB Compass with seeded data.

12.12.4.7 Step 7: Create Product Controller

We will create a `ProductController` to handle the display of products. Run the following command to create the controller:

```
| php artisan make:controller ProductController
```

Edit `app/Http/Controllers/ProductController.php`:

```
<?php

namespace App\Http\Controllers;

use App\Models\Product;

class ProductController extends Controller
{
    public function index()
    {
        $products = Product::paginate(10);
        return view('products.index', compact('products'));
    }
}
```

Save the file and close it.

12.12.4.8 Step 8: Define Route

We define the route for the `ProductController` in the `routes/web.php` file. Open the `routes/web.php` file and add the following code:

```
use App\Http\Controllers\ProductController;

Route::get('/products', [ProductController::class, 'index']);
```

Save the file and close it.

12.12.4.9 Step 9: Create Blade Layout and View

We will create a layout file and a view to display the products. Create `layout.blade.php` in the `resources/views` directory.

Codes for `resources/views/layout.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Paginated Products</title>
    <link href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2.19/dist/tailwind.min.css" rel="stylesheet">
</head>
<body class="bg-gray-100 text-gray-800">
<div class="container mx-auto p-6">
    @yield('content')
</div>
</body>
</html>
```

Create a new directory called `products` in the `resources/views` directory. Inside the `products` directory, create a file called `index.blade.php`.

Codes for resources/views/products/index.blade.php:

```
@extends('layout')

@section('content')


# Paginated Products



| Name                  | Price                    |
|-----------------------|--------------------------|
| {{ \$product->name }} | \${{ \$product->price }} |



{{ $products->links() }}


@endsection
```

12.12.4.10 Step 10: Run the Application

After creating the views, we can run the application using the built-in PHP server. Run the following command in your terminal:

```
| php artisan serve
```

Visit <http://localhost:8000/products> to view the paginated product list.

The screenshot shows a web browser window with the title 'Paginated Products'. The URL in the address bar is '127.0.0.1:8000/products'. The page content is titled 'Paginated Products' and displays a table of products. The table has two columns: 'Name' and 'Price'. The data is as follows:

Name	Price
Product 1	\$423
Product 2	\$134
Product 3	\$420
Product 4	\$491
Product 5	\$481
Product 6	\$48
Product 7	\$55
Product 8	\$119
Product 9	\$267
Product 10	\$407

Below the table, a message says 'Showing 1 to 10 of 100 results'. At the bottom, there is a navigation bar with links from 1 to 10, and arrows for 'Previous' and 'Next'.

Figure 12.18: Paginated product list.

You should see a paginated list of products with 10 products per page. You can navigate through the pages using the pagination links at the bottom.

Clicking on the pagination links will load the next set of products without refreshing the page. This is done using Laravel's built-in pagination system, which works seamlessly with MongoDB.

The screenshot shows a web browser window titled "Paginated Products" with the URL "127.0.0.1:8000/products?page=5". The page displays a table of 10 products from a total of 100. The table has two columns: "Name" and "Price". The products listed are Product 41 through Product 50. Below the table, a message says "Showing 41 to 50 of 100 results" and there is a navigation bar with links from 1 to 10.

Name	Price
Product 41	\$228
Product 42	\$153
Product 43	\$62
Product 44	\$23
Product 45	\$265
Product 46	\$60
Product 47	\$387
Product 48	\$197
Product 49	\$263
Product 50	\$178

Figure 12.19: Pagination links.

12.12.5 Summary

In this lab, we built a Laravel 12 web app that connects to MongoDB and uses Laravel's built-in pagination system with the `mongodb/laravel-mongodb` driver. We seeded 100 products, displayed them with pagination, and styled the output using Tailwind CSS. This demonstrates how easy it is to paginate MongoDB data in Laravel while retaining a clean, modern UI using default pagination rendering.

12.13 Conclusion

In this chapter, we explored how to integrate Laravel with MongoDB using the `mongodb/laravel-mongodb` package. We created a simple CRUD application, implemented embedded relationships, and demonstrated data pagination. These exercises provided hands-on experience with MongoDB's flexible data model and how it can be effectively used within the Laravel framework.

We also learned how to use Laravel's built-in features, such as Eloquent models, controllers, and Blade views, to create a seamless user experience. By the end of this chapter, you should have a solid understanding of how to work with MongoDB in Laravel and be able to apply these concepts to your own projects.

OceanofPDF.com

13 Authentication and Authorization

13.1 Introduction

Authentication and authorization are two critical components in modern web application security. **Authentication** is the process of verifying the identity of users, ensuring they are who they claim to be. On the other hand, **authorization** determines what an authenticated user is allowed to do within the application. Laravel provides a powerful and flexible authentication and authorization system out of the box, simplifying the development of secure web applications. This chapter explores how Laravel 12 supports these features, from basic login systems to advanced role and permission controls.

13.2 Setting Up Authentication with Laravel Breeze

Laravel Breeze is a minimal and simple implementation of Laravel's authentication features. It includes routes, controllers, and views for login, registration, password resets, and more. To get started, ensure that you have a Laravel project set up. Then, install Laravel Breeze using Composer:

```
| composer require laravel/breeze --dev
|   php artisan breeze:install
|   npm install && npm run dev
|   php artisan migrate
```

Once the scaffolding is in place, Laravel provides you with a complete authentication system using either Blade or Inertia. This includes user registration, login, logout, email verification (optional), and password resets. These features lay the groundwork for building secure user access systems.

13.3 Understanding the Authentication Flow

Laravel's authentication flow is built on middleware and guards. Middleware such as `auth` ensures that only authenticated users can access certain routes. The `web` middleware group includes session state and CSRF protection, making it suitable for traditional web applications.

When a user logs in, Laravel stores their session using secure cookies. If they attempt to access a protected route, the system verifies their session and grants or denies access accordingly.

13.4 Managing Users and Profiles

Beyond login and registration, real-world applications often require users to manage their profiles. Laravel allows developers to easily implement features for updating user information, changing passwords, and even uploading avatars. This functionality typically involves creating a user controller with update methods and protected routes accessible only to logged-in users.

13.5 Securing Routes with Middleware

Protecting routes is essential to restrict access to specific parts of your application. Laravel uses middleware to intercept requests and determine whether a user should be granted access. The `auth` middleware is commonly used to enforce authentication. You can apply it to individual routes or entire route groups.

```
| Route::middleware('auth')->group(function () {  
|     Route::get('/dashboard', [DashboardController::class, 'index']);  
|});
```

Unauthenticated users attempting to access these routes will be redirected to the login page.

Hands-on Lab: Create a `/dashboard` route protected by the `auth` middleware and customize the content for logged-in users.

13.6 Implementing Authorization with Gates and Policies

Laravel's authorization system is based on **Gates** and **Policies**. Gates are closures that determine if a user is authorized to perform a specific action, whereas Policies are classes that group authorization logic around a particular model.

Policies can be automatically mapped to Eloquent models, allowing you to centralize your access control. For example, a `PostPolicy` can define who can update

or delete a post.

```
| php artisan make:policy PostPolicy --model=Post
```

After registering the policy, you can enforce authorization in your controllers or Blade views using methods like `authorize()` or `@can`.

13.7 Role and Permission Management

While gates and policies are powerful, larger applications often require more granular control. This is where the **Spatie Laravel Permission** package comes in. It enables role-based and permission-based access control.

You can define roles such as “admin”, “editor”, and “user”, then assign permissions to those roles. Laravel provides convenient methods to check roles and permissions in controllers and views.

```
| composer require spatie/laravel-permission  
| php artisan vendor:publish --provider="Spatie\Permission\PermissionServiceProvider"  
| php artisan migrate
```

Users can be assigned roles, and permissions can be granted or revoked as needed.

13.8 Exercise 39: Authentication and Authorization with Laravel 12 Breeze

13.8.1 Description

In this lab, we will implement a basic authentication and authorization system using Laravel 12’s built-in **Breeze** starter kit. Users will be able to register, log in, and access a protected profile page. The system will hash passwords using `bcrypt` and restrict access to the profile route using Laravel’s `auth` middleware. SQLite will be used as the database engine for simplicity.

13.8.2 Objectives

By the end of this lab, you will be able to:

- Scaffold authentication using Laravel Breeze
- Secure user passwords with `brypt`
- Create a protected route that returns the logged-in user’s profile

- Use `auth` middleware to protect routes
- Configure Laravel to use SQLite as the database

13.8.3 Prerequisites

Before you begin, make sure you have the following:

- PHP 8.2 or higher
- Composer
- Node.js and npm
- SQLite already installed
- Laravel 12 installed
- Visual Studio Code or any text editor

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

13.8.4 Steps

Here are the steps to complete this lab:

13.8.4.1 Step 1: Create New Laravel Project

We can create a new Laravel project using the Laravel installer:

```
| laravel new auth-lab  
| cd auth-lab  
| code .
```

Accept all the default options during installation. This will create a new Laravel project named `auth-lab` and use SQLite as the default database. The last command will open the project in Visual Studio Code.

You should see the project in Visual Studio Code.

13.8.4.2 Step 2: Configure SQLite Database

Since we use default SQLite database, we don't do anything. We can verify this by checking the `.env` file. Open the `.env` file and check the following lines:

```
| DB_CONNECTION=sqlite
```

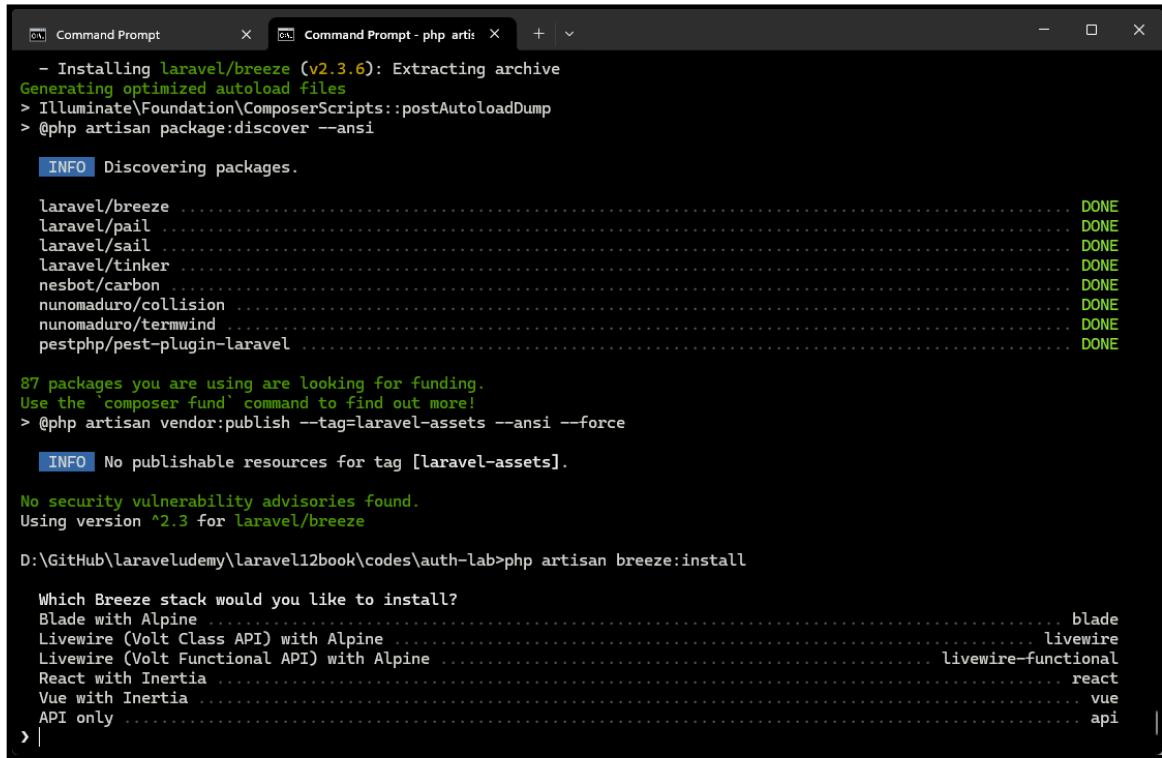
13.8.4.3 Step 3: Install Laravel Breeze

We will use Laravel Breeze for authentication scaffolding. Run the following commands in the terminal:

```
| composer require laravel/breeze --dev
```

Then, run the following command to install Breeze:

```
| php artisan breeze:install  
| npm install  
| php artisan migrate
```



```
Command Prompt      Command Prompt - php artis... + |   
- Installing laravel/breeze (v2.3.6): Extracting archive  
Generating optimized autoload files  
> Illuminate\Foundation\ComposerScripts::postAutoloadDump  
> @php artisan package:discover --ansi  
  
INFO | Discovering packages.  
  
laravel/breeze ..... DONE  
laravel/pail ..... DONE  
laravel/sail ..... DONE  
laravel/tinker ..... DONE  
nesbot/carbon ..... DONE  
nunomaduro/collision ..... DONE  
nunomaduro/termwind ..... DONE  
pestphp/pest-plugin-laravel ..... DONE  
  
87 packages you are using are looking for funding.  
Use the 'composer fund' command to find out more!  
> @php artisan vendor:publish --tag=laravel-assets --ansi --force  
  
INFO | No publishable resources for tag [laravel-assets].  
  
No security vulnerability advisories found.  
Using version ^2.3 for laravel/breeze  
  
D:\GitHub\laraveludem\laravel12book\codes\auth-lab>php artisan breeze:install  
Which Breeze stack would you like to install?  
Blade with Alpine ..... blade  
Livewire (Volt Class API) with Alpine ..... livewire  
Livewire (Volt Functional API) with Alpine ..... livewire-functional  
React with Inertia ..... react  
Vue with Inertia ..... vue  
API only ..... api  
> |
```

Figure 13.1: Install Laravel Breeze.

You can select the default options for the installation. This will install the Breeze package and set up the authentication scaffolding.

- Select `blade` as the frontend framework
- Dark mode is optional
- Select `yes` for the dark mode option

This will generate:

- Authentication scaffolding

- users table
- Registration, login, logout features

13.8.4.4 Step 4: Register and Login via Web Interface

After completing the installation, run the following command to start the development server:

```
| php artisan serve
```

Visit <http://localhost:8000>

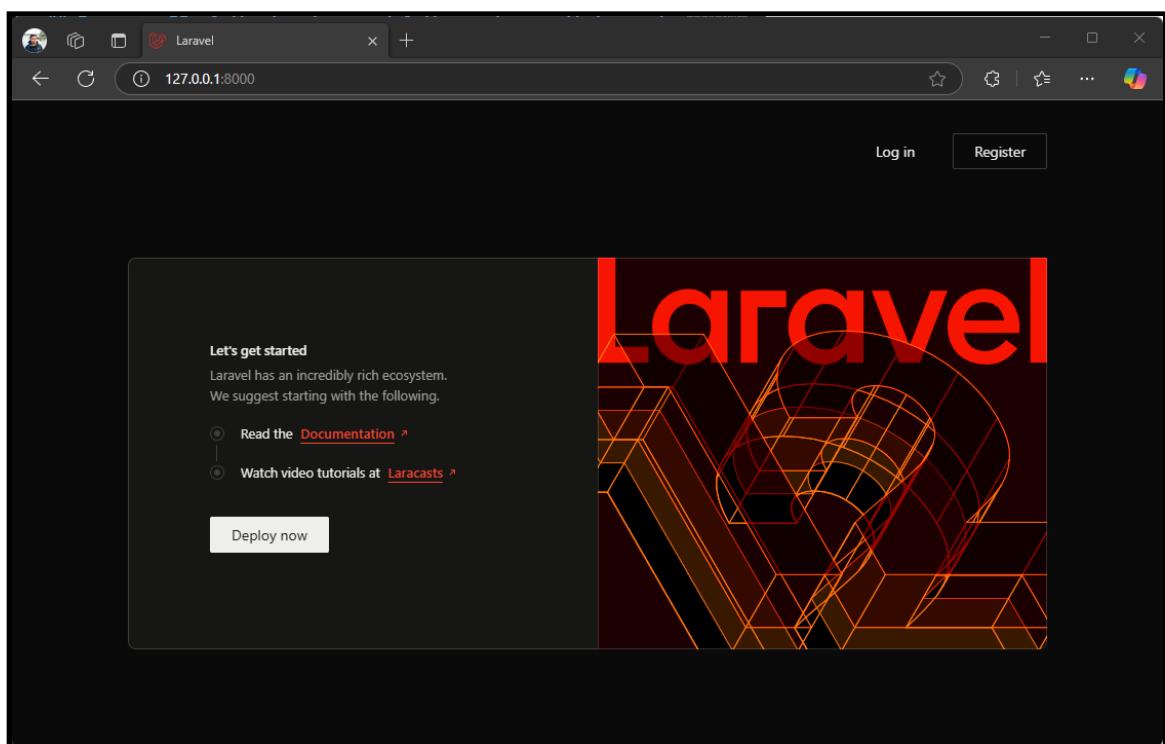


Figure 13.2: Laravel Breeze.

Then, we can register a new user by clicking on the **Register** link. Fill in the registration form with your details and click **Register**. This will create a new user in the database.

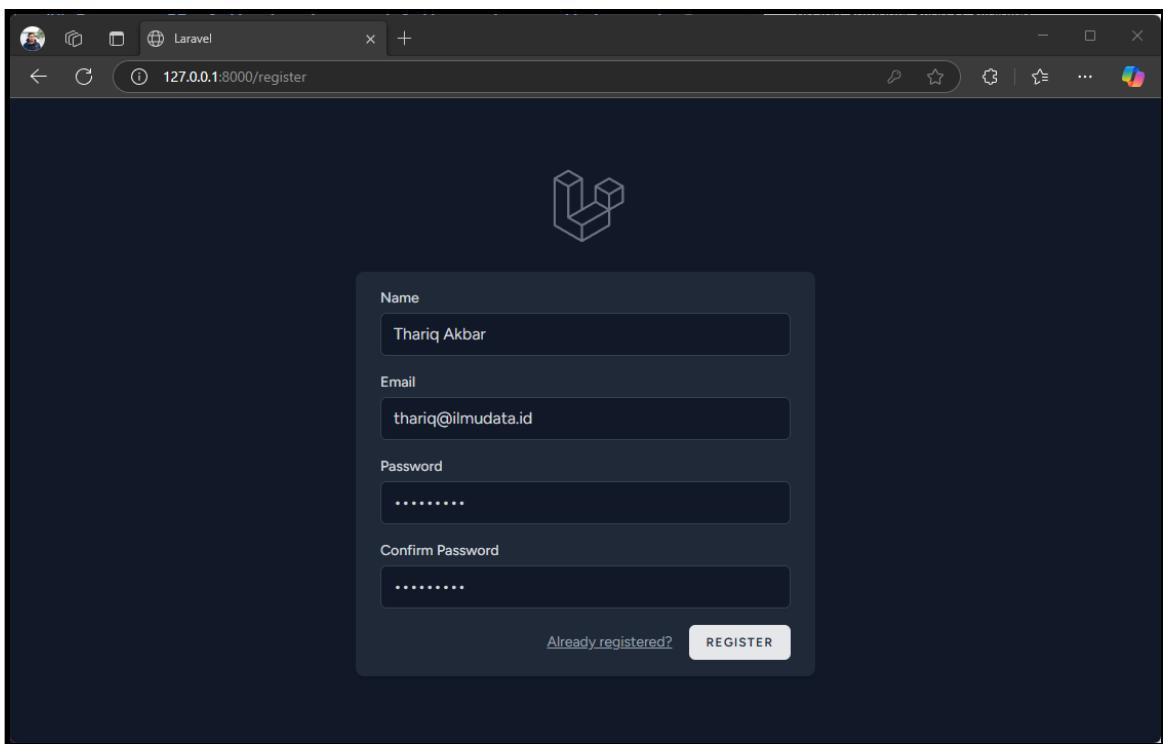


Figure 13.3: Register new user.

After registering, you will be redirected to the dashboard page.

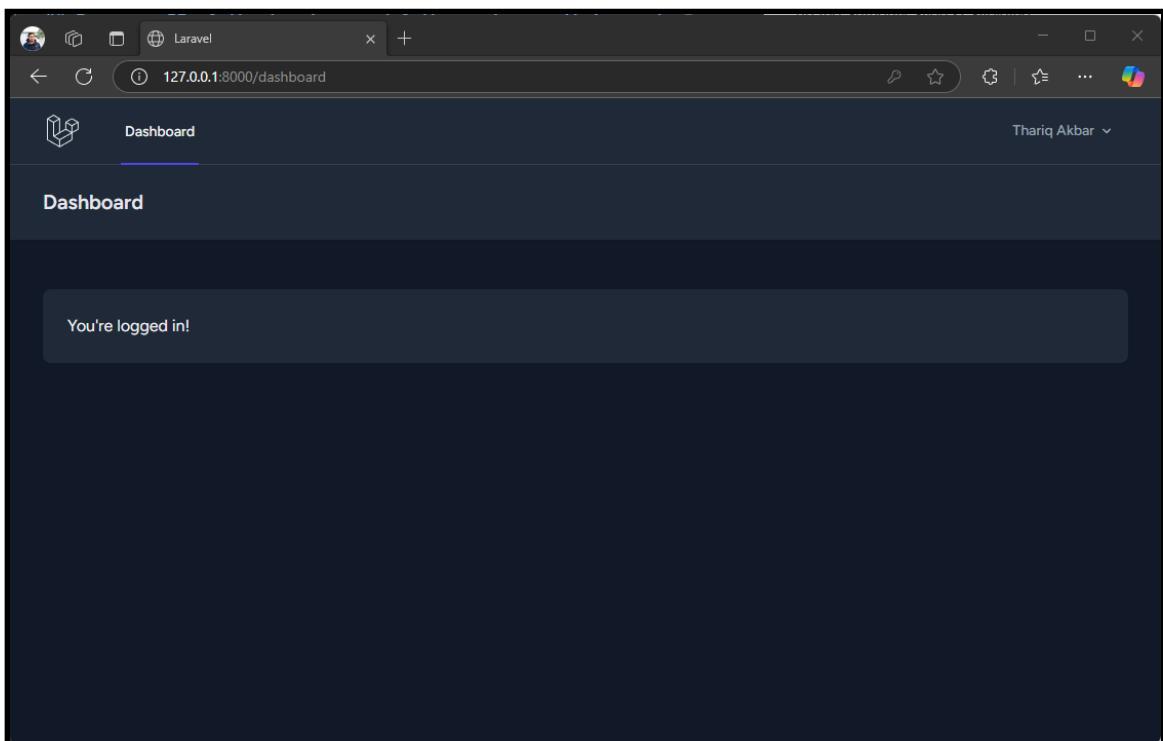


Figure 13.4: Dashboard page.

Try to log out and log in again using the credentials you just registered. You can do this by clicking on the **Login** link in the top right corner.

13.8.4.5 Step 5: Create a Protected Profile Route

We can verify the authentication system by creating a protected route that returns the logged-in user's profile.

Open `routes/web.php`:

```
<?php

use App\Http\Controllers\ProfileController;
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    return view('welcome');
});

Route::get('/dashboard', function () {
    return view('dashboard');
})->middleware(['auth', 'verified'])->name('dashboard');

Route::middleware('auth')->group(function () {
    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');
    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');
    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');
});

require __DIR__.'/auth.php';
```

This code creates a new route `/profile` that is protected by the `auth` middleware. This means that only authenticated users can access this route.

Try to add `/myprofile` and return JSON representation of the logged-in user:

```
...

Route::middleware('auth')->group(function () {
    ...

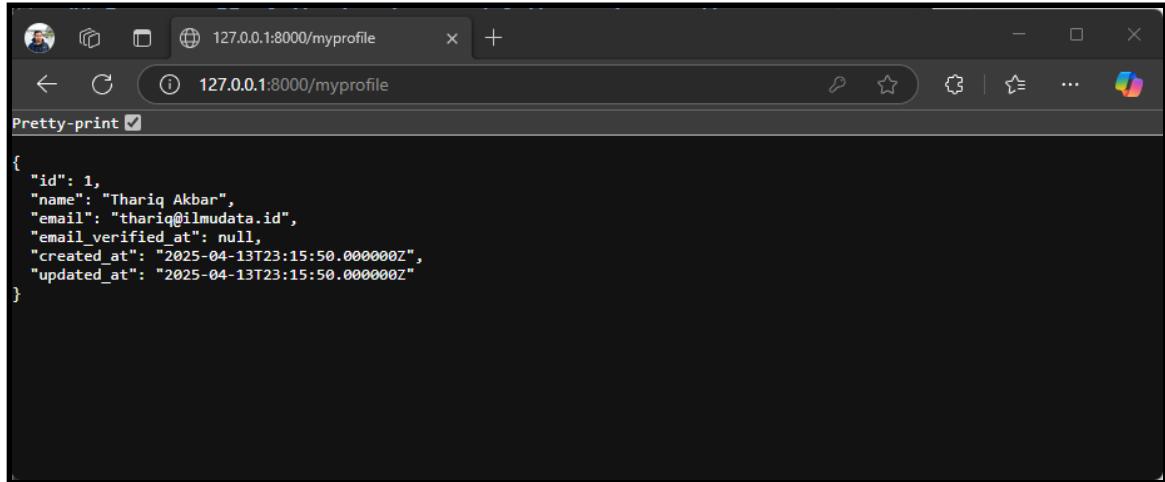
    Route::get('/myprofile', function () {
        return Auth::user();
    })->name('myprofile');
});
```

This will return a JSON representation of the logged-in user's data.

13.8.4.6 Step 6: Test Profile Access

We can run the server and test the `/myprofile` route. Open your browser and visit `http://localhost:8000/myprofile`. You should see the profile page.

If you are not logged in, you will be redirected to the login page. If you are logged in, you will see the JSON representation of the logged-in user.

A screenshot of a web browser window. The address bar shows the URL "127.0.0.1:8000/myprofile". Below the address bar, there is a "Pretty-print" checkbox which is checked. The main content area of the browser displays a JSON object:

```
{  
  "id": 1,  
  "name": "Thariq Akbar",  
  "email": "thariq@ilmudata.id",  
  "email_verified_at": null,  
  "created_at": "2025-04-13T23:15:50.000000Z",  
  "updated_at": "2025-04-13T23:15:50.000000Z"  
}
```

The JSON object contains fields for id, name, email, email verification status, creation timestamp, and update timestamp.

Figure 13.5: Profile in JSON format.

You can edit the profile by clicking on the **Profile** link. This will take you to the profile edit page where you can update your name, email, and password.

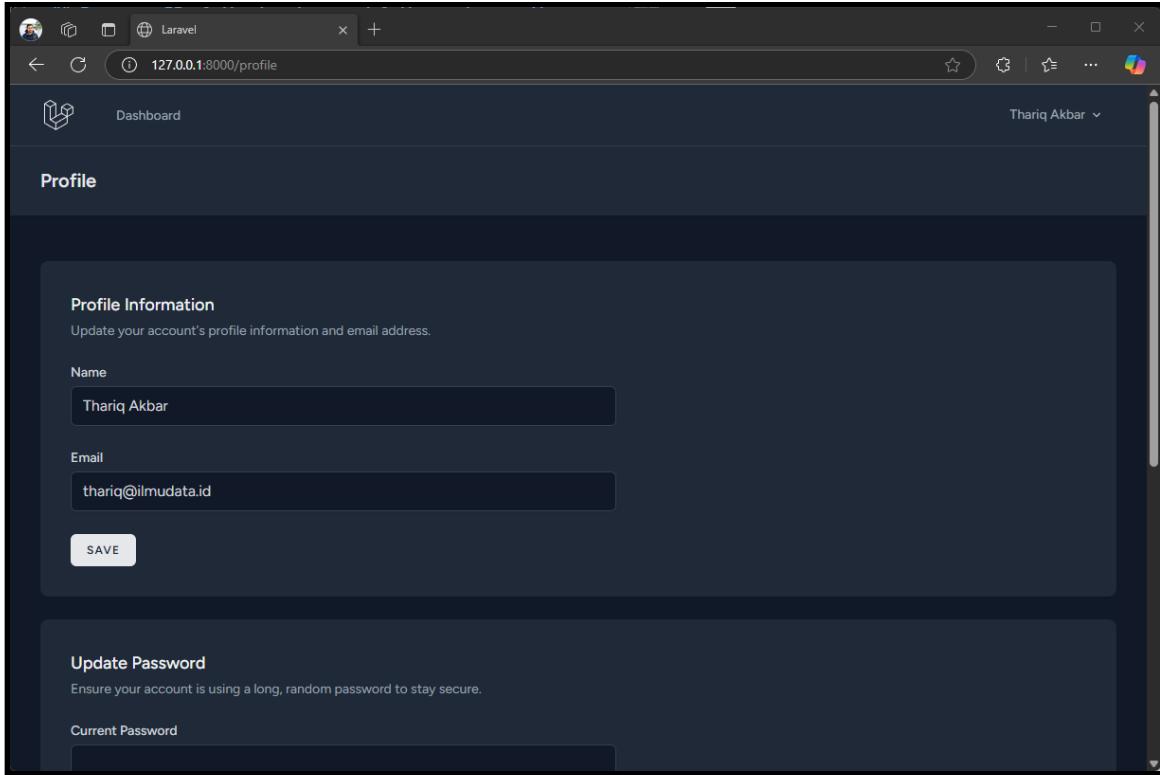


Figure 13.6: Edit profile page.

13.8.5 Summary

In this hands-on lab, we built a basic authentication and authorization system using Laravel Breeze with SQLite. We:

- Installed and configured Laravel Breeze
- Registered users and hashed their passwords with bcrypt
- Created a protected `/profile` route
- Used `auth` middleware to restrict access

This lab introduces the fundamentals of Laravel's built-in authentication system, paving the way for more advanced topics like policies, gates, and roles in future chapters.

13.9 Exercise 40: Restrict Access Based on Role in Laravel 12

13.9.1 Description

In this lab, you will learn how to implement role-based access control in Laravel 12. We'll define three roles: `admin`, `manager`, and `user`. Each role will have its own view and only users with the correct role will be able to access their designated page. All authenticated users will be able to access a shared “general view.” The roles will be assigned using a seeder, and SQLite will be used as the database backend.

13.9.2 Objectives

By the end of this lab, you will be able to:

- Define roles and assign them to users
- Create a seeder for multiple users with roles
- Show different views for each role
- Restrict route access based on role using middleware
- Provide a common view for all authenticated users

13.9.3 Prerequisites

- Laravel 12 installed
- PHP 8.2 or higher
- Composer
- Node.js and npm
- SQLite installed and configured
- Laravel Breeze for authentication
- Visual Studio Code or any editor

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

13.9.4 Steps

Here are the steps to complete this lab.

13.9.4.1 Step 1: Create Laravel Project

We can create a new Laravel project using the Laravel installer:

```
| laravel new role-lab  
| cd role-lab
```

```
| code .
```

Acept all the default options during installation. This will create a new Laravel project named `new role-lab` and use SQLite as the default database. The last command will open the project in Visual Studio Code.

You should see the project in Visual Studio Code.

13.9.4.2 Step 2: Configure SQLite

Since we use default SQLite database, we don't do anything. We can verify this by checking the `.env` file. Open the `.env` file and check the following lines:

```
| DB_CONNECTION=sqlite
```

13.9.4.3 Step 3: Install Laravel Breeze

We will use Laravel Breeze for authentication scaffolding. Run the following commands in the terminal:

```
| composer require laravel/breeze --dev
| php artisan breeze:install
| npm install
| php artisan migrate
```

We can select the default options for the installation. This will install the Breeze package and set up the authentication scaffolding.

- Select `blade` as the frontend framework
- Dark mode is optional
- Select `yes` for the dark mode option

13.9.4.4 Step 4: Add `role` Field to Users Table

We need to add a `role` field to the `users` table. This will allow us to assign roles to users.

Create a migration:

```
| php artisan make:migration add_role_to_users_table --table=users
```

Edit the migration file:

```
public function up(): void
{
    Schema::table('users', function (Blueprint $table) {
        $table->string('role')->default('user');
    });
}
```

Then run:

```
| php artisan migrate
```

You can check the `users` table in the SQLite database to verify that the `role` column has been added.

13.9.4.5 Step 5: Seed Users with Different Roles

We will create some users with different roles. We use `DatabaseSeeder.php` to seed the database with users.

Edit `database/seeders/DatabaseSeeder.php`:

```
use App\Models\User;
use Illuminate\Support\Facades\Hash;

public function run(): void
{
    User::create([
        'name' => 'Admin User',
        'email' => 'admin@ilmudata.id',
        'password' => Hash::make('password123'),
        'role' => 'admin',
    ]);

    User::create([
        'name' => 'Manager User',
        'email' => 'manager@ilmudata.id',
        'password' => Hash::make('password123'),
        'role' => 'manager',
    ]);

    User::create([
        'name' => 'General User',
        'email' => 'user@ilmudata.id',
        'password' => Hash::make('password123'),
        'role' => 'user',
    ]);
}
```

Then run:

```
| php artisan db:seed
```

You can check the `users` table in the SQLite database to verify that the users have been added with their respective roles.

id	name	email	password	role	remember_token
1	Admin User	admin@ilmudata.id	\$2y\$12\$flDRw4Ra64T82WPoN3TN8ejbxDbBlhttLYE1K2...	admin	NULL
2	Manager User	manager@ilmudata.id	\$2y\$12\$O4HfWpTC.mF.ozsF8bZO.8vJnPT5tQ/q/c5PV...	manager	NULL
3	General User	user@ilmudata.id	\$2y\$12\$z/1eEclRhw6GCZJPSeUfxO50xuUcAw0)XZckgxX...	user	NULL

Figure 13.7: Users table with roles.

13.9.4.6 Step 6: Create Role Middleware

We will create a middleware to check the user's role. This middleware will restrict access to certain routes based on the user's role.

Generate middleware:

```
| php artisan make:middleware RoleMiddleware
```

Update `app/Http/Middleware/RoleMiddleware.php`:

```
namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class RoleMiddleware
{
    public function handle(Request $request, Closure $next, string $role): Response
    {
        if ($request->user() && $request->user()->role === $role) {
            return $next($request);
        }

        abort(403, 'Unauthorized');
    }
}
```

Register it in `bootstrap/app.php` inside `withMiddleware()`.

```

<?php

use Illuminate\Foundation\Application;
use Illuminate\Foundation\Configuration\Exceptions;
use Illuminate\Foundation\Configuration\Middleware;

use App\Http\Middleware\RoleMiddleware;

return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
    )
    ->withMiddleware(function (Middleware $middleware) {
        //
        $middleware->alias([
            'role' => RoleMiddleware::class,
        ]);
    })
    ->withExceptions(function (Exceptions $exceptions) {
        //
    })->create();

```

Save the file.

13.9.4.7 Step 7: Create Views for Each Role

We will create different views for each role. In `resources/views`, create the following files:

In `resources/views`, create:

- `admin.blade.php`
- `manager.blade.php`
- `user.blade.php`
- `all.blade.php`

Here are the **complete Blade views** for `admin`, `manager`, `user`, and `all`, all using the **same structure as** `dashboard.blade.php` with customized headers and messages for each role.

Codes for `resources/views/admin.blade.php`.

```

<x-app-layout>
    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 dark:text-gray-200 leading-tight">
            {{ __('Admin Dashboard') }}
        </h2>
    </x-slot>

```

```

<div class="py-12">
    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
        <div class="bg-white dark:bg-gray-800 overflow-hidden shadow-sm sm:rounded-lg">
            <div class="p-6 text-gray-900 dark:text-gray-100">
                {{ __("Welcome, Admin! You have full access.") }}
            </div>
        </div>
    </div>
</x-app-layout>

```

Codes for resources/views/manager.blade.php.

```

<x-app-layout>
    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 dark:text-gray-200 leading-tight">
            {{ __('Manager Dashboard') }}
        </h2>
    </x-slot>

    <div class="py-12">
        <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
            <div class="bg-white dark:bg-gray-800 overflow-hidden shadow-sm sm:rounded-lg">
                <div class="p-6 text-gray-900 dark:text-gray-100">
                    {{ __("Welcome, Manager! You can manage and monitor resources.") }}
                </div>
            </div>
        </div>
    </div>
</x-app-layout>

```

Codes for resources/views/user.blade.php.

```

<x-app-layout>
    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 dark:text-gray-200 leading-tight">
            {{ __('User Dashboard') }}
        </h2>
    </x-slot>

    <div class="py-12">
        <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
            <div class="bg-white dark:bg-gray-800 overflow-hidden shadow-sm sm:rounded-lg">
                <div class="p-6 text-gray-900 dark:text-gray-100">
                    {{ __("Welcome, User! You have limited access.") }}
                </div>
            </div>
        </div>
    </div>
</x-app-layout>

```

Codes for resources/views/all.blade.php.

```

<x-app-layout>
    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 dark:text-gray-200 leading-tight">
            {{ __('General Dashboard') }}
        </h2>
    </x-slot>

```

```
<div class="py-12">
    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
        <div class="bg-white dark:bg-gray-800 overflow-hidden shadow-sm sm:rounded-lg">
            <div class="p-6 text-gray-900 dark:text-gray-100">
                {{ __("Welcome! This view is accessible by all authenticated roles.")}}
            </div>
        </div>
    </div>
</x-app-layout>
```

These views are styled consistently with Laravel Breeze's default `dashboard.blade.php` using **Tailwind CSS components** and the `<x-app-layout>` wrapper.

Save all the files.

13.9.4.8 Step 8: Define Routes for Role-Based Views

We will define routes for each role and the common view. Open `routes/web.php` and add the following code:

```
use Illuminate\Support\Facades\Route;

Route::middleware('auth')->group(function () {
    Route::get('/all', function () {
        return view('all');
    });

    Route::get('/admin', function () {
        return view('admin');
    })->middleware('role:admin');

    Route::get('/manager', function () {
        return view('manager');
    })->middleware('role:manager');

    Route::get('/user', function () {
        return view('user');
    })->middleware('role:user');
});
```

13.9.4.9 Step 9: Login and Test Role-Based Access

After completing the above steps, you can run the server:

```
| php artisan serve
```

Visit `http://localhost:8000` and log in using the seeded users:

- admin@ilmudata.id / password
- manager@ilmudata.id / password
- user@ilmudata.id / password

Themn, try accessing:

- /admin
- /manager
- /user
- /all

For instance, we sign in using admin@ilmudata.id and try to access /admin, /manager, and /all. You should see the admin dashboard.

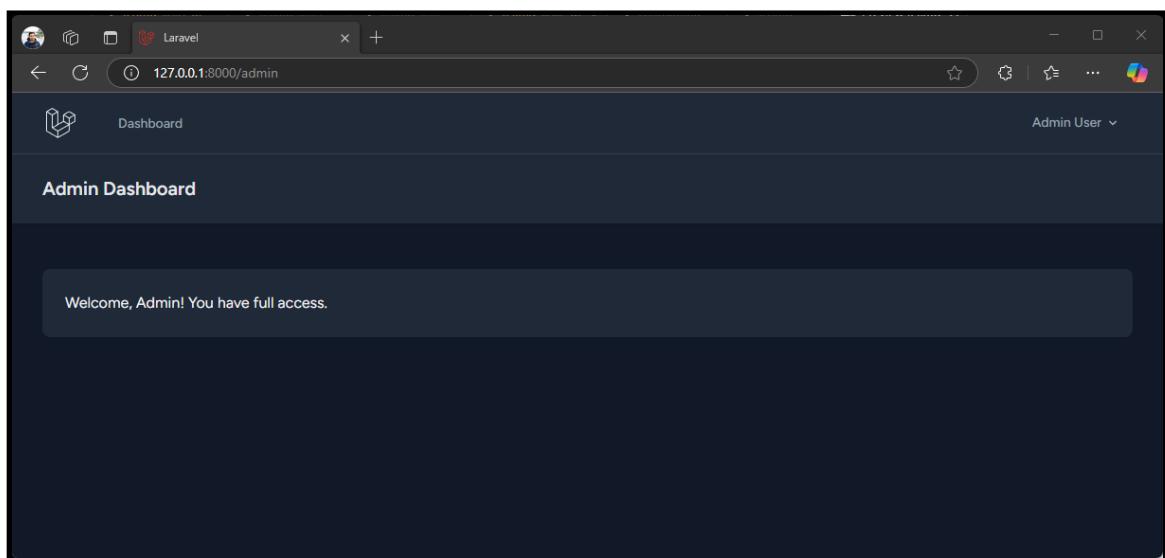


Figure 13.8: Admin /admin dashboard.

If you try to access /manager or /user, you will get a 403 Unauthorized error.

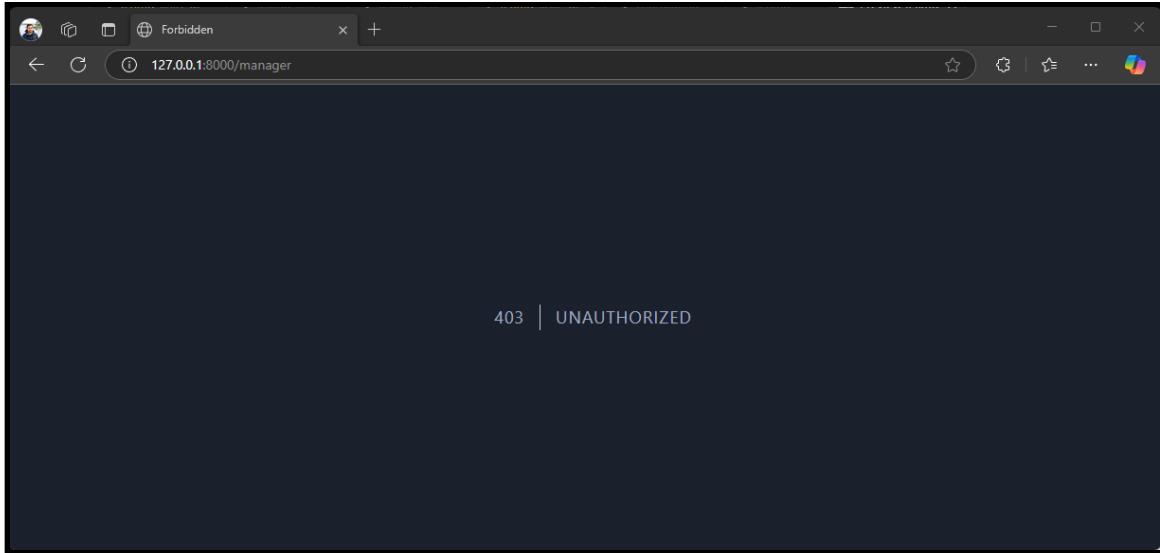


Figure 13.9: Unauthorized access.

Each role should only access their route; all can access /all.

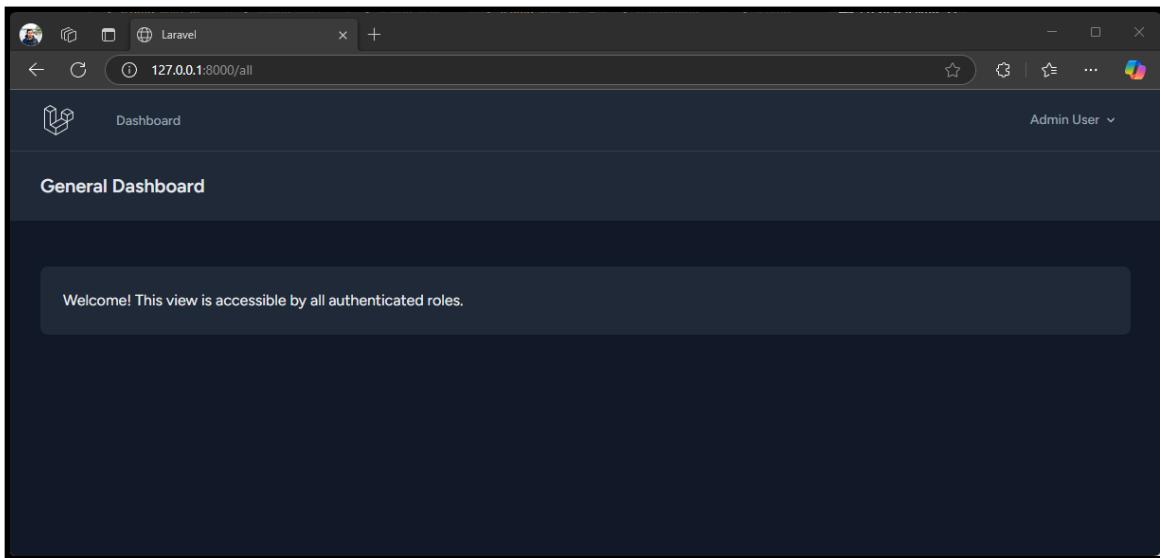


Figure 13.10: All authenticated users can access /all.

13.9.5 Summary

In this lab, you implemented role-based access control in Laravel 12. You learned to:

- Add and seed roles (`admin`, `manager`, `user`)
- Create a custom middleware for role checking
- Create and protect views specific to each role

- Allow a common view for all roles

This lab sets the foundation for scalable RBAC (Role-Based Access Control) systems in Laravel apps.

13.10 Exercise 41: Authentication with Google using Laravel Socialite in Laravel 12

13.10.1 Description

In this lab, you will implement user authentication via **Google Login** using Laravel's official package **Socialite**. Once users are authenticated with Google, they will be logged into the application and redirected to their profile page. This profile page will be protected using Laravel's `auth` middleware. The database backend used will be SQLite for simplicity.

13.10.2 Objectives

By the end of this lab, you will be able to:

- Configure Laravel Socialite for Google OAuth
- Create login functionality using a Google account
- Store authenticated user data in SQLite
- Protect routes using the `auth` middleware

13.10.3 Prerequisites

- Laravel 12 installed
- PHP 8.2 or higher
- Composer
- Node.js and npm
- SQLite installed and configured
- Google Cloud Console account with OAuth 2.0 credentials
- Laravel Breeze installed (optional, for base auth setup)
- A code editor like Visual Studio Code

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

13.10.4 Steps

Here are the steps to complete this lab.

13.10.4.1 Step 1: Create Laravel Project

We can create a new Laravel project using the Laravel installer:

```
| laravel new google-login-lab  
| cd google-login-lab  
| code .
```

Acept all the default options during installation. This will create a new Laravel project named `google-login-lab` and use SQLite as the default database. The last command will open the project in Visual Studio Code.

You should see the project in Visual Studio Code.

13.10.4.2 Step 2: Configure SQLite

Since we use default SQLite database, we don't do anything. We can verify this by checking the `.env` file. Open the `.env` file and check the following lines:

```
| DB_CONNECTION=sqlite
```

13.10.4.3 Step 3: Install Laravel Socialite

We will use Laravel Socialite for Google authentication. Run the following command in the terminal:

```
| composer require laravel/socialite
```

Laravel 12 will automatically register the Socialite service provider and facade.

13.10.4.4 Step 4: Set Up Google OAuth

We need to create a Google Cloud project and set up OAuth 2.0 credentials.

1. Go to [Google Cloud Console](#).
2. Create a project and navigate to **APIs & Services > Credentials**.
3. Create **OAuth 2.0 Client ID**:
 - App type: Web application

- o Authorized redirect URI: `http://localhost:8000/auth/google/callback`

4. Save the Client ID and Client Secret.

Name	Creation date	Type	Client ID
laravel1	Apr 14, 2025	Web application	478598820

Figure 13.11: Google Cloud Console.

Update `.env`:

```
GOOGLE_CLIENT_ID=your-google-client-id
GOOGLE_CLIENT_SECRET=your-google-client-secret
GOOGLE_REDIRECT_URI=http://localhost:8000/auth/google/callback
```

Change `your-google-client-id` and `your-google-client-secret` with the values you got from Google Cloud Console.

Make sure to set the `GOOGLE_REDIRECT_URI` to match the one you set in the Google Cloud Console. This is where Google will redirect users after they authenticate.

13.10.4.5 Step 5: Configure Socialite in `services.php`

We need to configure Socialite to use Google as the authentication provider.

Edit `config/services.php`:

```
'google' => [
    'client_id' => env('GOOGLE_CLIENT_ID'),
    'client_secret' => env('GOOGLE_CLIENT_SECRET'),
    'redirect' => env('GOOGLE_REDIRECT_URI'),
],
```

13.10.4.6 Step 6: Update the User Model

Make sure your `User` model allows `name`, `email`, and `password` to be filled:

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

Also, run:

```
| php artisan make:migration add_google_id_to_users_table --table=users
```

We will add a `google_id` column to the `users` table to store the Google ID of the authenticated user.

Update the migration:

```
| public function up(): void
| {
|     Schema::table('users', function (Blueprint $table) {
|         $table->string('google_id')->nullable()->unique();
|     });
| }
```

Run the migration:

```
| php artisan migrate
```

13.10.4.7 Step 7: Create Google Auth Controller

We will create a controller to handle Google authentication. Run the following command to create a new controller:

```
| php artisan make:controller GoogleAuthController
```

In `app/Http/Controllers/GoogleAuthController.php`:

```
| <?php
|
| namespace App\Http\Controllers;
|
| use Illuminate\Http\Request;
| use Socialite;
| use App\Models\User;
| use Illuminate\Support\Facades\Auth;
|
| class GoogleAuthController extends Controller
| {
|     public function redirectToGoogle()
|     {
|         return Socialite::driver('google')->redirect();
|     }
| }
```

```

public function handleGoogleCallback()
{
    $googleUser = Socialite::driver('google')->stateless()->user();

    // Try to find the user by email
    $user = User::where('email', $googleUser->getEmail())->first();

    if ($user) {
        // If user exists but has no google_id, update it
        if (!$user->google_id) {
            $user->update([
                'google_id' => $googleUser->getId(),
            ]);
        }
    } else {
        // No user with that email, create a new one
        $user = User::create([
            'name' => $googleUser->getName(),
            'email' => $googleUser->getEmail(),
            'google_id' => $googleUser->getId(),
            'password' => bcrypt(str()>random(16)), // Placeholder
        ]);
    }

    Auth::login($user);

    return redirect('/profile');
}

public function logout()
{
    Auth::logout();
    return redirect('/');
}
}

```

13.10.4.8 Step 8: Create Routes

We need to create routes for Google authentication. Open `routes/web.php` and add the following code:

```

<?php

use Illuminate\Support\Facades\Route;

use App\Http\Controllers\GoogleAuthController;
use Illuminate\Support\Facades\Auth;

Route::get('/auth/google', [GoogleAuthController::class, 'redirectToGoogle']);
Route::get('/auth/google/callback', [GoogleAuthController::class, 'handleGoogleCallback']);

Route::middleware(['auth'])->group(function () {
    Route::get('/profile', function () {
        return Auth::user();
    });
});

```

```
| Route::get('/logout', [GoogleAuthController::class, 'logout'])->middleware('auth');

| Route::get('/', function () {
|     return view('welcome');
| });

|
```

Save the file.

13.10.4.9 Step 9: Add Login Button to Welcome View

We need to add a button to the welcome view to initiate Google login. Open `resources/views/welcome.blade.php` and add the following code:

Edit `resources/views/welcome.blade.php`:

```
...
    <ul class="flex gap-3 text-sm leading-normal">
        <li>
            <a href="https://cloud.laravel.com" target="_blank" class="inline-block" style="background-color: #eeeeec; border: 1px solid #eeeeec; color: #1C1C1A; text-decoration: none; padding: 5px 1.5px; border-radius: 10px; border: 1px solid black; text-align: center; font-weight: bold; font-size: 14px; margin-right: 10px;">
                Deploy now
            </a>
            <a href="{{ url('auth/google') }}" target="_blank" class="inline-block" style="background-color: #eeeeec; border: 1px solid #eeeeec; color: #1C1C1A; text-decoration: none; padding: 5px 1.5px; border-radius: 10px; border: 1px solid black; text-align: center; font-weight: bold; font-size: 14px;">
                Login with Google
            </a>
        </li>
    </ul>
    ...
```

We added a button that links to the `/auth/google` route near the **Deploy now** button. This will redirect users to Google for authentication.

13.10.4.10 Step 10: Run and Test

Start the Laravel server:

```
| php artisan serve
```

Visit:

- <http://localhost:8000> → click **Login with Google**

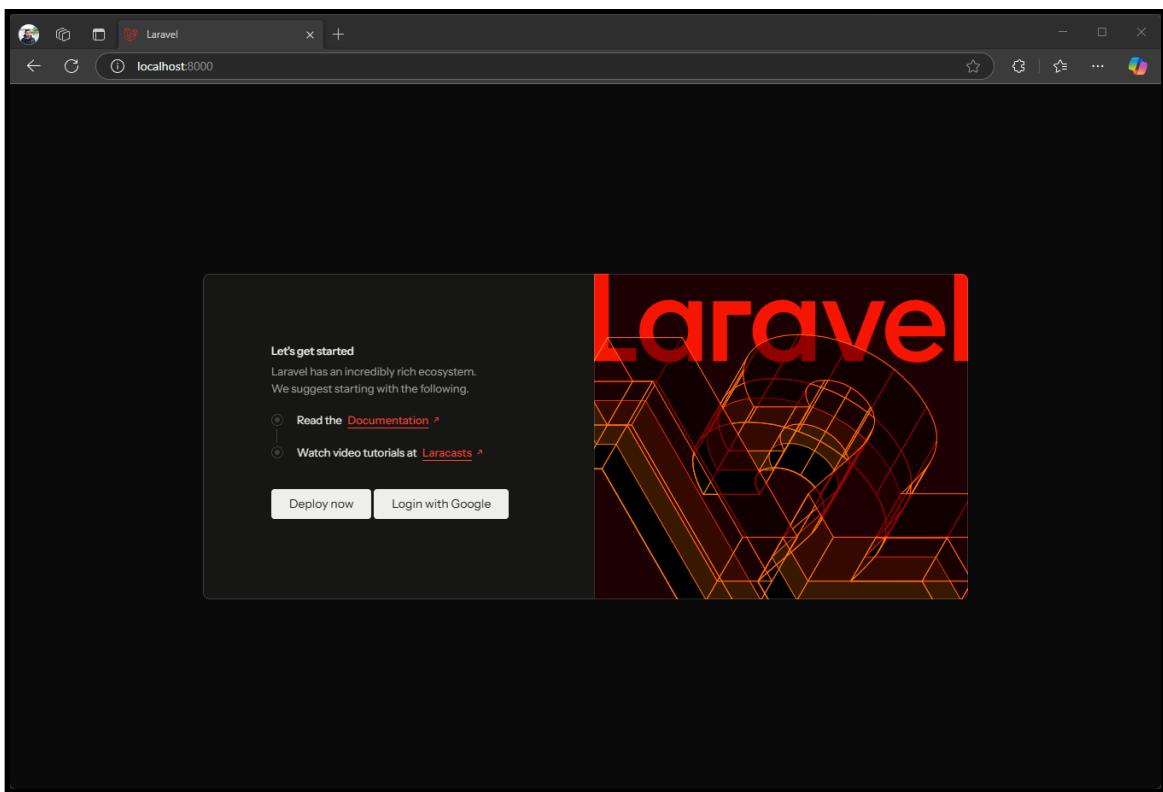


Figure 13.12: Login with Google.

- After clicking the button, you will be redirected to Google for authentication.

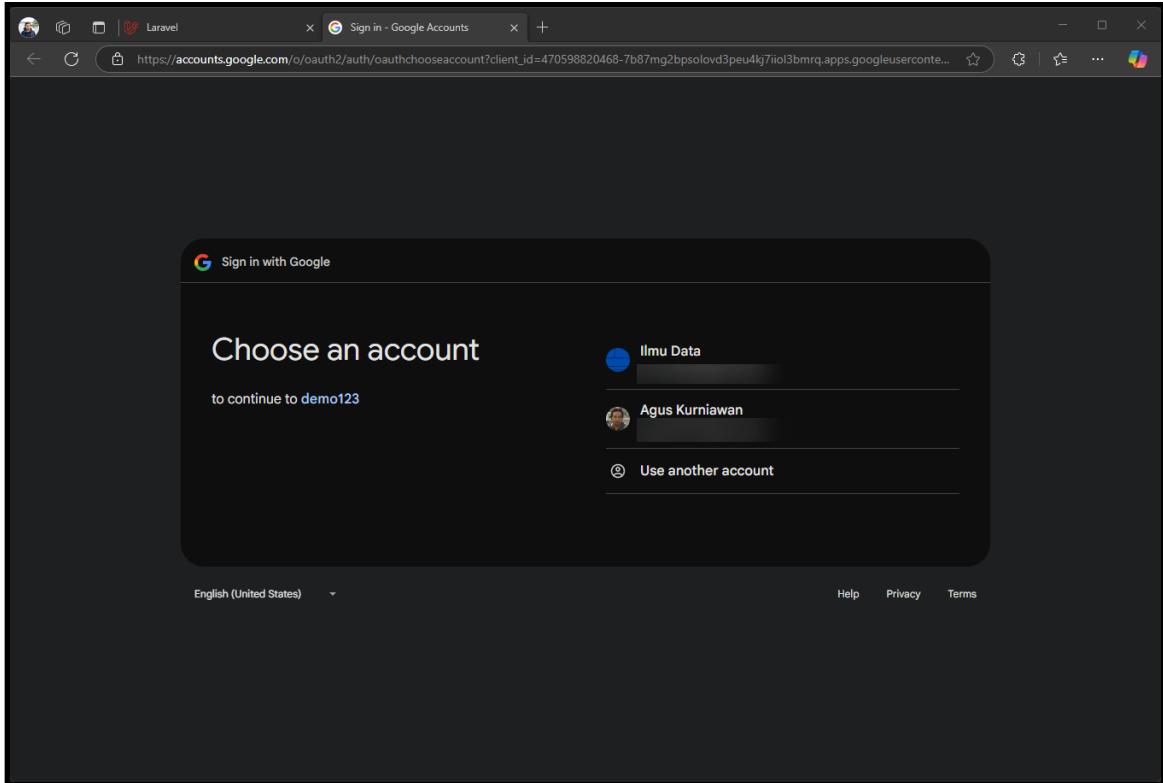


Figure 13.13: Google authentication.

- Select your Google account and allow permissions.
- After login → you will be redirected to `/profile` and see the user JSON

```
{  
  "id": 1,  
  "name": "Ilmu Data",  
  "email": "ilmu.data18@gmail.com",  
  "email_verified_at": null,  
  "created_at": "2025-04-14T09:11:58.000000Z",  
  "updated_at": "2025-04-14T09:11:58.000000Z",  
  "google_id": null  
}
```

Figure 13.14: Profile page.

If not logged in, trying to access `/profile` will redirect to the login page or throw 403 (depending on guard).

If you want to log out, visit `/logout`:

```
| http://localhost:8000/logout
```

This will log you out and redirect to the home page.

13.10.5 Summary

In this hands-on lab, you implemented authentication using Google OAuth in Laravel 12 with Socialite. You learned how to:

- Set up and configure Laravel Socialite
- Authenticate users via Google
- Store user profile data
- Protect routes with the `auth` middleware

This integration is ideal for simplifying user login and increasing security with trusted identity providers like Google.

13.11 Conclusion

In this chapter, we explored the authentication and authorization features of Laravel 12. We learned how to set up user authentication using Laravel Breeze, manage user profiles, and secure routes with middleware. We also implemented role-based access control using custom middleware and Laravel's built-in authorization features. Finally, we integrated Google OAuth authentication using Laravel Socialite.

These features are essential for building secure and user-friendly web applications. In the next chapter, we will explore how to build RESTful APIs using Laravel 12.

14 REST API Development with Laravel

In this chapter, you will learn how to build RESTful APIs using Laravel 12. You will explore the core principles of REST, how Laravel facilitates API development, and how to build a simple yet complete API for a product catalog. The chapter also covers input validation, JSON responses, authentication using Laravel Sanctum, pagination, and testing. By the end of this chapter, you will have a solid foundation for building professional APIs using Laravel.

14.1 Introduction to REST APIs

REST (Representational State Transfer) is an architectural style that defines a set of constraints for building web services. RESTful APIs use HTTP methods such as GET, POST, PUT, and DELETE to perform operations on resources. Each resource is identified by a URL and typically represented in JSON format.

Compared to older protocols like SOAP, REST is lightweight, stateless, and easier to implement. In Laravel, RESTful APIs can be created quickly using built-in routing and controller features. A well-designed RESTful API improves client-server communication, especially in mobile and single-page applications.

14.2 Setting Up the Laravel API Project

To get started, you can either use an existing Laravel project or create a new one using the command:

```
| laravel new laravel-api-demo
```

Once the project is set up, update the `.env` file with your database configuration. Make sure the database is created and accessible. Laravel uses `routes/api.php` to define routes that are specifically intended for API requests. These routes are automatically prefixed with `/api` and are stateless by default.

14.3 Defining API Routes

Laravel provides a convenient way to define RESTful routes using the `Route::apiResource` method. This automatically generates routes for the standard CRUD operations. Here is an example of defining API routes for a `ProductController`:

```
| Route::apiResource('products', ProductController::class);
```

This single line defines routes for `index`, `store`, `show`, `update`, and `destroy` actions. You can also group these routes with middleware, such as authentication, for better control and security.

14.4 Building the Product API

Begin by creating a `Product` model along with its migration file:

```
| php artisan make:model Product -m
```

In the generated migration file, add the necessary fields such as `name`, `description`, `price`, and `stock`. After defining the schema, run the migration:

```
| php artisan migrate
```

Next, create a controller to handle API logic:

```
| php artisan make:controller API/ProductController --api
```

This generates a resource controller with all necessary methods. You can now implement logic in each method to perform CRUD operations using the `Product` model.

14.5 Using JSON Resources for API Responses

Laravel offers a powerful way to format API responses using Resource classes. To create a resource for the `Product` model, use the following command:

```
| php artisan make:resource ProductResource
```

In the `ProductResource` class, define how the model should be represented in the API response. This allows for a consistent and clean structure of your JSON output:

```
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'price' => $this->price,
        'stock' => $this->stock,
        'created_at' => $this->created_at->toDateString(),
    ];
}
```

Using resources ensures that your API responses remain consistent across endpoints and are easy to maintain.

14.6 Input Validation and Error Handling

To validate incoming API requests, it is recommended to use Laravel Form Requests. Create request classes using:

```
| php artisan make:request StoreProductRequest  
| php artisan make:request UpdateProductRequest
```

In these classes, define the validation rules. Laravel automatically handles failed validation by returning JSON errors with the appropriate HTTP status code (422 Unprocessable Entity). Proper validation not only ensures data integrity but also improves the developer experience for API consumers.

14.7 Securing APIs with Laravel Sanctum

To protect your API, you can use Laravel Sanctum, which provides token-based authentication. First, install Sanctum:

```
| composer require laravel/sanctum
```

Then publish the configuration and run the migrations:

```
| php artisan vendor:publish --provider="Laravel\\Sanctum\\SanctumServiceProvider"  
| php artisan migrate
```

Enable Sanctum by adding the middleware `auth:sanctum` to your API routes. Users can authenticate and receive a token, which must be included in the Authorization header for subsequent requests. Sanctum is ideal for single-page applications and mobile apps.

14.8 Pagination and Filtering

To handle large datasets, Laravel's built-in pagination makes it easy to return results in chunks. You can paginate using:

```
| return ProductResource::collection(Product::paginate(10));
```

You can also allow filtering by query parameters such as `name` or `price` range. This makes your API more flexible and user-friendly for front-end integration.

14.9 Best Practices in API Development

To build a robust API, it is important to follow industry best practices. Always return consistent JSON responses, use appropriate HTTP status codes, and validate all input data. Consider versioning your APIs using a prefix like `/api/v1` to maintain backward compatibility. Also, rate limiting can help prevent abuse by controlling the number of requests from a single client.

Logging errors and monitoring API performance are also essential practices for a production-ready system.

14.10 Exercise 42: Building a “Hello World” REST API in Laravel 12

14.10.1 Description

In this hands-on lab, we will build a **simple “Hello World” REST API in Laravel 12** that supports **GET, POST, PUT, and DELETE** requests. Each request will return a **message** and the **current server date and time**.

We will also **activate API support** in Laravel 12 before implementing our API.

14.10.2 Objectives

By the end of this lab, you will:

- Learn how to **enable API support in Laravel 12**
- Create a **REST API with GET, POST, PUT, and DELETE endpoints**
- Return a **structured response with a message and timestamp**
- Test the API using **Postman or cURL**

14.10.3 Prerequisites

Before starting, ensure you have:

- **PHP 8.2+**, Composer, and MySQL installed
- **Laravel 12 installed**
- **Postman or cURL** for API testing

This lab uses PHP 8.4, Laravel 12 and Visual Studio Code as the primary editor.

If you want to test Api with Postman, you can download it from Postman
<https://www.postman.com/downloads/>.

14.10.4 Steps

Here are the steps to complete this lab:

14.10.4.1 Step 1: Create a New Laravel 12 Project

Open your terminal and create a new Laravel project:

```
| laravel new hello-api  
| cd hello-api  
| code .
```

You should see the Laravel project structure in your code editor.

14.10.4.2 Step 2: Enable API Support in Laravel 12

Laravel 12 **does not enable API routing by default**. Activate API support by running:

```
| php artisan install:api
```

This command:

- ✓ Creates the `routes/api.php` file
- ✓ Configures **middleware** for API authentication
- ✓ Installs Laravel Sanctum (optional for authentication)

14.10.4.3 Step 3: Create a Controller for API Logic

We will create a controller to handle the API logic. Run the following command to generate a controller:

```
| php artisan make:controller HelloController
```

Now, open `app/Http/Controllers/HelloController.php` and update it with the following code:

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
use Illuminate\Http\JsonResponse;  
  
class HelloController extends Controller  
{  
    // GET: Return a welcome message with timestamp  
    public function getMessage(): JsonResponse  
    {  
        return response()->json([  
            'message' => 'Hello World from GET!',  
            'timestamp' => now()->toDateTimeString()  
        ], 200);  
    }  
  
    // POST: Accept input and return a message  
    public function postMessage(Request $request): JsonResponse
```

```

{
    $name = $request->input('name', 'Guest');

    return response()->json([
        'message' => "Hello, $name! This is a POST request.",
        'timestamp' => now()->toDateTimeString()
    ], 201);
}

// PUT: Update a message
public function putMessage(Request $request): JsonResponse
{
    return response()->json([
        'message' => 'Hello World has been updated via PUT!',
        'timestamp' => now()->toDateTimeString()
    ], 200);
}

// DELETE: Delete a message
public function deleteMessage(): JsonResponse
{
    return response()->json([
        'message' => 'Hello World has been deleted!',
        'timestamp' => now()->toDateTimeString()
    ], 200);
}
}

```

14.10.4.4 Step 4: Define API Routes

Edit `routes/api.php` to define API endpoints for **GET, POST, PUT, and DELETE**:

```

use Illuminate\Support\Facades\Route;
use App\Http\Controllers\HelloController;

Route::get('/hello', [HelloController::class, 'getMessage']);
Route::post('/hello', [HelloController::class, 'postMessage']);
Route::put('/hello', [HelloController::class, 'putMessage']);
Route::delete('/hello', [HelloController::class, 'deleteMessage']);

```

Keep existing routes intact. This file is where you define all your API routes.

This creates the following endpoints:

Method	URL	Description
GET	/api/hello	Returns a welcome message
POST	/api/hello	Accepts input and returns a message
PUT	/api/hello	Updates a message
DELETE	/api/hello	Deletes a message

You can test these endpoints using Postman or cURL. The `HelloController` will handle the logic for each request.

14.10.4.5 Step 5: Test the API

After completing the above steps, start the Laravel development server:

```
| php artisan serve
```

Next, open Postman or your preferred API testing tool and test the endpoints.

1. Test with Postman

- **GET Request:** `http://127.0.0.1:8000/api/hello`
- **POST Request:** `http://127.0.0.1:8000/api/hello` with header `Content-Type: application/json` and JSON body:

```
{ "name": "Mekka" }
```
- **PUT Request:** `http://127.0.0.1:8000/api/hello`
- **DELETE Request:** `http://127.0.0.1:8000/api/hello`

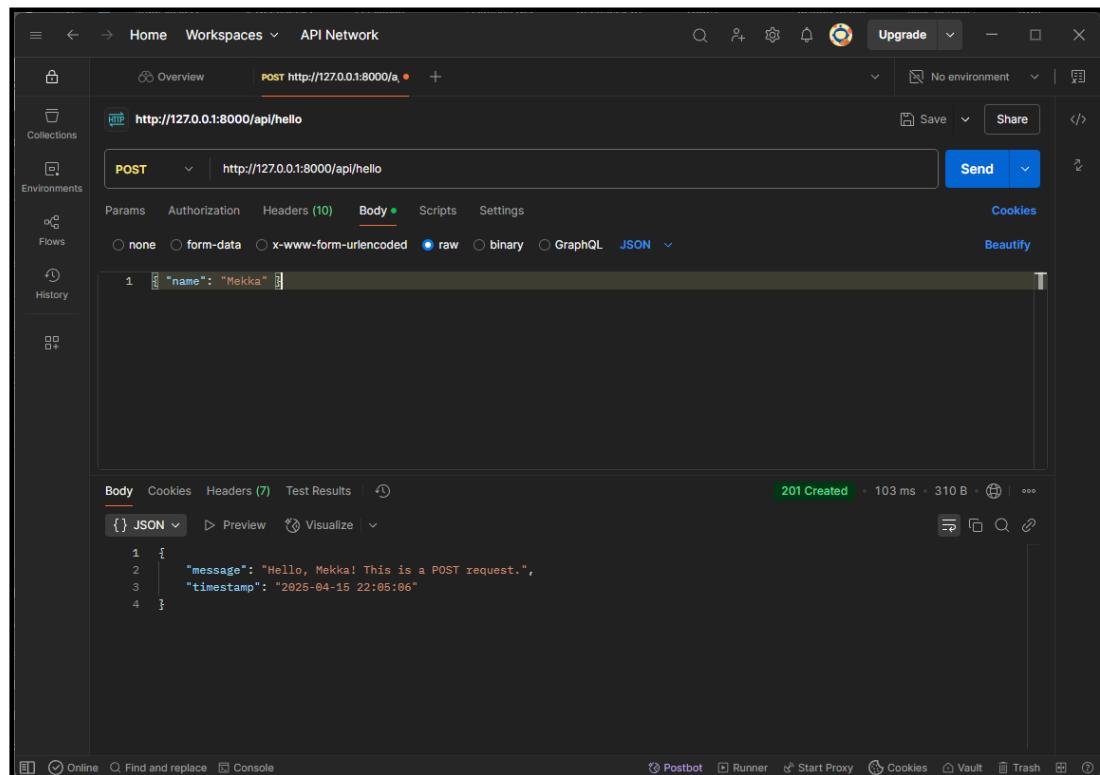


Figure 14.1: Postman API Testing.

2. Test with cURL

- **GET Request**

```
| curl -X GET http://127.0.0.1:8000/api/hello
```

- **POST Request**

```
| curl -X POST http://127.0.0.1:8000/api/hello \
-H "Content-Type: application/json" \
-d '{"name":"Mekka"}'
```

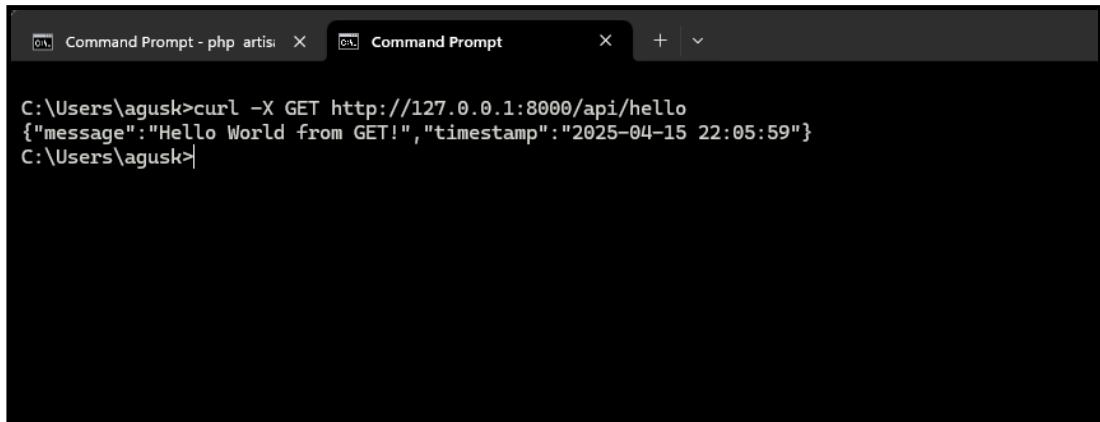
For Windows, change \ to ^ in the above command.

- **PUT Request**

```
| curl -X PUT http://127.0.0.1:8000/api/hello
```

- **DELETE Request**

```
| curl -X DELETE http://127.0.0.1:8000/api/hello
```



The screenshot shows a Windows Command Prompt window with two tabs open. The active tab displays the command `curl -X GET http://127.0.0.1:8000/api/hello` and its response: `{"message": "Hello World from GET!", "timestamp": "2025-04-15 22:05:59"}`. The other tab is titled "Command Prompt - php artisan".

Figure 14.2: cURL API Testing.

14.10.5 Summary

In this hands-on lab, you have successfully:

- Activated **API support** in Laravel 12
- Created a **REST API** with GET, POST, PUT, and DELETE methods
- Implemented a **Hello World API** that returns a message and timestamp
- Tested the API using **Postman and cURL**

This lab sets the foundation for REST API development in Laravel 12.

14.11 Exercise 43: Building a Calculator REST API in Laravel 12

14.11.1 Description

In this hands-on lab, we will create a simple **Calculator API** using **Laravel 12**. The API will allow users to perform basic arithmetic operations (addition, subtraction, multiplication, and division). We will implement a **Data Transfer Object (DTO)** to handle request validation and data structuring.

14.11.2 Objectives

By the end of this lab, you will:

- Understand how to create a **REST API in Laravel 12**
- Learn how to use **Controllers and Routes** in Laravel
- Implement a **Data Transfer Object (DTO)** for request validation
- Apply **Service Layer Architecture** to separate business logic
- Test the API using **Postman or cURL**

14.11.3 Prerequisites

Before starting, ensure you have:

- **PHP 8.2+**, Composer, and MySQL installed
- **Laravel 12 installed**
- **Postman or cURL** for API testing
- Basic knowledge of Laravel

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

14.11.4 Steps

Here are the steps to complete this lab:

14.11.4.1 Step 1: Create a New Laravel 12 Project

Open your terminal and create a new Laravel project:

```
| laravel new calculator-api  
| cd calculator-api  
| code .
```

You should see the Laravel project structure in your code editor.

14.11.4.2 Step 2: Enable API in Laravel 12

By default, **API routes are disabled** in Laravel 12. We must **enable API support** using the following command:

```
| php artisan install:api
```

This command:

- ✓ Creates the `routes/api.php` file
- ✓ Configures **middleware** for API authentication
- ✓ Installs Laravel Sanctum for **API security** (not used in this lab, but useful for authentication)

When you run this command, you will be asked to perform migration. Press `y` to continue.

14.11.4.3 Step 3: Create a Data Transfer Object (DTO)

Laravel doesn't provide built-in DTOs, so we will create a **custom DTO class**. Run the following command to create a **DTO class** inside the `app/DTOs` folder:

```
| mkdir app/DTOs  
| touch app/DTOs/CalculatorDTO.php
```

Now, edit the `app/DTOs/CalculatorDTO.php` file:

```
<?php  
  
namespace App\DTOs;  
  
class CalculatorDTO  
{  
    public float $num1;  
    public float $num2;  
    public string $operation;  
  
    public function __construct(float $num1, float $num2, string $operation)  
    {  
        $this->num1 = $num1;  
        $this->num2 = $num2;  
        $this->operation = $operation;  
    }  
}
```

14.11.4.4 Step 4: Create a Service Class for Business Logic

Run the following command to create a **CalculatorService** class:

```
| mkdir app/Services  
| touch app/Services/CalculatorService.php
```

Edit app/Services/CalculatorService.php:

```
<?php  
  
namespace App\Services;  
  
use App\DTOs\CalculatorDTO;  
  
class CalculatorService  
{  
    public function calculate(CalculatorDTO $dto): float  
    {  
        switch ($dto->operation) {  
            case 'add':  
                return $dto->num1 + $dto->num2;  
            case 'subtract':  
                return $dto->num1 - $dto->num2;  
            case 'multiply':  
                return $dto->num1 * $dto->num2;  
            case 'divide':  
                if ($dto->num2 == 0) {  
                    throw new \Exception("Cannot divide by zero");  
                }  
                return $dto->num1 / $dto->num2;  
            default:  
                throw new \Exception("Invalid operation");  
        }  
    }  
}
```

14.11.4.5 Step 5: Create the Calculator API Controller

Run the following command to create a **CalculatorController**:

```
| php artisan make:controller CalculatorController
```

Now, edit app/Http/Controllers/CalculatorController.php:

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
use App\DTOs\CalculatorDTO;  
use App\Services\CalculatorService;  
use Illuminate\Http\JsonResponse;  
  
class CalculatorController extends Controller  
{  
    private $calculatorService;  
  
    public function __construct(CalculatorService $calculatorService)  
    {  
        $this->calculatorService = $calculatorService;  
    }
```

```

public function calculate(Request $request): JsonResponse
{
    // Validate request
    $validated = $request->validate([
        'num1' => 'required|numeric',
        'num2' => 'required|numeric',
        'operation' => 'required|string|in:add,subtract,multiply,divide',
    ]);

    try {
        // Create DTO instance
        $dto = new CalculatorDTO($validated['num1'], $validated['num2'], $validated['operation']);

        // Process calculation
        $result = $this->calculatorService->calculate($dto);

        return response()->json([
            'success' => true,
            'num1' => $dto->num1,
            'num2' => $dto->num2,
            'operation' => $dto->operation,
            'result' => $result
        ], 200);
    } catch (\Exception $e) {
        return response()->json(['error' => $e->getMessage()], 400);
    }
}
}

```

This controller handles the API logic for the calculator. It validates the input, creates a DTO instance, and calls the `CalculatorService` to perform the calculation. If an error occurs (e.g., division by zero), it returns a JSON error response.

14.11.4.6 Step 6: Set Up the API Routes

Edit the `routes/api.php` file and define API routes:

```

use Illuminate\Support\Facades\Route;
use App\Http\Controllers\CalculatorController;

Route::post('/calculate', [CalculatorController::class, 'calculate']);

```

This creates a single endpoint for the calculator API:

14.11.4.7 Step 7: Test the API with Postman or cURL

After completing the above steps, start the Laravel development server:

```
| php artisan serve
```

Next, open Postman or your preferred API testing tool and test the endpoint.

Test with Postman

Send a `POST` request to `http://127.0.0.1:8000/api/calculate` with header `Content-Type: application/json` the following JSON payload:

```
{  
    "num1": 10,  
    "num2": 5,  
    "operation": "add"  
}
```

After sending the request, you should see a JSON response similar to this:

```
{  
    "success": true,  
    "num1": 10,  
    "num2": 5,  
    "operation": "add",  
    "result": 15  
}
```

Try to change the `operation` to `subtract`, `multiply`, or `divide` and observe the results.

Test Using cURL

Run this command in your terminal:

```
curl -X POST http://127.0.0.1:8000/api/calculate \  
-H "Content-Type: application/json" \  
-d '{"num1":10, "num2":5, "operation":"add"}'
```

For Windows, you may run with one line:

```
curl -X POST http://127.0.0.1:8000/api/calculate -H "Content-Type: application/json" -d "{\"num1":10, "num2":5, "operation":"add"}"
```

Try changing the `operation` to `subtract`, `multiply`, or `divide` and observe the results.

Try to perform a division by zero:

```
{  
    "num1": 10,  
    "num2": 0,  
    "operation": "divide"  
}
```

And observe the responses.

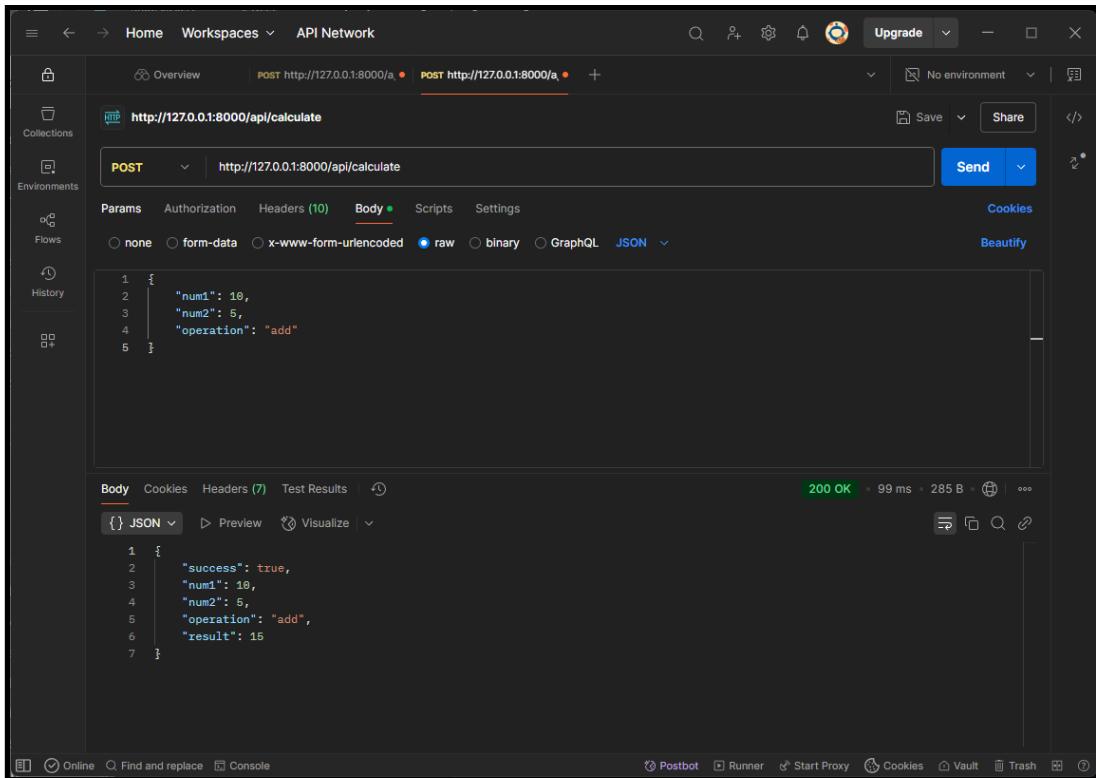


Figure 14.3: Error on handling divide.

14.11.5 Summary

In this hands-on lab, you have successfully:

- Created a **REST API** with Laravel 12
- Implemented a **Data Transfer Object (DTO)** for structured data handling
- Separated business logic using a **Service Layer**
- Validated API inputs and handled errors
- Tested the API using **Postman and cURL**

14.12 Exercise 44: Creating a CRUD Resource REST API

14.12.1 Description

In this hands-on lab, we will build a **CRUD (Create, Read, Update, Delete) REST API** for a **Product Management System** using **Laravel 12** and **MySQL**. This API will allow users to perform operations on product data, including listing, creating, updating, and deleting products.

We will follow **RESTful principles** and implement **Laravel API Resource Controllers** for clean and maintainable code.

14.12.2 Objectives

By the end of this lab, you will:

- Activate API support in Laravel 12
- Set up API routes using Laravel 12
- Use Laravel API Resource Controllers for CRUD operations
- Connect Laravel 12 to a MySQL database
- Implement request validation in Laravel
- Test the API using Postman or cURL

14.12.3 Prerequisites

Before starting, ensure you have:

- **PHP 8.2+**, Composer, and MySQL installed
- **Laravel 12 installed**
- **Postman or cURL** for API testing
- **A MySQL database already set up**

This lab uses PHP 8.4, Laravel 12 and Visual Studio Code as the primary editor.

Let's get started!

14.12.4 Steps

Here are the steps to complete this lab:

14.12.4.1 Step 1: Create Laravel Project

Open your terminal and create a new Laravel project:

```
| laravel new product-api  
| cd product-api  
| code .
```

You should see the Laravel project structure in your code editor.

14.12.4.2 Step 2: Configure the Database Connection

Open the `.env` file and configure the MySQL connection:

```
| DB_CONNECTION=mysql  
| DB_HOST=127.0.0.1  
| DB_PORT=3306  
| DB_DATABASE=productdb  
| DB_USERNAME=root  
| DB_PASSWORD=yourpassword
```

Change the **database name**, **username**, and **password** to match your MySQL setup.

Run the following command to apply the database configuration:

```
| php artisan config:clear
```

14.12.4.3 Step 3: Enable API in Laravel 12

By default, **API routes are disabled** in Laravel 12. We must **enable API support** using the following command:

```
| php artisan install:api
```

This command:

- ✓ Creates the `routes/api.php` file
- ✓ Configures **middleware** for API authentication
- ✓ Installs Laravel Sanctum for **API security** (not used in this lab, but useful for authentication)

When you run this command, you will be asked to perform migration. Press `Y` to continue.

14.12.4.4 Step 4: Create a Product Model and Migration

Run the following command to generate a **Product model with a migration file**:

```
| php artisan make:model Product -m
```

Edit the **migration file** in `database/migrations/YYYY_MM_DD_create_products_table.php` to define the **products table schema**:

```
| public function up()  
{  
    Schema::create('products', function (Blueprint $table) {  
        $table->id();  
        $table->string('name');  
        $table->text('description')->nullable();  
        $table->decimal('price', 10, 2);  
        $table->integer('stock');  
        $table->timestamps();  
    });  
}
```

Modify Product model in `app/Models/Product.php` to allow mass assignment:

```
class Product extends Model
{
    protected $fillable = [
        'name',
        'description',
        'price',
        'stock',
    ];

    protected $casts = [
        'price' => 'decimal:2',
        'stock' => 'integer',
    ];
}
```

Save the changes. This model represents the `products` table in the database and allows mass assignment for the specified fields.

Run the migration to create the table in MySQL:

```
| php artisan migrate
```

You should see a message indicating that the migration was successful. This creates a `products` table with the specified columns in your MySQL database.

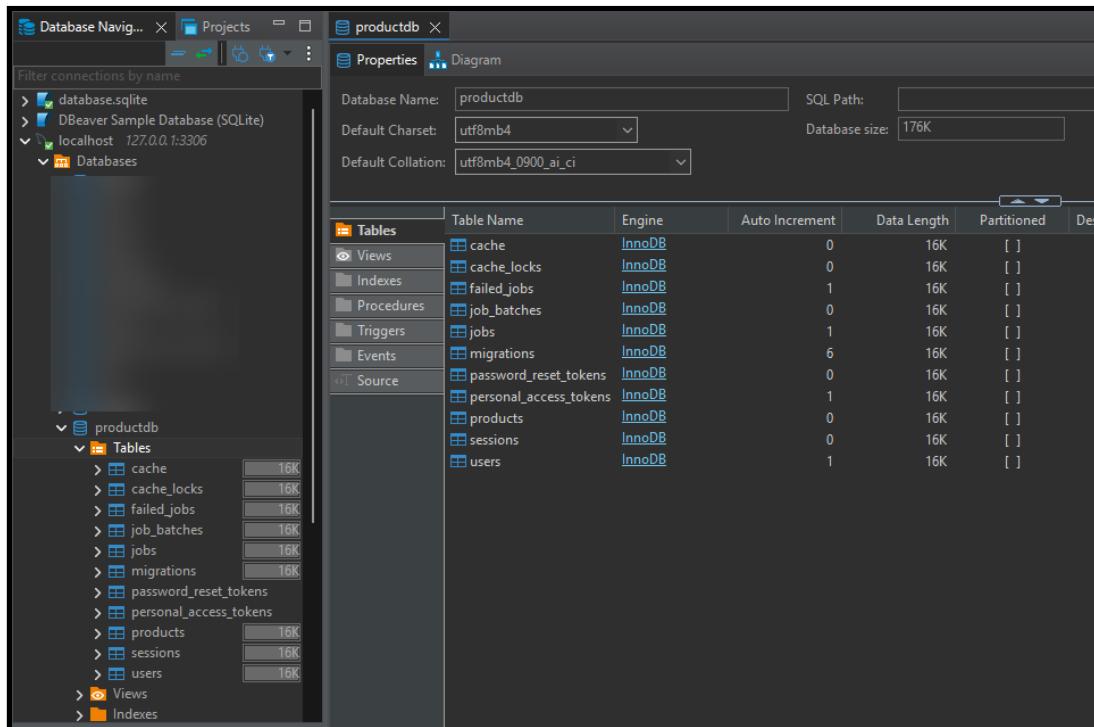


Figure 14.6: Migration successful.

14.12.4.5 Step 5: Create a Resource Controller

Run the following command to create a **resource controller**:

```
| php artisan make:controller ProductController --api
```

This creates `app/Http/Controllers/ProductController.php` with **predefined CRUD methods**.

14.12.4.6 Step 6: Implement CRUD Methods in ProductController

Edit `app/Http/Controllers/ProductController.php` and update the methods:

```
<?php

namespace App\Http\Controllers;

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

class ProductController extends Controller
{
    // Get all products
    public function index(): JsonResponse
    {
        return response()->json(Product::all(), 200);
    }

    // Create a new product
    public function store(Request $request): JsonResponse
    {
        $validated = $request->validate([
            'name' => 'required|string|max:255',
            'description' => 'nullable|string',
            'price' => 'required|numeric|min:0',
            'stock' => 'required|integer|min:0',
        ]);

        $product = Product::create($validated);

        return response()->json($product, 201);
    }

    // Get a specific product
    public function show($id): JsonResponse
    {
        $product = Product::find($id);

        if (!$product) {
            return response()->json(['error' => 'Product not found'], 404);
        }

        return response()->json($product, 200);
    }

    // Update a product
    public function update(Request $request, $id): JsonResponse
    {
        $product = Product::find($id);
```

```

    if (!$product) {
        return response()->json(['error' => 'Product not found'], 404);
    }

    $validated = $request->validate([
        'name' => 'string|max:255',
        'description' => 'nullable|string',
        'price' => 'numeric|min:0',
        'stock' => 'integer|min:0',
    ]);

    $product->update($validated);

    return response()->json($product, 200);
}

// Delete a product
public function destroy($id): JsonResponse
{
    $product = Product::find($id);

    if (!$product) {
        return response()->json(['error' => 'Product not found'], 404);
    }

    $product->delete();

    return response()->json(['message' => 'Product deleted successfully'], 200);
}

```

This controller handles all CRUD operations for the `Product` model. Each method validates input data and returns appropriate JSON responses.

14.12.4.7 Step 7: Define API Routes

Edit `routes/api.php` to define API endpoints:

```

use Illuminate\Support\Facades\Route;
use App\Http\Controllers\ProductController;

Route::apiResource('products', ProductController::class);

```

This automatically sets up the following **REST API endpoints**:

Method	URL	Description
GET	/api/products	Get all products
POST	/api/products	Create a new product
GET	/api/products/{id}	Get a specific product

Method	URL	Description
PUT/PATCH	/api/products/{id}	Update a product
DELETE	/api/products/{id}	Delete a product

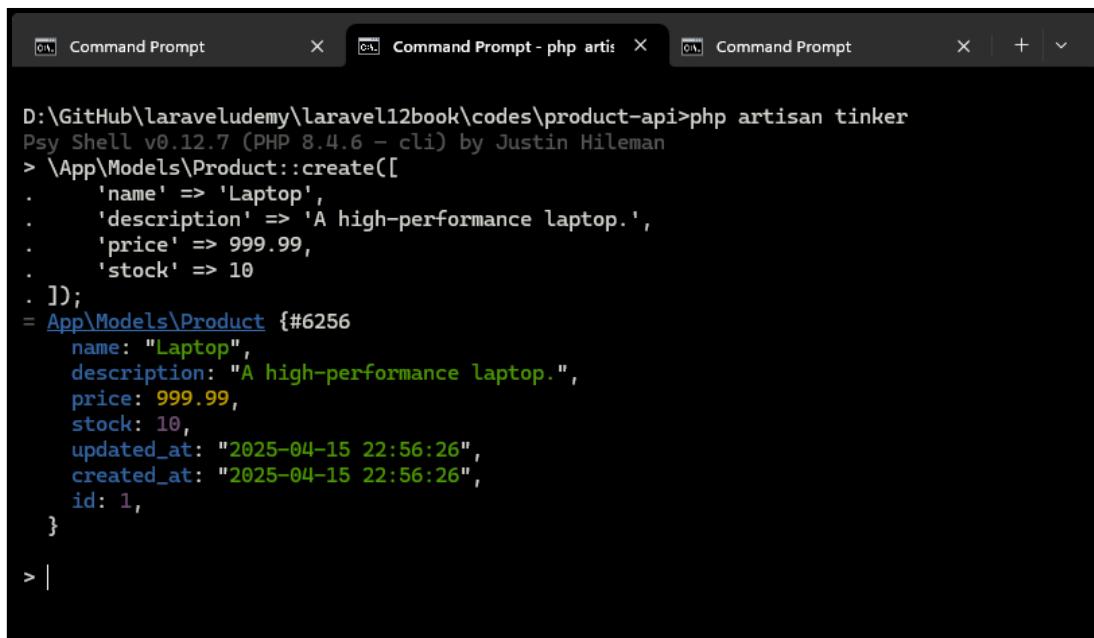
14.12.4.8 Step 8: Seed Sample Data (Optional)

To quickly test the API, you can generate sample data. Run:

```
| php artisan tinker
```

Then enter the following command to create a sample product:

```
\App\Models\Product::create([
    'name' => 'Laptop',
    'description' => 'A high-performance laptop.',
    'price' => 999.99,
    'stock' => 10
]);
```



The screenshot shows three terminal windows side-by-side. The left window is titled 'Command Prompt' and contains the command 'php artisan tinker'. The middle window is titled 'Command Prompt - php artisan tinker' and shows the execution of the code provided above, which creates a new Product instance with attributes: name ('Laptop'), description ('A high-performance laptop.'), price (999.99), and stock (10). The right window is also titled 'Command Prompt' and shows the resulting output: a new `App\Models\Product` object (#6256) with the same attributes, plus additional database timestamps for `updated_at` and `created_at`.

```
D:\GitHub\laraveludemy\laravel12book\codes\product-api>php artisan tinker
Psy Shell v0.12.7 (PHP 8.4.6 - cli) by Justin Hileman
> \App\Models\Product::create([
>     'name' => 'Laptop',
>     'description' => 'A high-performance laptop.',
>     'price' => 999.99,
>     'stock' => 10
> ]);
= App\Models\Product {#6256
    name: "Laptop",
    description: "A high-performance laptop.",
    price: 999.99,
    stock: 10,
    updated_at: "2025-04-15 22:56:26",
    created_at: "2025-04-15 22:56:26",
    id: 1,
}
> |
```

Figure 14.7: Sample product created.

To exit Tinker, type `exit` and press Enter. This creates a sample product in the database.

14.12.4.9 Step 9: Test the API

After completing the above steps, start the Laravel development server:

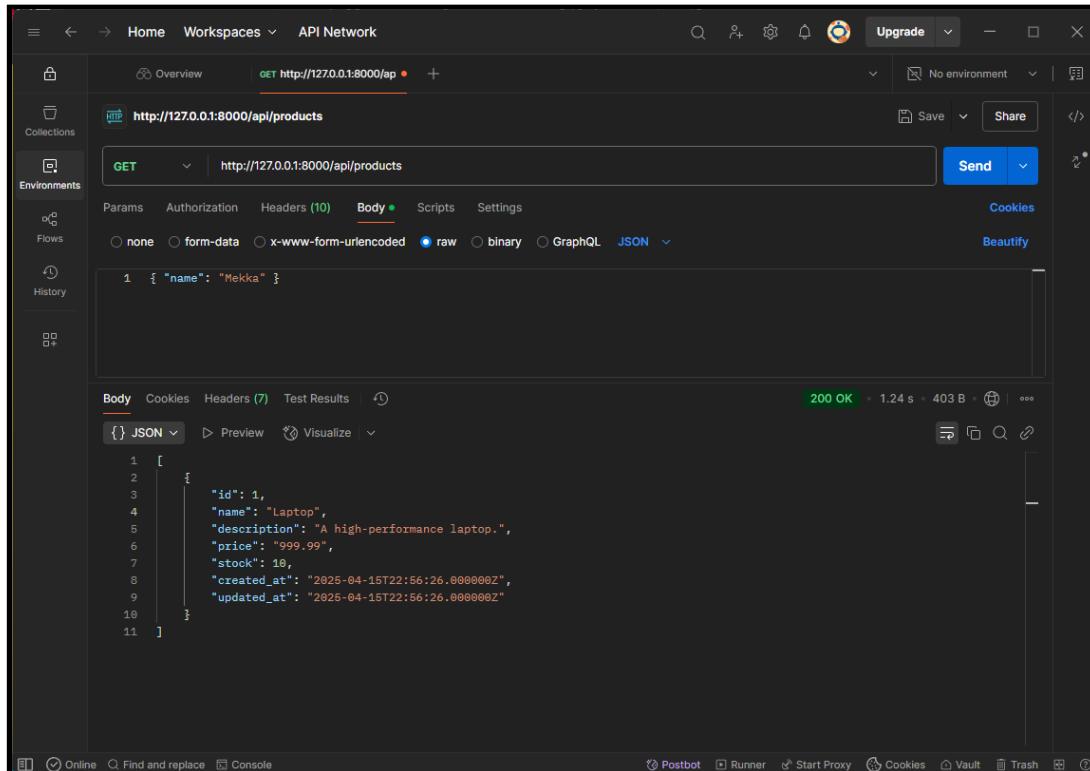
```
| php artisan serve
```

Next, open Postman or your preferred API testing tool and test the endpoints.

Test with Postman

Send a `GET` request to:

```
| http://127.0.0.1:8000/api/products
```



The screenshot shows the Postman application interface. In the top navigation bar, the URL `http://127.0.0.1:8000/api/products` is entered under the 'Overview' tab. Below the URL, there is a 'Send' button. The 'Body' tab is selected, showing a raw JSON payload:

```
1 { "name": "Mekka" }
```

Under the 'Body' tab, the response is displayed as a JSON array:

```
1 [ 2 { 3 "id": 1, 4 "name": "Laptop", 5 "description": "A high-performance laptop.", 6 "price": "999.99", 7 "stock": 10, 8 "created_at": "2025-04-15T22:56:26.000000Z", 9 "updated_at": "2025-04-15T22:56:26.000000Z" 10 } 11 ]
```

The status bar at the bottom indicates a `200 OK` response with a time of `1.24 s` and a size of `403 B`.

Figure 14.8: Postman to get all products.

You should see a JSON response with the list of products. If you want to get details of a specific product, for instance product id `1`, send a `GET` request to:

```
| http://127.0.0.1:8000/api/products/1
```

You should see a JSON response with the product details.

Try other methods like `POST`, `PUT`, and `DELETE` to test the CRUD operations.

For example, to create a product, you perform this test:

- Set method to `POST`
- Set Url `http://127.0.0.1:8000/api/products`

- Set header Content-Type: application/json
- Set the body to raw and use the following JSON:

```
{
  "name": "Phone",
  "description": "Smartphone",
  "price": 500,
  "stock": 50
}
```

- Click Send to create the product.
- You should see a JSON response with the created product details.

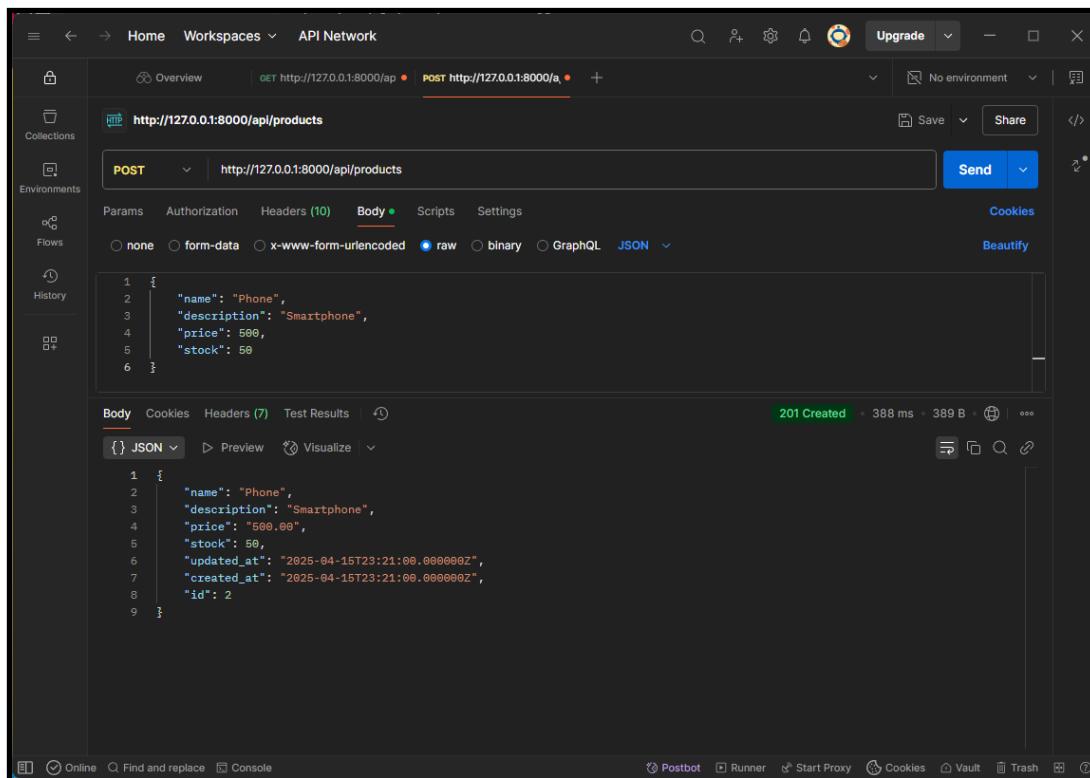


Figure 14.9: Postman to create a product.

For updating a product, for instance, we want to update the product with ID 1. You can perform this test:

- Set method to PUT
- Set Url <http://127.0.0.1:8000/api/products/1>
- Set header Content-Type: application/json
- Set the body to raw and use the following JSON:

```
| {  
|     "id": 1,  
|     "price": 1200,  
|     "stock": 5  
| }
```

- Click `Send` to update the product.
- You should see a JSON response with the updated product details.

For deleting a product, for instance, we want to delete the product with ID `1`. You can perform this test:

- Set method to `DELETE`
- Set Url `http://127.0.0.1:8000/api/products/1`
- Click `Send` to delete the product.
- You should see a JSON response confirming the deletion.

Test with cURL

We also test the API using `cURL` commands. Here are some examples:

Create a product:

```
| curl -X POST http://127.0.0.1:8000/api/products \  
|   -H "Content-Type: application/json" \  
|   -d '{"name":"Phone", "description":"Smartphone", "price":500, "stock":50}'
```

Get all products:

```
| curl -X GET http://127.0.0.1:8000/api/products
```

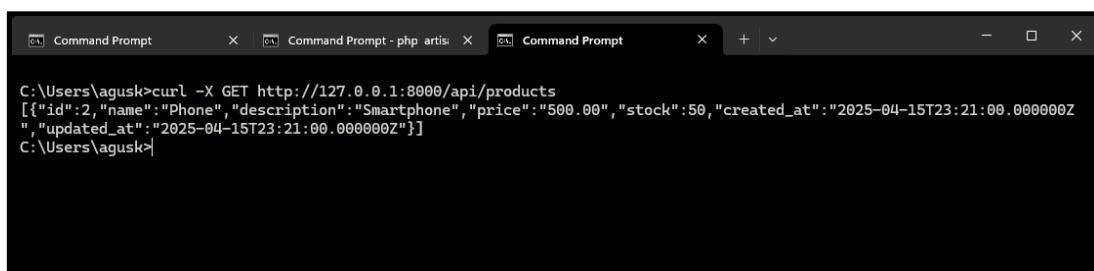


Figure 14.10: cURL to get all products.

Update a product:

```
| curl -X PUT http://127.0.0.1:8000/api/products/1 \  
|   -H "Content-Type: application/json" \  
|   -d '{"id": 1, "price":1200, "stock":5}'
```

Delete a product:

```
| curl -X DELETE http://127.0.0.1:8000/api/products/1
```

14.12.5 Summary

In this hands-on lab, you have successfully:

- Activated **API support** in Laravel 12
- Connected Laravel 12 to a **MySQL database**
- Created a **Product API** with CRUD operations
- Used **API Resource Controllers** for clean code
- Implemented **Request Validation**
- Tested the API using **Postman and cURL**

14.13 Exercise 45: Upload File via REST API with Form Data in Laravel 12

14.13.1 Description

In this lab, we will build a REST API using Laravel 12 that allows clients to upload a file (e.g., a photo or document) along with additional user information such as name, email, and phone number. This API accepts multipart form data (form-data header), stores the uploaded file in Laravel's local storage, and records the metadata in the database.

14.13.2 Objectives

- Understand how to activate and configure REST API in Laravel 12.
- Build an API endpoint to accept file uploads and associated user data.
- Use multipart form-data headers in requests.
- Store file uploads in Laravel's `storage/app/public` directory.
- Save metadata (name, email, phone, file path) in the database.

14.13.3 Prerequisites

Before starting this lab, ensure you have the following installed and configured:

- PHP 8.2 or higher
- Composer
- Laravel 12
- A local or remote database (MySQL or SQLite recommended)
- Visual Studio Code or any code editor
- Postman or Thunder Client (for testing API with form-data)

- Git (optional)

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

14.13.4 Steps

Here are the steps to complete this lab:

14.13.4.1 Step 1: Create a New Laravel 12 Project

Open your terminal and run:

```
| laravel new file-upload-api  
| cd file-upload-api  
| code .
```

Accept the default options when prompted. This will create a new Laravel project named `file-upload-api` with SQLite database and open it in Visual Studio Code.

You should see the Laravel project structure in your code editor.

14.13.4.2 Step 2: Configure Database

Since we are using default SQLite database, we can verify to configure open the `.env` file.

```
| DB_CONNECTION=sqlite
```

14.13.4.3 Step 3: Enable API Support in Laravel 12

Laravel 12 **does not enable API routing by default**. Activate API support by running:

```
| php artisan install:api
```

This command:

- ✓ Creates the `routes/api.php` file
- ✓ Configures **middleware** for API authentication
- ✓ Installs Laravel Sanctum for **API security**

When you run the command, you may asked to perform migrations. You can skip it for now. Type `no` and press `Enter`. We will run the migrations later.

14.13.4.4 Step 4: Create the Model and Migration

We will create a model named `Submission` to handle the file uploads and user data. Run the following command:

Generate a model with migration:

```
| php artisan make:model Submission -m
```

Open the migration file under `database/migrations/xxxx_xx_xx_create_submissions_table.php` and modify the schema:

```
| Schema::create('submissions', function (Blueprint $table) {
|     $table->id();
|     $table->string('name');
|     $table->string('email');
|     $table->string('phone');
|     $table->string('file_path');
|     $table->timestamps();
| });
| }
```

We also modify the `Submission` model in `app/Models/Submission.php` to allow mass assignment:

```
| namespace App\Models;
| use Illuminate\Database\Eloquent\Model;
|
| class Submission extends Model
| {
|
|     protected $fillable = [
|         'name',
|         'email',
|         'phone',
|         'file_path',
|     ];
| }
```

This model represents the `submissions` table in the database and allows mass assignment for the specified fields.

Run the migration:

```
| php artisan migrate
```

You should see a message indicating that the migration was successful. This creates a `submissions` table with the specified columns in your SQLite database.

14.13.4.5 Step 5: Create a Controller for the API

We will create a controller to handle the API logic. Run the following command to generate a controller:

```
| php artisan make:controller API/SubmissionController
```

Edit `app/Http/Controllers/API/SubmissionController.php`:

```
namespace App\Http\Controllers\API;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\Submission;
use Illuminate\Support\Facades\Storage;

class SubmissionController extends Controller
{
    public function store(Request $request)
    {
        $validated = $request->validate([
            'name' => 'required|string',
            'email' => 'required|email',
            'phone' => 'required|string',
            'file' => 'required|file|max:5120', // max 5MB
        ]);

        // Store the file
        $filePath = $request->file('file')->store('uploads', 'public');

        // Save the submission
        $submission = Submission::create([
            'name' => $validated['name'],
            'email' => $validated['email'],
            'phone' => $validated['phone'],
            'file_path' => $filePath,
        ]);

        return response()->json([
            'message' => 'File uploaded successfully.',
            'data' => $submission
        ], 201);
    }
}
```

This controller handles the file upload and saves the metadata in the database. It validates the input, stores the file in the `storage/app/public/uploads` directory, and returns a JSON response with the submission data.

14.13.4.6 Step 6: Register Routes in `api.php`

Open `routes/api.php` and add the route:

```
use App\Http\Controllers\API\SubmissionController;

Route::post('/submit', [SubmissionController::class, 'store']);
```

14.13.4.7 Step 7: Configure File Storage

To allow access to uploaded files, run:

```
| php artisan storage:link
```

Ensure `FILESYSTEM_DISK=public` is set in `.env`.

14.13.4.8 Step 8: Test the API

After completing the above steps, start the Laravel development server:

```
| php artisan serve
```

You can test the API using Postman or Thunder Client.

Open Postman or Thunder Client:

- **Method:** `POST`
- **URL:** `http://localhost:8000/api/submit`
- **Headers:** `Content-Type: multipart/form-data`
- **Body:** `form-data`
 - `name` (type: text)
 - `email` (type: text)
 - `phone` (type: text)
 - `file` (type: file)

Here is an example of how to set up the form-data in Postman:

The screenshot shows the Postman application interface. On the left, there's a sidebar with 'Collections', 'Environments', 'Flows', and 'History'. The main area shows a 'POST' request to 'http://localhost:8000/api/submit'. The 'Body' tab is selected, showing 'form-data' selected as the type. A table lists four fields: 'name' (Text, value: 'mr. a'), 'email' (Text, value: 'a@ilmudata.id'), 'phone' (Text, value: '12345678'), and 'file' (File, value: 'img1.png'). Below the table, there's a 'Response' section and a 'History' dropdown.

Figure 14.11: Postman form-data setup.

Click `Send` to submit the form. Make sure to select a file to upload.

You should get a JSON response like:

```
{  
    "message": "File uploaded successfully.",  
    "data": {  
        "id": 1,  
        "name": "mr.a",  
        "email": "a@ilmudata.id",  
        "phone": "12345678",  
        "file_path": "uploads/photo.jpg",  
        "created_at": "...",  
        "updated_at": "..."  
    }  
}
```

The screenshot shows the Postman interface for a POST request to `http://localhost:8000/api/submit`. The 'Body' tab is selected, showing the form-data configuration. The 'file' field is selected and contains the value `img1.png`. The 'Headers' tab shows the response with status `201 Created`, time `139 ms`, and size `494 B`. The 'Body' tab displays the JSON response from the server, which includes a message and a data object containing user information and a file path.

Key	Value	Description	Bulk Edit
name	mr. a		
email	a@ilmudata.id		
phone	12345678		
file	img1.png		

```
{  
    "message": "File uploaded successfully.",  
    "data": {  
        "name": "mr. a",  
        "email": "a@ilmudata.id",  
        "phone": "12345678",  
        "file_path": "uploads/RVpSxgVf5BXvcmy0WdmZTCy0iTc5E8xCKe3mSsJU.png",  
        "updated_at": "2025-04-16T00:31:06.000000Z",  
        "created_at": "2025-04-16T00:31:06.000000Z",  
        "id": 1  
    }  
}
```

Figure 14.12: Postman response.

14.13.5 Summary

In this hands-on lab, we built a REST API in Laravel 12 to upload files along with additional user data using a form-data header. We started by creating a new Laravel project, configured the database, defined the API route and controller, handled file storage, and validated input. This type of API is essential in many real-world applications such as registration systems, contact forms, or document uploads.

14.14 Exercise 46: Authentication & Authorization REST API

14.14.1 Description

In this hands-on lab, we will build an **authentication and authorization system** for a REST API using **Laravel 12**, **Sanctum**, and **MySQL**. We use JWT-like tokens for authentication and protect certain API routes.

The API will support:

- **User Registration** (`/api/register`)
- **User Login** (`/api/login`)
- **JWT-based Authentication** with expiration time
- **User Profile Retrieval** (`/api/profile`, protected by JWT token)

We will use **bcrypt** to hash passwords and **Laravel Sanctum** to generate **JWT-like API tokens**.

14.14.2 Objectives

By the end of this lab, you will:

- Activate **API support** in Laravel 12
- Implement **user registration with bcrypt password hashing**
- Implement **JWT authentication using Laravel Sanctum**
- Protect **profile API using JWT token**
- Test the API using **Postman or cURL**

14.14.3 Prerequisites

Before starting, ensure you have:

- **PHP 8.2+**, Composer, and MySQL installed
- **Laravel 12 installed**
- **Postman or cURL** for API testing

This lab uses PHP 8.4, Laravel 12 and Visual Studio Code as the primary editor.

14.14.4 Steps

Here are the steps to complete this lab:

14.14.4.1 Step 1: Create a New Laravel 12 Project

Open your terminal and create a new Laravel project:

```
| laravel new auth-api  
| cd auth-api  
| code .
```

You should see the Laravel project structure in your code editor.

14.14.4.2 Step 2: Configure the Database Connection

Open the `.env` file and configure the MySQL connection:

```
| DB_CONNECTION=mysql  
| DB_HOST=127.0.0.1  
| DB_PORT=3306  
| DB_DATABASE=authdb  
| DB_USERNAME=root  
| DB_PASSWORD=yourpassword
```

Change the **database name**, **username**, and **password** as needed.

Run the following command to apply the database configuration:

```
| php artisan config:clear
```

14.14.4.3 Step 3: Enable API Support in Laravel 12

Laravel 12 **does not enable API routing by default**. Activate API support by running:

```
| php artisan install:api
```

This command:

- ✓ Creates the `routes/api.php` file
- ✓ Configures **middleware** for API authentication
- ✓ Installs Laravel Sanctum for **API security**

When you run the command, you may be asked to perform migrations. You can skip it for now. Type `no` and press `Enter`. We will run the migrations later.

14.14.4.4 Step 4: Install Laravel Sanctum

After enabling API support, Laravel 12 installs **Sanctum** for API security. You can check the `composer.json` file to confirm.

If Sanctum is not installed, you can run the following command to install it:

```
| composer require laravel/sanctum
```

This will configure Sanctum for API authentication. You can check the `config/sanctum.php` file to confirm the installation.

14.14.4.5 Step 5: Create a User Model with Sanctum Token Ability

Make sure we have `HasApiTokens` trait in the `User` model. Edit `app/Models/User.php` to use Sanctum:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;

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

    protected $hidden = [
        'password',
        'remember_token',
    ];
}
```

14.14.4.6 Step 6: Create Authentication Controller

Run the following command to generate a controller:

```
| php artisan make:controller AuthController
```

Now, open `app/Http/Controllers/AuthController.php` and update it with the following code:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
```

```

use Illuminate\Http\JsonResponse;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Hash;
use App\Models\User;

class AuthController extends Controller
{
    // Register a new user
    public function register(Request $request): JsonResponse
    {
        $validated = $request->validate([
            'name' => 'required|string|max:255',
            'email' => 'required|email|unique:users',
            'password' => 'required|string|min:6'
        ]);

        $validated['password'] = Hash::make($validated['password']);

        $user = User::create($validated);

        return response()->json([
            'message' => 'User registered successfully',
            'user' => $user
        ], 201);
    }

    // Login and return a JWT token
    public function login(Request $request): JsonResponse
    {
        $validated = $request->validate([
            'email' => 'required|email',
            'password' => 'required|string'
        ]);

        $user = User::where('email', $validated['email'])->first();

        if (!$user || !Hash::check($validated['password'], $user->password)) {
            return response()->json(['error' => 'Invalid credentials'], 401);
        }

        $token = $user->createToken('authToken')->plainTextToken;

        return response()->json([
            'message' => 'Login successful',
            'token' => $token,
            'expires_in' => now()->addHours(24)->toDateTimeString()
        ], 200);
    }

    // Get the authenticated user's profile
    public function profile(Request $request): JsonResponse
    {
        return response()->json([
            'user' => $request->user()
        ], 200);
    }

    // Logout and revoke the token
    public function logout(Request $request): JsonResponse
    {
        $request->user()->tokens()->delete();
        return response()->json(['message' => 'Logged out successfully'], 200);
    }
}

```

```
| } }
```

14.14.4.7 Step 7: Define API Routes

Edit `routes/api.php` to define authentication routes:

```
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\AuthController;

Route::post('/register', [AuthController::class, 'register']);
Route::post('/login', [AuthController::class, 'login']);

Route::middleware('auth:sanctum')->group(function () {
    Route::get('/profile', [AuthController::class, 'profile']);
    Route::post('/logout', [AuthController::class, 'logout']);
});
```

This creates the following API endpoints:

Method	URL	Description
POST	/api/register	Register a new user
POST	/api/login	Login and get a JWT token
GET	/api/profile	Get the authenticated user's profile (protected)
POST	/api/logout	Logout and revoke the token (protected)

Save the changes. This sets up the authentication routes for user registration, login, profile retrieval, and logout.

14.14.4.8 Step 8: Test the API

After completing the above steps, start the Laravel development server:

```
| php artisan serve
```

In this step, you can test the API using **Postman** or **cURL**.

Register a New User

If you have cURL installed, you can run the following command to register a new user:

You can run this command in your terminal:

```
curl -X POST http://127.0.0.1:8000/api/register \
-H "Content-Type: application/json" \
-d '{"name":"Sheila", "email":"sheila@ilmudata.id", "password":"password12345"}'
```

For Windows users, you can use **Postman** to send the same request.

- **Method:** POST
- **URL:** <http://127.0.0.1:8000/api/register>
- **Headers:** Content-Type: application/json
- **Body:**

```
{
  "name": "Sheila",
  "email": "sheila@ilmudata.id",
  "password": "password12345"
}
```

- Click **Send** button to send the request.

You should see a response from the server.

The screenshot shows the Postman application interface. In the top navigation bar, the URL is set to `http://127.0.0.1:8000/api/register`. The 'Body' tab is selected, showing the JSON payload for user registration. The response at the bottom indicates a `201 Created` status with a response time of `2.81 s` and a size of `414 B`. The response body is displayed as JSON, showing the registered user's details including their ID, name, email, and timestamps for creation and update.

```
201 Created
2.81 s
414 B
{
  "message": "User registered successfully",
  "user": {
    "name": "Sheila",
    "email": "sheila@ilmudata.id",
    "updated_at": "2025-04-16T03:39:08.00000Z",
    "created_at": "2025-04-16T03:39:08.00000Z",
    "id": 1
  }
}
```

Figure 14.13: Register a new user.

Login and Get Token

You can run this command in your terminal:

```
| curl -X POST http://127.0.0.1:8000/api/login \
|   -H "Content-Type: application/json" \
|   -d '{"email":"sheila@ilmudata.id", "password":"password12345"}'
```

Here is an example of how to set up the login request in Postman:

- **Method:** POST
- **URL:** <http://127.0.0.1:8000/api/login>
- **Headers:** Content-Type: application/json
- **Body:**

```
| {
|   "email": "sheila@ilmudata.id",
|   "password": "password12345"
| }
```

- Click **Send** button to send the request.

You will get response from server like this.

```
| {
|   "message": "Login successful",
|   "token": "1|hXwX...",
|   "expires_in": "2025-04-17 03:43:50"
| }
```

You will receive a token in the response. This token is used for authentication in subsequent requests.

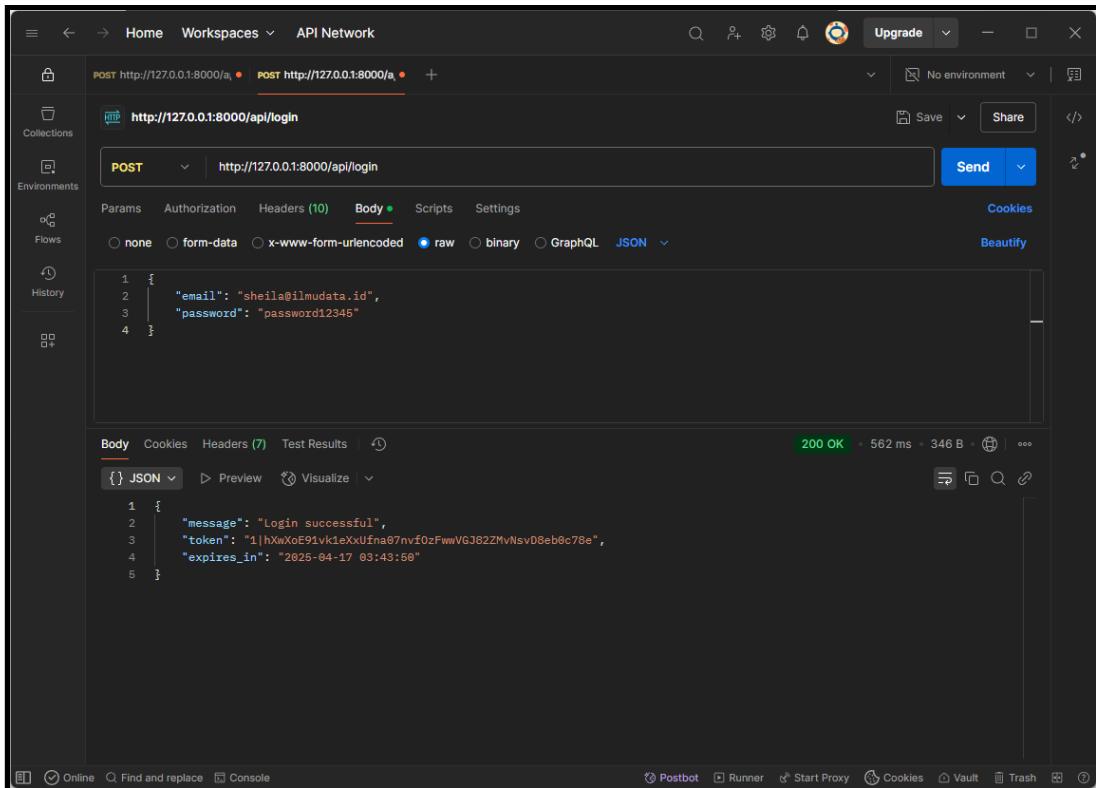


Figure 14.14: Getting token after login.

Access Profile API (Protected)

To access the profile API, you need to include the token in the request header. You can run this command in your terminal:

```
| curl -X GET http://127.0.0.1:8000/api/profile \
      -H "Authorization: Bearer YOUR_TOKEN_HERE"
```

Change `YOUR_TOKEN_HERE` with the token you received from the login response.

Here is an example of how to set up the profile request in Postman:

- **Method:** GET
- **URL:** `http://127.0.0.1:8000/api/profile`
- **Headers:** Content-Type: application/json
- **Headers:** Accept: application/json
- **Headers:** Authorization: Bearer YOUR_TOKEN_HERE
- Click **Send** button to send the request.

You will receive a response from server about user profile.

The screenshot shows the Postman application interface. On the left, there's a sidebar with sections for Collections, Environments, Flows, and History. The main area shows a request to `http://127.0.0.1:8000/api/profile` using the `GET` method. The `Headers` tab is selected, displaying the following configuration:

Key	Value
<input checked="" type="checkbox"/> Content-Type	application/json
<input checked="" type="checkbox"/> Authorization	Bearer 1 hXwXoE91vk1eXxUfna07nvfOzFvvVGJ82ZMvNsvD8eb0c78e
<input checked="" type="checkbox"/> Accept	application/json
Key	Description

Below the headers, the `Body` tab is selected, showing the JSON response:

```
1 {  
2   "user": {  
3     "id": 1,  
4     "name": "Sheila",  
5     "email": "sheila@ilmudata.id",  
6     "email_verified_at": null,  
7     "created_at": "2025-04-16T03:39:08.000000Z",  
8     "updated_at": "2025-04-16T03:39:08.000000Z"  
9   }  
10 }
```

The test results show a `200 OK` status with a response time of `161 ms` and a size of `393 B`. At the bottom, there are various navigation and utility buttons.

Figure 14.15: Accessing a profile with token.

Try to access the profile API without the token, and you should receive a `401 Unauthorized` error.

```
{  
  "message": "Unauthenticated."  
}
```

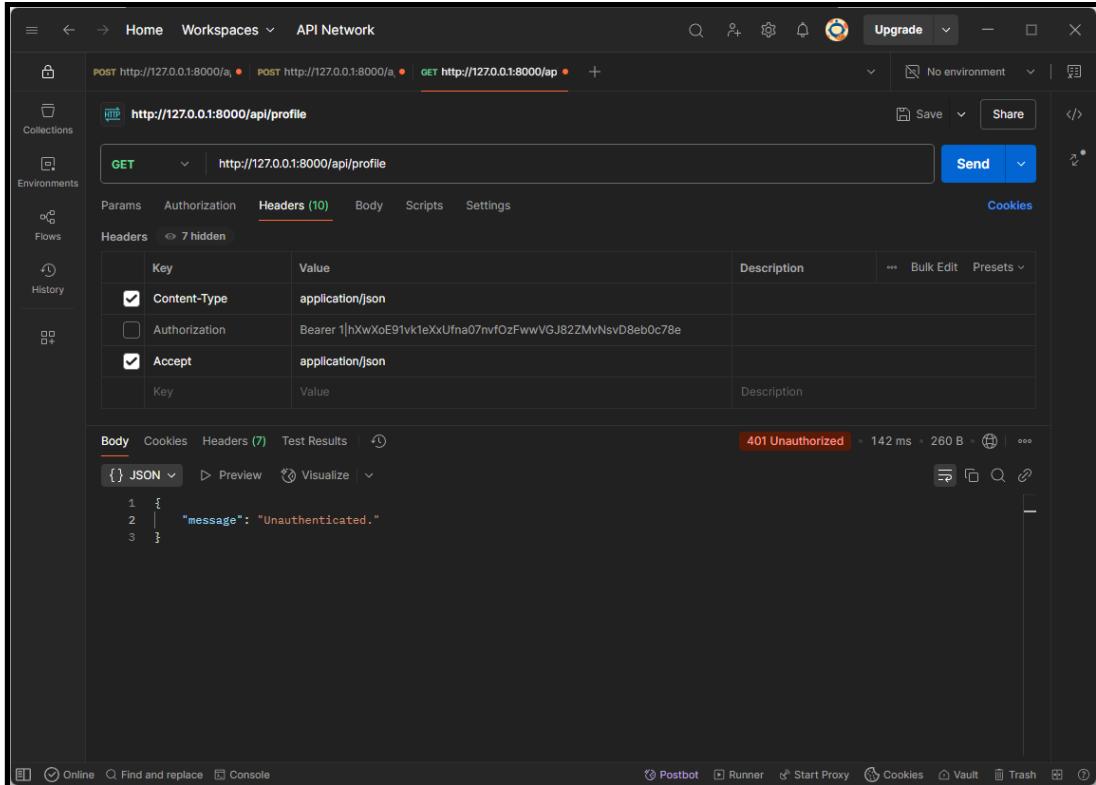


Figure 14.16: Getting unauthorized when sending without token.

Logout and Revoke Token

To logout and revoke the token, you can run this command in your terminal:

```
curl -X POST http://127.0.0.1:8000/api/logout \
-H "Authorization: Bearer YOUR_TOKEN_HERE"
```

Here is an example of how to set up the logout request in Postman:

- **Method:** POST
- **URL:** `http://127.0.0.1:8000/api/logout`
- **Headers:** Content-Type: application/json
- **Headers:** Accept: application/json
- **Headers:** Authorization: aBearer YOUR_TOKEN_HERE
- Click **Send** button to send the request.

You will receive a response from server like this.

```
{
  "message": "Logged out successfully"
}
```

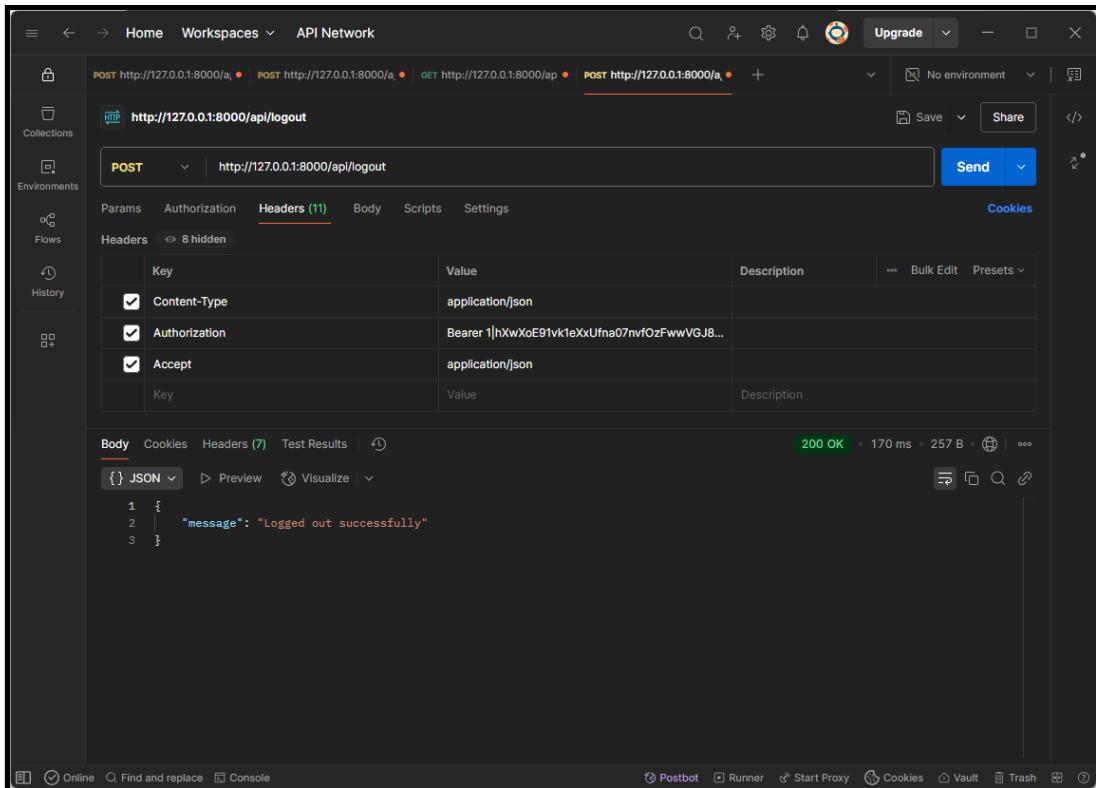


Figure 14.17: Performing logout.

14.14.5 Summary

In this hands-on lab, you have successfully:

- Activated **API support** in Laravel 12
- Implemented **user registration and login** with **bcrypt password hashing**
- Secured the API with **Laravel Sanctum JWT authentication**
- Protected the **profile API** with token-based authentication
- Tested the API using **Postman and cURL**

14.15 Conclusion

In this chapter, we delved into the process of building REST APIs using Laravel 12, covering a wide range of topics essential for modern API development. We explored creating a calculator API, building a CRUD resource API for product management, and implementing file uploads using form-data. Additionally, we set up secure authentication and authorization using Laravel Sanctum, enabling user registration, login, and profile management. These features were tested using tools like Postman and cURL to ensure functionality and reliability.

We also highlighted Laravel's powerful built-in features, such as Eloquent ORM for database interactions, middleware for route protection, and validation for input handling. By leveraging these tools, we demonstrated how to handle error responses, format API outputs, and manage file storage effectively. This chapter provides a solid foundation for building robust, secure, and scalable REST APIs with Laravel 12, equipping you with the skills to tackle real-world API development challenges.

OceanofPDF.com

15 Frontend Development with Starter Kits

Laravel 12 brings a fresh and simplified approach to full-stack development by integrating popular frontend stacks directly into the project scaffolding process. Using the new Laravel Installer CLI, developers can now select a frontend starter kit interactively at the time of project creation. This seamless integration allows for quick setup of modern frontend technologies—**React**, **Vue.js**, or **Livewire**—without needing to install them manually after project creation.

In this chapter, you will explore how Laravel 12 supports modern frontend development through starter kits. You will learn how to initialize a Laravel project using the Laravel CLI, choose and configure a frontend stack, and build simple interactive components using React, Vue.js, or Livewire. You will also compare the features and use cases of each frontend option to help guide your selection for real-world applications.

15.1 Introduction to Laravel Frontend Scaffolding

Frontend development in Laravel 12 has been streamlined with the improved **Laravel Installer CLI**. Instead of manually installing Breeze or Jetstream via Composer, developers can now choose a frontend stack during the project creation process. Laravel 12 supports three official starter kits: **React**, **Vue.js**, and **Livewire**. These kits are powered by Breeze or Jetstream behind the scenes but are presented through a user-friendly selection menu when running `laravel new`.

This new approach makes Laravel even more approachable for both new and experienced developers by minimizing configuration overhead and enabling a quick start with a full-stack environment.

15.2 React Starter Kit

React is a declarative, component-based JavaScript library used to build dynamic user interfaces. Laravel 12 allows developers to scaffold a new Laravel + React project right from the beginning using the Laravel CLI.

To create a new project with React as the frontend stack, run the following command:

```
| laravel new laravel-react-app
```

You will be prompted to choose a starter kit:



```
| Which starter kit would you like to install? [None]:  
| [none] None  
| [react] React  
| [vue] Vue  
| [livewire] Livewire  
| >react
```

After selecting **React**, Laravel installs Breeze with React, sets up Tailwind CSS, Vite, and configures authentication and basic pages.

The frontend code is placed under `resources/js`, with React components located in `Components` and `Pages`. A good starting point is to create a component like `Counter.jsx`, where you can manage state using the `useState` hook and respond to events like button clicks. This helps you grasp React's interactive capabilities right within your Laravel app.

React communicates with Laravel backend routes via HTTP, typically using `axios`. You can fetch API data, update components, and maintain a responsive UI without full page reloads.

15.3 Vue.js Starter Kit

Vue.js is a progressive JavaScript framework known for its ease of use and flexibility. Laravel 12 supports Vue.js directly through the Laravel Installer. The setup process is nearly identical to React, but with a different selection.

To create a new Laravel project with Vue.js:

```
| laravel new laravel-vue-app
```

Select Vue.js from the starter kit prompt:



```
| | /` | / \ \ / - \ | | | |
| | ( | | | ( | \| v / |  
| | \_,_| \ \_,_| \ / \_|_ |
```

```
| Which starter kit would you like to install? [None]:
```

- [none] None
- [react] React
- [vue] Vue
- [livewire] Livewire

```
|>vue
```

Laravel then sets up Breeze with Vue.js, complete with authentication views, Tailwind CSS, and Vite.

Vue components reside under `resources/js/Components` and `resources/js/Pages`. You can build single-file components (`.vue`) that include HTML templates, JavaScript logic, and scoped styles. For instance, a simple to-do list Vue component can demonstrate two-way binding with `v-model`, conditional rendering with `v-if`, and list iteration using `v-for`.

Vue also integrates easily with Laravel's backend APIs using `axios`, letting you fetch and submit data asynchronously.

15.4 Livewire Starter Kit

Livewire offers a Laravel-native way to build interactive web interfaces without writing much JavaScript. It's especially suited for teams or developers who are comfortable with Blade and want to keep the entire development stack within Laravel.

To create a Laravel project using Livewire, run:

```
| laravel new laravel-livewire-app
```

Then choose the Livewire starter kit:

```
| |  
| | /--|---|---| / \ / / - \ | | |
| | ( | | | | ( | \| v / |  
| | \_,_| \ \_,_| \ / \_|_ |
```

```
| Which starter kit would you like to install? [None]:
```

- [none] None
- [react] React
- [vue] Vue
- [livewire] Livewire

```
|>livewire
```

Laravel installs Breeze with Blade and integrates Livewire out-of-the-box. Livewire components combine PHP classes and Blade templates. These components can manage their own state and emit events that are handled via AJAX behind the scenes, without the developer needing to write any JavaScript.

As an example, you can build a simple form that validates user input and updates the UI in real time. Livewire's server-driven approach keeps everything cohesive and is ideal for teams already familiar with Laravel conventions.

15.5 Comparison: React vs Vue.js vs Livewire

Each starter kit supported by Laravel 12 is tailored to a different developer experience and application requirement. Here's a side-by-side comparison:

Feature	React	Vue.js	Livewire
JS Dependency	High	Moderate	Low
SPA Capable			(Hybrid)
Component-based			
API Interaction Needed			
Laravel Integration	Moderate	Moderate	Deep
Real-time Interactivity	(via JS)	(via JS)	
Dev Experience	JavaScript-heavy	Balanced	PHP-first

Laravel 12's revamped CLI experience simplifies frontend scaffolding by letting developers choose their frontend stack at the moment of project creation. With support for React, Vue.js, and Livewire, Laravel provides flexibility for a wide range of frontend strategies—from JavaScript-heavy SPAs to backend-driven, Blade-based interactivity. Each approach has its advantages, and Laravel makes it easy to get started with any of them in just a few commands.

15.6 Exercise 47: Full Stack Product CRUD with Laravel 12 API and React UI

15.6.1 Description

React is a popular JavaScript library for building user interfaces, particularly single-page applications. It allows developers to create reusable UI components and manage the state of their applications efficiently. React is often used in conjunction with other libraries and frameworks to build full-stack applications.

In this hands-on lab, we will build a full stack web application using Laravel 12 and React. The backend will expose a RESTful API for managing `Product` data, and the frontend will be a React interface for performing Create, Read, Update, and Delete operations. We will use the Laravel Starter Kit with React and SQLite as the database.

15.6.2 Objectives

By the end of this lab, you will be able to:

- Scaffold a Laravel 12 project with the React starter kit using Laravel CLI
- Create a RESTful Product API using Laravel
- Connect the React frontend to the backend via `axios`
- Perform full CRUD operations from the React interface
- Use SQLite as the persistent database

15.6.3 Prerequisites

Before starting, make sure the following are installed:

- PHP ≥ 8.2
- Composer
- Laravel Installer (`composer global require laravel/installer`)
- Node.js and npm
- SQLite installed on your system
- A text editor (e.g., Visual Studio Code)

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

15.6.4 Steps

Here are the steps to create a full stack product CRUD application using Laravel 12 and React:

15.6.4.1 Step 1: Create a New Laravel Project with React

We will use the Laravel Installer to create a new project with React as the frontend stack. Run the following command:

```
| laravel new product-crud-app
```

When prompted:



```
| Which starter kit would you like to install? [None]:
```

```
[none] None
[react] React
[vue] Vue
[livewire] Livewire
>react
```

Select **React** as the starter kit by typing `react`. Accept the default options for Authentication and Unit Testing. This will install the Laravel Breeze starter kit with React, which includes authentication scaffolding and a basic layout.

This will take a few moments to set up. Once complete, navigate into the project directory:

```
| cd product-crud-app
| code .
```

You should see the project structure in your code editor.

You can find the React app in the `resources/js` directory. The main entry point is `resources/js/app.jsx`, which is where the React app is bootstrapped.

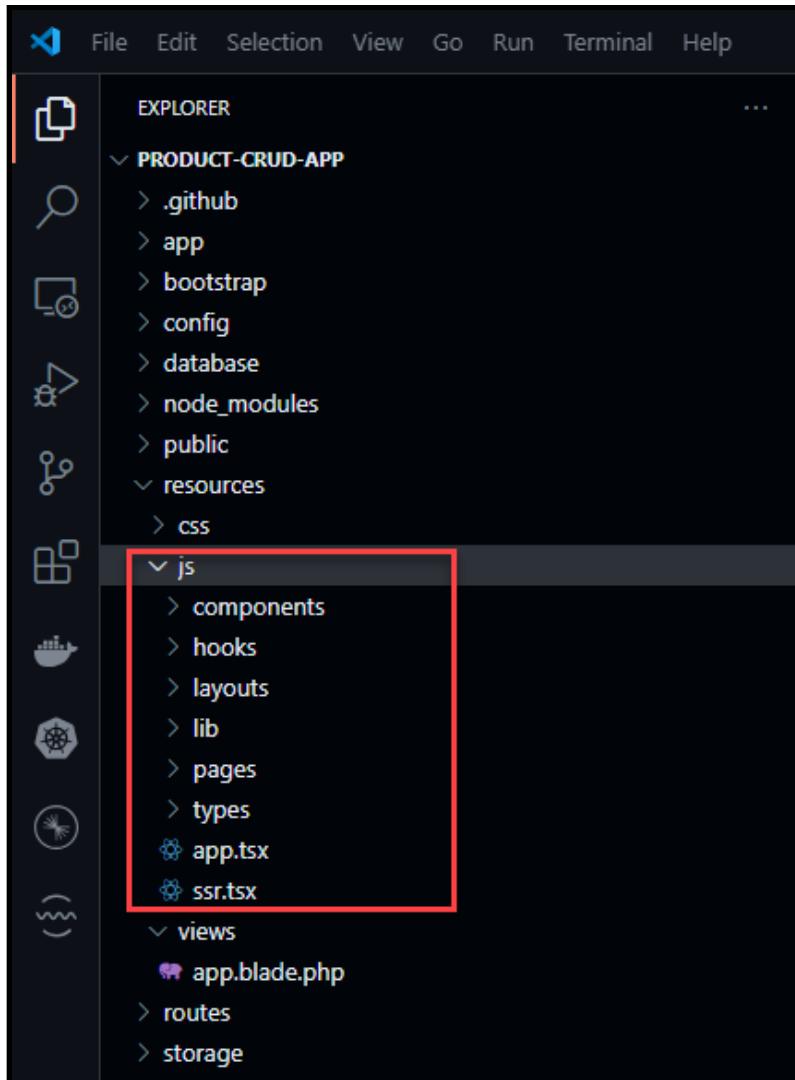


Figure 15.1: React app inside Laravel project.

15.6.4.2 Step 2: Configure SQLite Database

Since we are accepting SQLite as the database, we can verify the `.env` file to ensure it is set up correctly. Open the `.env` file in the root of your project and check the database connection settings.

```
| DB_CONNECTION=sqlite
```

We will keep the default SQLite settings.

15.6.4.3 Step 3: Create Product Model, Migration, and Controller

We will create a `Product` model, migration, and controller using the Artisan command line tool. Run the following command in your terminal:

```
| php artisan make:model Product -mcr
```

We have now created the `Product` model, migration, and controller. The migration file creates a `products` table with `name`, `description`, and `price` fields. The `-mcr` flag creates a migration file, a resource controller, and a model.

In the generated migration (`database/migrations/xxxx_xx_xx_create_products_table.php`), update the schema:

```
public function up()
{
    Schema::create('products', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->text('description')->nullable();
        $table->decimal('price', 10, 2);
        $table->timestamps();
    });
}
```

Run the migration:

```
| php artisan migrate
```

Modify model `app/Models/Product.php` to include the fillable fields:

```
class Product extends Model
{
    protected $fillable = ['name', 'description', 'price'];
}
```

15.6.4.4 Step 4: Implement ProductController

In the generated `ProductController` (`app/Http/Controllers/ProductController.php`), implement the CRUD methods. Use the `Inertia` facade to render React components.

In `app/Http/Controllers/ProductController.php`, implement the CRUD methods:

```
<?php

namespace App\Http\Controllers;

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

```
class ProductController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
    {
        return Inertia::render('products/index', [
            'products' => Product::all()
        ]);
    }

    /**
     * Show the form for creating a new resource.
     */
    public function create()
    {
        return Inertia::render('products/create');
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
    {
        $request->validate([
            'name' => 'required|string',
            'description' => 'nullable|string',
            'price' => 'required|numeric'
        ]);

        Product::create($request->all());
        return redirect()->route('products.index')->with('success', 'Product created!');
    }

    /**
     * Display the specified resource.
     */
    public function show(Product $product)
    {
        return Inertia::render('products/show', ['product' => $product]);
    }

    /**
     * Show the form for editing the specified resource.
     */
    public function edit(Product $product)
    {
        return Inertia::render('products/edit', ['product' => $product]);
    }

    /**
     * Update the specified resource in storage.
     */
    public function update(Request $request, Product $product)
    {
        $request->validate([
            'name' => 'required|string',
            'description' => 'nullable|string',
        ]);
    }
}
```

```

        'price' => 'required|numeric'
    ]);

    $product->update($request->all());
    return redirect()->route('products.index')->with('success', 'Product updated!');
}

/**
 * Remove the specified resource from storage.
 */
public function destroy(Product $product)
{
    $product->delete();
    return redirect()->route('products.index')->with('success', 'Product deleted!');
}
}

```

This controller handles all CRUD operations for the `Product` model. It uses Inertia to render React components for each action.

- The `index` method fetches all products and passes them to the `products/index` React component. The `create`, `edit`, and `show` methods render their respective components. The `store`, `update`, and `destroy` methods handle the creation, updating, and deletion of products, respectively.
- The `store` and `update` methods validate the incoming request data before creating or updating a product.
- The `destroy` method deletes the specified product.
- The `ProductController` uses the `Inertia` facade to render React components. The `index`, `create`, `edit`, and `show` methods return Inertia responses that load the corresponding React components.

15.6.4.5 Step 5: Create API Routes

In `routes/web.php`, define the routes for the `ProductController`:

```

use App\Http\Controllers\ProductController;

Route::middleware(['auth', 'verified'])->group(function () {
    Route::get('/products', [ProductController::class, 'index'])->name('products.index');
    Route::get('/products/create', [ProductController::class, 'create'])->name('products.create');
    Route::get('/products/{product}', [ProductController::class, 'show'])->name('products.show');
    Route::post('/products', [ProductController::class, 'store'])->name('products.store');
    Route::get('/products/{product}/edit', [ProductController::class, 'edit'])->name('products.edit');
    Route::put('/products/{product}', [ProductController::class, 'update'])->name('products.update');
});

```

```
|     Route::delete('/products/{product}', [ProductController::class, 'destroy'])->name('product.destroy');
| });
| 
```

This sets up the routes for the `ProductController` methods. The routes are protected by authentication middleware, ensuring that only authenticated users can access them.

15.6.4.6 Step 6: Set Up React Frontend

We will create a simple React component to manage products. Create a folder named `products` inside `resources/js/pages` and create the following files:

- `index.tsx`
- `create.tsx`
- `edit.tsx`
- `show.tsx`

Create `resources/js/pages/products/index.tsx` and replace with the following code:

```
import AppLayout from '@/layouts/app-layout'
import { Head, Link, usePage } from '@inertiajs/react'
import { type BreadcrumbItem } from '@/types'

const breadcrumbs: BreadcrumbItem[] = [
  { title: 'Products', href: '/products' },
]

export default function ProductIndex({ products }: { products: any[] }) {
  const { flash } = usePage().props as { flash?: { success?: string } }

  return (
    <AppLayout breadcrumbs={breadcrumbs}>
      <Head title="Products" />

      <div className="container mx-auto p-4">
        {flash?.success && <div className="bg-green-100 text-green-800 p-2 mb-4 rounded">
          {flash.success}
        </div>}

        <div className="flex justify-between items-center mb-4">
          <h1 className="text-2xl font-bold">Products</h1>
          <Link
            href="/products/create"
            className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600"
          >
            Create
          </Link>
        </div>

        <div className="overflow-x-auto">
          <table className="min-w-full bg-white dark:bg-gray-800 shadow rounded-lg text-sm">
            <thead>
              <tr className="bg-gray-200 dark:bg-gray-700 text-left text-gray-800 dark:text-gray-100">
                <th className="py-2 px-4 border-b border-gray-300 dark:border-gray-400" style="width: 60px;">
```

```

    600">Name</th>
    <th className="py-2 px-4 border-b border-gray-300 dark:border-gray-
600">Price</th>
    <th className="py-2 px-4 border-b border-gray-300 dark:border-gray-
600">Description</th>
    <th className="py-2 px-4 border-b border-gray-300 dark:border-gray-
600">Actions</th>
    </tr>
    </thead>
    <tbody>
      {products.map((product) => (
        <tr
          key={product.id}
          className="hover:bg-gray-100 dark:hover:bg-gray-700 border-b border-
gray-200 dark:border-gray-600 text-gray-800 dark:text-gray-100"
        >
          <td className="py-2 px-4">{product.name}</td>
          <td className="py-2 px-4">${product.price}</td>
          <td className="py-2 px-4">{product.description}</td>
          <td className="py-2 px-4 flex gap-2">
            <Link
              href={route('products.show', product.id)}
              className="bg-indigo-500 hover:bg-indigo-600 text-white px-3 py-1
rounded text-xs mr-2"
            >
              Detail
            </Link>

            <Link
              href={`/products/${product.id}/edit`}
              className="bg-green-500 hover:bg-green-600 text-white px-3 py-1
rounded text-xs"
            >
              Edit
            </Link>
            <Link
              href={route('products.destroy', product.id)}
              method="delete"
              as="button"
              className="bg-red-500 hover:bg-red-600 text-white px-3 py-1
rounded text-xs"
            >
              onClick={(e) => {
                if (!confirm('Are you sure you want to delete this product?'))
                  e.preventDefault()
                }
              }
            >
              Delete
            </Link>
          </td>
        </tr>
      )))
    </tbody>
  </table>
</div>
</div>
</AppLayout>
)
}

```

This component displays a list of products in a table format. It uses Inertia.js to handle navigation and state management. The `Link` component is used to create links to the product creation and editing pages.

Create resources/js/pages/products/create.tsx and replace with the following code:

```
import AppLayout from '@/layouts/app-layout'
import { Head, useForm } from '@inertiajs/react'
import { type BreadcrumbItem } from '@/types'

const breadcrumbs: BreadcrumbItem[] = [
    { title: 'Products', href: '/products' },
    { title: 'Create Product', href: '/products/create' }
]

export default function ProductCreate() {
    const { data, setData, post, processing, errors } = useForm({
        name: '',
        price: '',
        description: ''
    })

    const handleSubmit = (e: React.FormEvent) => {
        e.preventDefault()
        post('/products')
    }

    return (
        <AppLayout breadcrumbs={breadcrumbs}>
            <Head title="Create Product" />
            <div className="container mx-auto p-4 max-w-xl">
                <h1 className="text-2xl font-bold mb-4">Create Product</h1>

                <form onSubmit={handleSubmit} className="space-y-4">
                    <div>
                        <input
                            type="text"
                            className="form-input w-full"
                            placeholder="Product name"
                            value={data.name}
                            onChange={(e) => setData('name', e.target.value)}
                        />
                        {errors.name && <div className="text-red-500 text-sm">{errors.name}</div>}
                    </div>

                    <div>
                        <input
                            type="text"
                            className="form-input w-full"
                            placeholder="Price"
                            value={data.price}
                            onChange={(e) => setData('price', e.target.value)}
                        />
                        {errors.price && <div className="text-red-500 text-sm">{errors.price}</div>}
                    </div>

                    <div>
                        <textarea
                            className="form-textarea w-full"
                            placeholder="Description"
                            value={data.description}
                            onChange={(e) => setData('description', e.target.value)}
                        />
                        {errors.description && <div className="text-red-500 text-sm">{errors.description}</div>}
                    </div>

                    <button
                        type="submit"
                        className="w-full py-2 px-4 rounded-md bg-blue-500 text-white font-semibold
                        transition-colors duration-200 ease-in-out
                        :hover:bg-blue-600
                        :disabled:disabled">Create Product</button>
                </form>
            </div>
        </AppLayout>
    )
}
```

```

        type="submit"
        className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600"
        disabled={processing}
      >
      Save
    </button>
  </form>
</div>
</AppLayout>
)
}

```

This component provides a form for creating a new product. It uses the `useForm` hook from Inertia.js to manage form state and handle submission. The `post` method sends the form data to the backend API.

Create `resources/js/pages/products/edit.tsx` and replace with the following code:

```

import AppLayout from '@/layouts/app-layout'
import { Head, useForm } from '@inertiajs/react'
import { type BreadcrumbItem } from '@/types'

export default function ProductEdit({ product }: { product: any }) {
  const breadcrumbs: BreadcrumbItem[] = [
    { title: 'Products', href: '/products' },
    { title: 'Edit Product', href: `/products/${product.id}/edit` }
  ]

  const { data, setData, put, processing, errors } = useForm({
    name: product.name,
    price: product.price,
    description: product.description || '',
  })

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault()
    put(`/products/${product.id}`)
  }

  return (
    <AppLayout breadcrumbs={breadcrumbs}>
      <Head title="Edit Product" />
      <div className="container mx-auto p-4 max-w-xl">
        <h1 className="text-2xl font-bold mb-4">Edit Product</h1>

        <form onSubmit={handleSubmit} className="space-y-4">
          <div>
            <input
              type="text"
              className="form-input w-full"
              placeholder="Product name"
              value={data.name}
              onChange={(e) => setData('name', e.target.value)}
            />
            {errors.name && <div className="text-red-500 text-sm">{errors.name}</div>}
          </div>
          <div>
            <input
              type="text"
              className="form-input w-full"
              placeholder="Price"
              value={data.price}
            />
          </div>
        </form>
      </div>
    </AppLayout>
  )
}

```

```

        onChange={(e) => setData('price', e.target.value)}
      />
      {errors.price && <div className="text-red-500 text-sm">{errors.price}</div>}
    </div>

    <div>
      <textarea
        className="form-textarea w-full"
        placeholder="Description"
        value={data.description}
        onChange={(e) => setData('description', e.target.value)}
      />
      {errors.description && <div className="text-red-500 text-sm">
{errors.description}</div>}
    </div>

    <button
      type="submit"
      className="bg-yellow-500 text-white px-4 py-2 rounded hover:bg-yellow-600"
      disabled={processing}
    >
      Update
    </button>
  </form>
</div>
</AppLayout>
)
}
}

```

This component provides a form for editing an existing product. It pre-fills the form fields with the current product data and uses the `put` method to send the updated data to the backend API.

Create `resources/js/pages/products/show.tsx` and replace with the following code:

```

import AppLayout from '@/layouts/app-layout'
import { Head } from '@inertiajs/react'
import { type BreadcrumbItem } from '@types'

export default function ProductShow({ product }: { product: any }) {
  const breadcrumbs: BreadcrumbItem[] = [
    { title: 'Products', href: '/products' },
    { title: 'Detail Product', href: `/products/${product.id}` },
  ]

  return (
    <AppLayout breadcrumbs={breadcrumbs}>
      <Head title={`Product: ${product.name}`} />

      <div className="container mx-auto p-4 max-w-xl">
        <h1 className="text-2xl font-bold mb-4">Product Detail</h1>

        <div className="bg-white dark:bg-gray-800 rounded shadow p-4 space-y-4 text-gray-900 dark:text-white">
          <div><strong>Name:</strong> {product.name}</div>
          <div><strong>Price:</strong> ${product.price}</div>
          <div><strong>Description:</strong> {product.description}</div>
          <div className="mt-4">
            <a
              href="/products"
              className="text-blue-600 hover:underline"
            >
              ← Back to Products
            </a>
          </div>
        </div>
      </AppLayout>
    )
}

```

```
        </a>
    </div>
    </div>
</div>
</AppLayout>
)
}
```

This component displays the details of a single product. It shows the product name, price, and description. A link is provided to navigate back to the product list.

We create menu `Products` in dashboard. Open `resources/js/components/app-sidebar.tsx` and add the following code, a part of the `mainNavItems` array:

```
...
const mainNavItems: NavItem[] = [
{
    title: 'Dashboard',
    href: '/dashboard',
    icon: LayoutGrid,
},
{
    title: 'Products',
    href: '/products',
    icon: BookOpen,
},
];
...
```

This adds a link to the `Products` page in the sidebar navigation. The `icon` property specifies the icon to be displayed next to the link.

Last, we need to change `.env` file to set the `APP_URL` to `http://localhost:8000`:

```
| APP_URL=http://localhost:8000
```

This is important for the Vite development server to work correctly with the Laravel backend.

Save all the files.

15.6.4.7 Step 7: Test the App

After completing the above steps, you can run the application. First, we run vite app:

```
| npm run dev
```

You should see the output similar to this:

```
VITE v6.2.0 ready in 479 ms

→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help

LARAVEL v12.9.2 plugin v1.2.0

→ APP_URL: http://localhost:8000
```

Then, open another terminal window and run:

```
| php artisan serve
```

You should see the application running at <http://localhost:8000>. You can register a new user or log in with an existing user. Once logged in, you will be redirected to the dashboard.

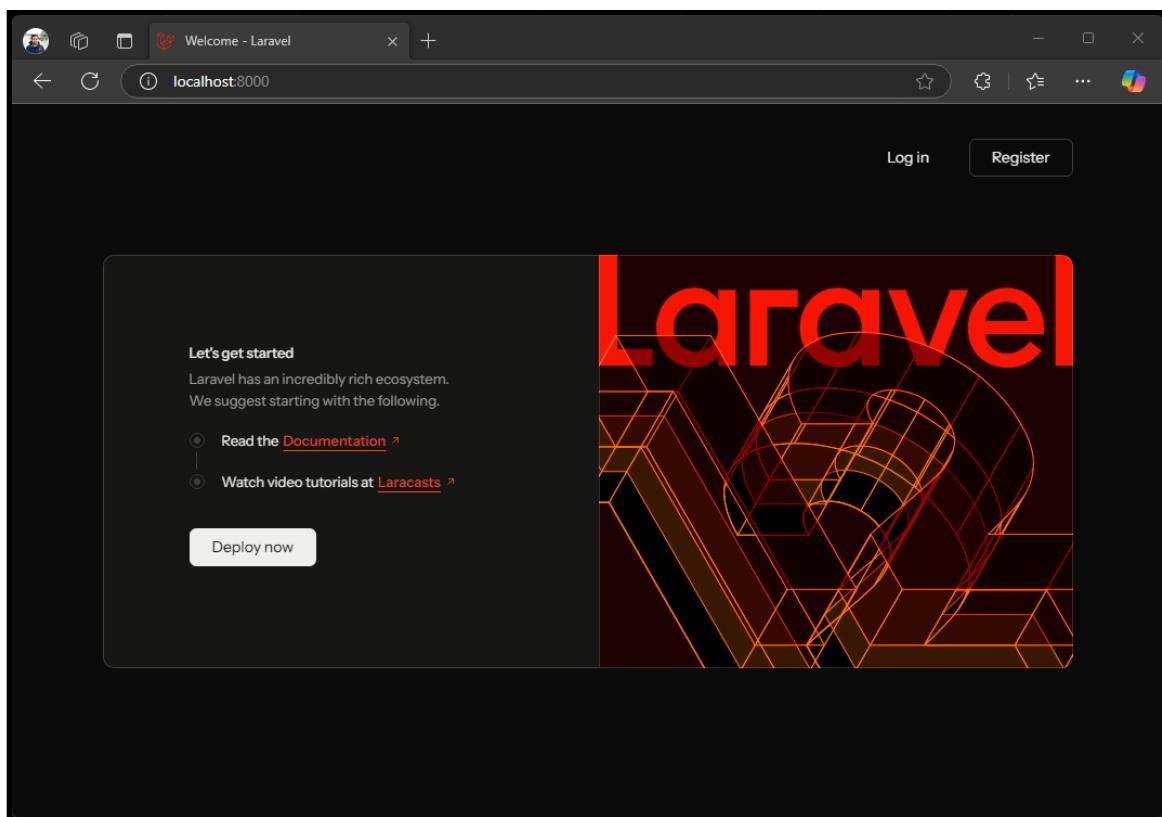


Figure 15.2: Laravel 12 + React app running.

Try to register a new user and log in. Click Register and fill in the form. After registering, you will be redirected to the dashboard.

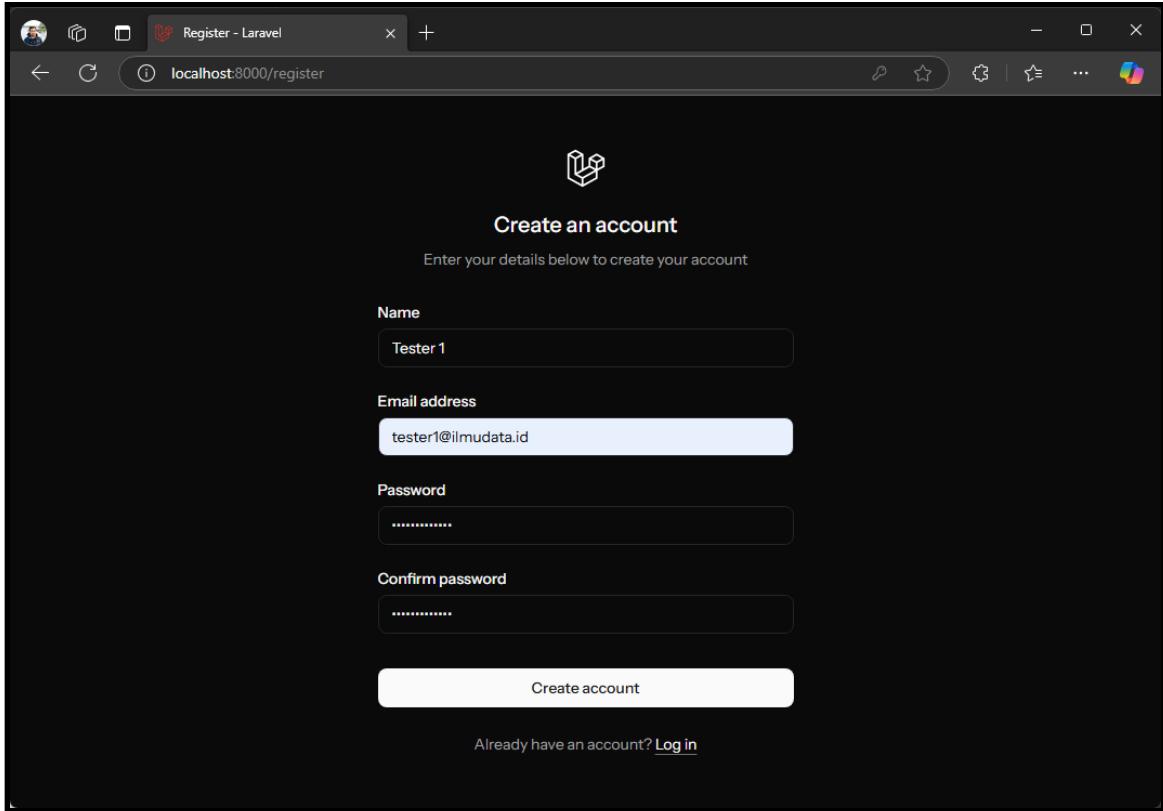


Figure 15.3: Register a new user.

After logging in, you will see the dashboard with the `Products` link in the sidebar.

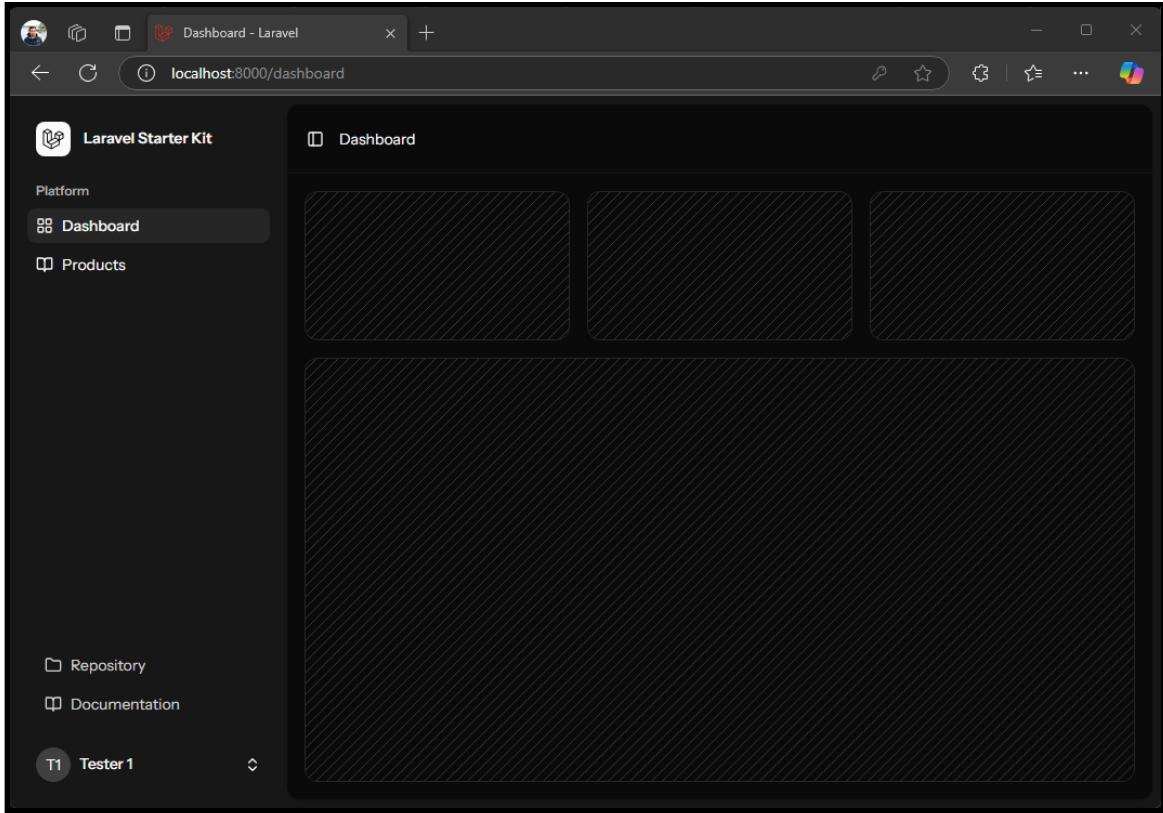


Figure 15.4: Dashboard with Products link.

Click on the `Products` link in the sidebar. You will be redirected to the products page.

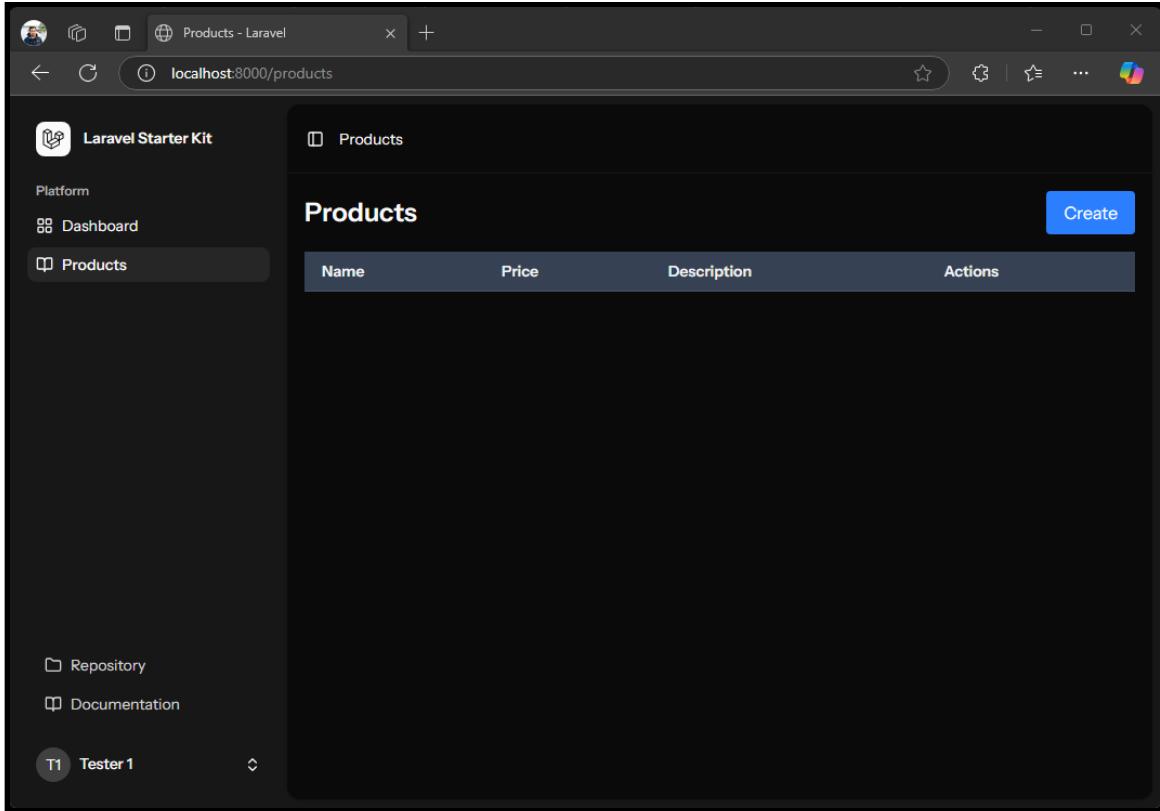


Figure 15.5: Products page.

Now, you can create a new product by clicking the `Create` button. Fill in the form and click `Save`. You will be redirected to the products page, and you should see the newly created product in the list.

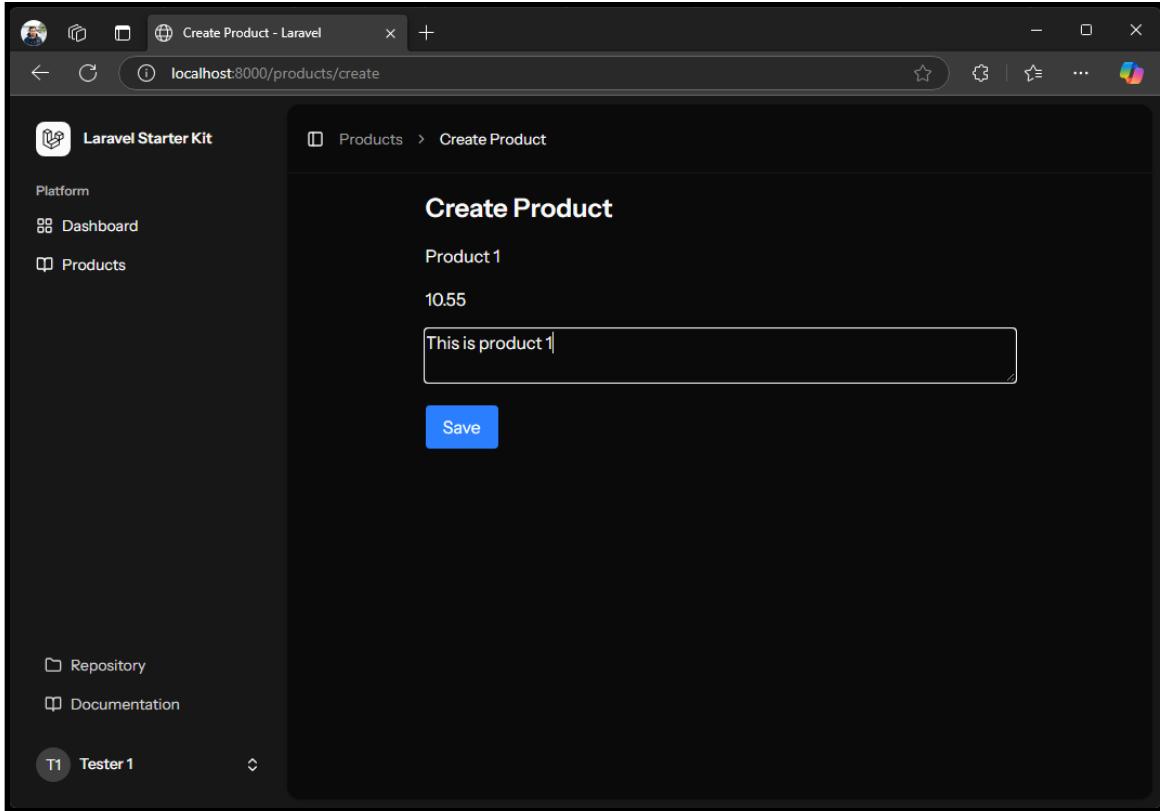


Figure 15.6: Create a new product.

Fill in the form with the product name, price, and description. Click `Save` to create the product.

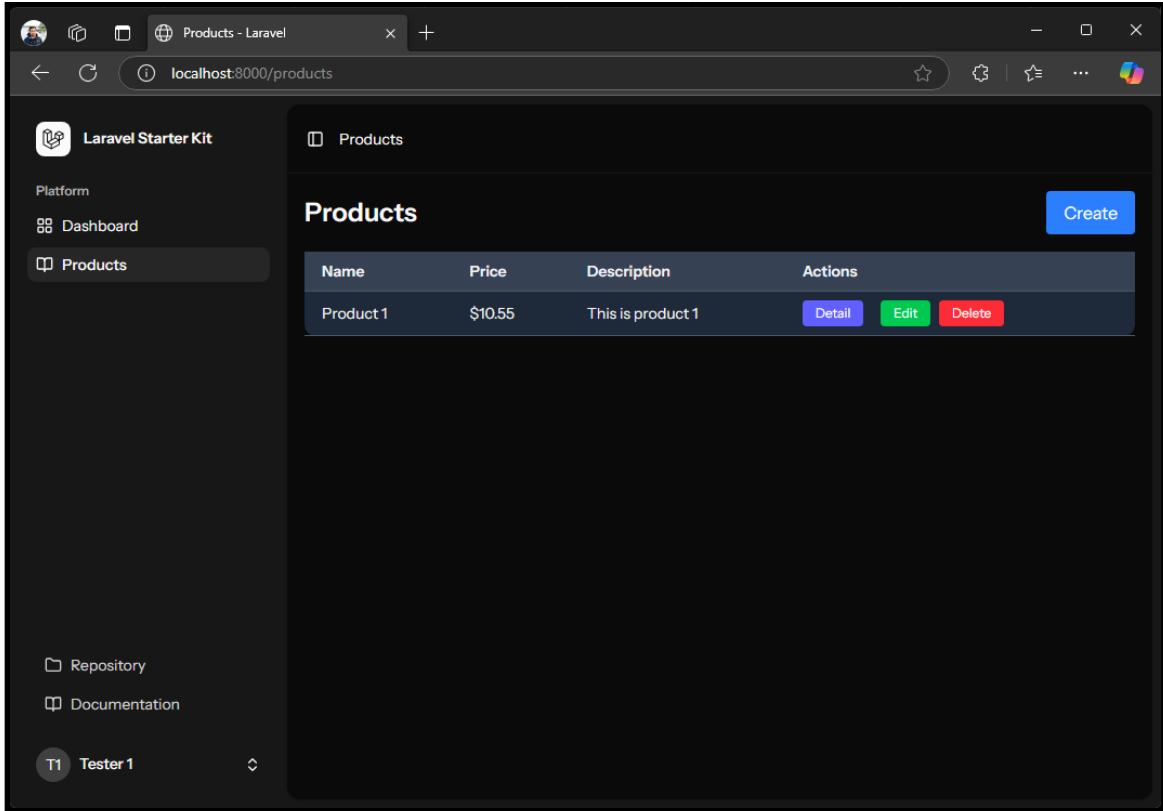


Figure 15.7: Fill in the form.

Try to create more products and see them listed on the products page.

Laravel Starter Kit

Platform

Dashboard

Products

Products

Name Price Description Actions

Product 1 \$10.55 This is product 1 Detail Edit Delete

Product 2 \$12.45 Product 2 description Detail Edit Delete

Product 3 \$80.45 Product 3 description Detail Edit Delete

Product 4 \$10.22 Product 4 description Detail Edit Delete

Create

Repository

Documentation

T1 Tester 1

Figure 15.8: Products list.

To see details of a product, click on the product name. You will be redirected to the product detail page.

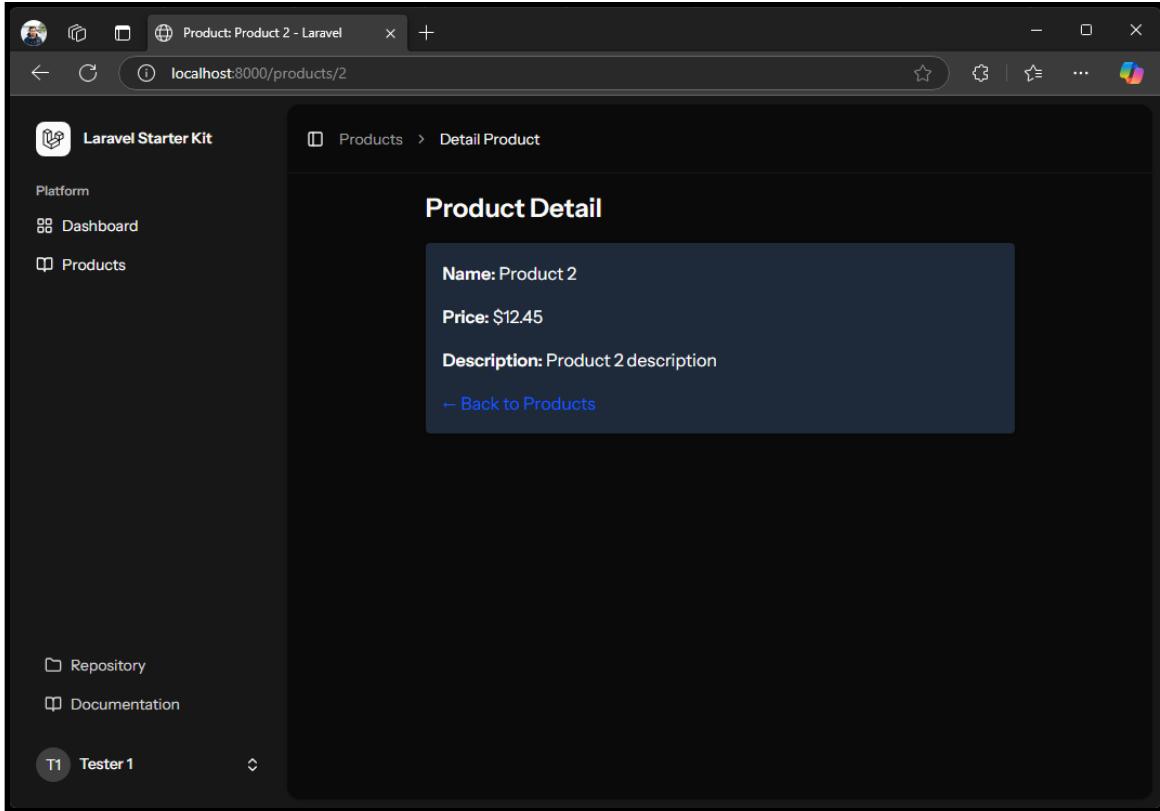
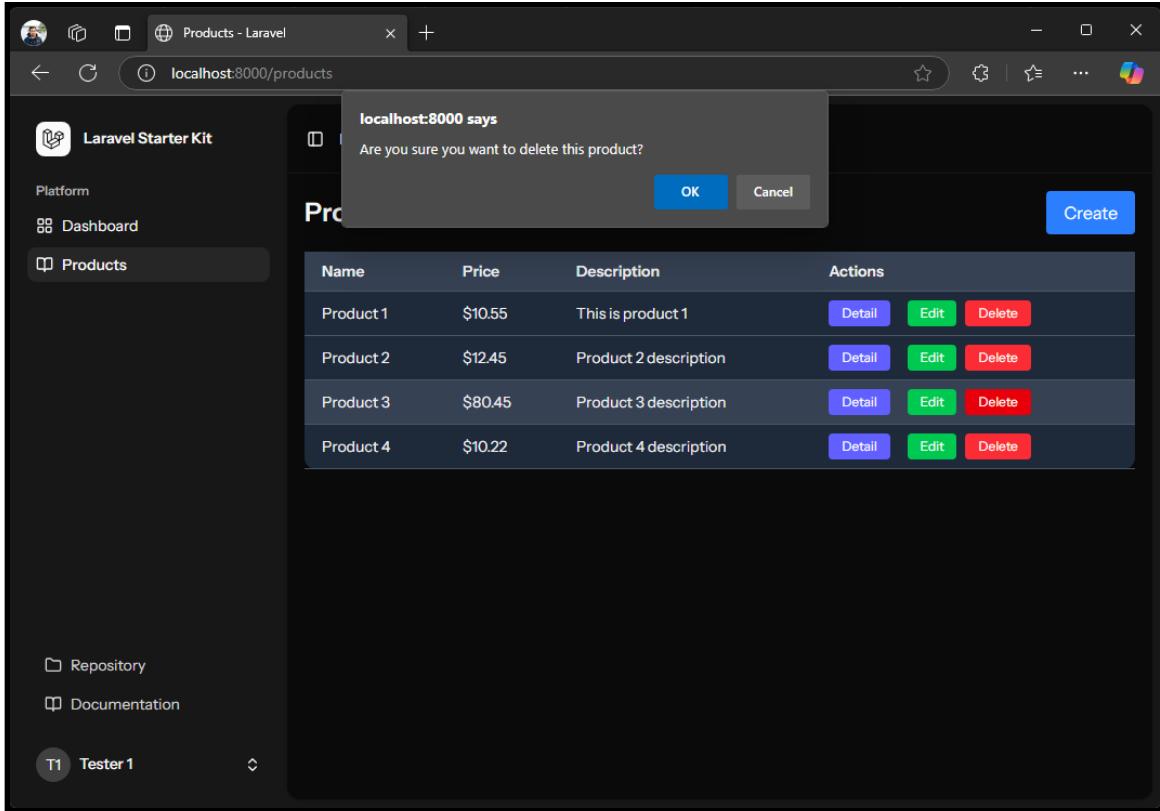


Figure 15.9: Product detail page.

And to edit a product, click the `Edit` button next to the product name. You will be redirected to the edit page.

You also can delete a product by clicking the `Delete` button. A confirmation dialog will appear. Click `OK` to delete the product.



Name	Price	Description	Actions
Product 1	\$10.55	This is product 1	<button>Detail</button> <button>Edit</button> <button>Delete</button>
Product 2	\$12.45	Product 2 description	<button>Detail</button> <button>Edit</button> <button>Delete</button>
Product 3	\$80.45	Product 3 description	<button>Detail</button> <button>Edit</button> <button>Delete</button>
Product 4	\$10.22	Product 4 description	<button>Detail</button> <button>Edit</button> <button>Delete</button>

Figure 15.10: Edit product page.

After confirming, the product will be deleted, and you will be redirected to the products page.

15.6.5 Summary

In this lab, you built a full stack Laravel 12 application using the React starter kit. You implemented a RESTful Product API and connected it to a React UI using `axios`. Laravel's official starter kits and SQLite made it fast and easy to scaffold a production-ready development stack with modern tools. This project serves as a solid foundation for real-world full stack applications using Laravel 12.

15.7 Exercise 48: Full Stack Product CRUD with Laravel 12 API and Vue.js UI

15.7.1 Description

Vue.js is a progressive JavaScript framework for building user interfaces. It is designed to be incrementally adoptable, meaning you can use it for small parts of

your application or as a full-fledged framework for building complex single-page applications (SPAs). Vue.js is known for its simplicity, flexibility, and ease of integration with other libraries or existing projects.

In this hands-on lab, we will develop a full stack web application using **Laravel 12** as the backend and **Vue.js** as the frontend. The backend will expose a RESTful API to manage product data, and the frontend will be built using Vue.js with the official Laravel 12 starter kit. The database used is SQLite for simplicity and local development.

15.7.2 Objectives

By the end of this lab, learners will be able to:

- Scaffold a Laravel 12 + Vue.js project using the Laravel CLI
- Develop RESTful API endpoints for Product CRUD operations
- Build a Vue.js interface to consume the Laravel API
- Perform Create, Read, Update, and Delete operations from the Vue UI
- Connect frontend and backend using `axios`

15.7.3 Prerequisites

Ensure the following are installed on your system:

- PHP \geq 8.2
- Composer
- Laravel Installer
- Node.js and npm
- SQLite installed
- A code editor (e.g., Visual Studio Code)

If you use **Visual Studio Code**, you can install the **Vue - Official** extension for better syntax highlighting and code completion.

15.7.4 Steps

Here are the steps to create a full stack product CRUD application using Laravel 12 and Vue.js.

15.7.4.1 Step 1: Create Laravel 12 Project with Vue.js Starter Kit

We will use the Laravel Installer to create a new project with Vue.js as the frontend stack. Run the following command:

```
| laravel new product-crud-vue
```

When prompted:



```
| Which starter kit would you like to install? [None]:  
| [none] None  
| [react] React  
| [vue] Vue  
| [livewire] Livewire  
>vue
```

Select **Vue.js** as the starter kit by typing `vue`. Accept the default options for Authentication and Unit Testing. This will install the Laravel Breeze starter kit with Vue.js, which includes authentication scaffolding and a basic layout.

This will take a few moments to set up. Once complete, navigate into the project directory:

```
| cd product-crud-vue  
| code .
```

You should see the project structure in your code editor.

You can find the React app in the `resources/js` directory. The main entry point is `resources/js/app.jsx`, which is where the React app is bootstrapped.

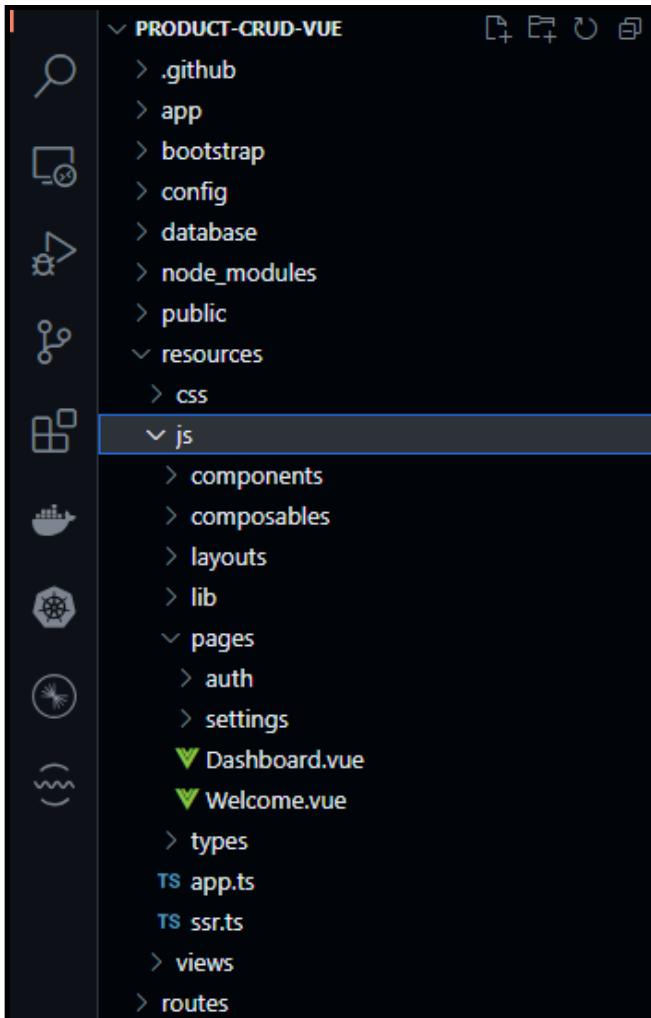


Figure 15.11: Vue app inside Laravel project.

15.7.4.2 Step 2: Configure SQLite Database

Since we are accepting SQLite as the database, we can verify the `.env` file to ensure it is set up correctly. Open the `.env` file in the root of your project and check the database connection settings.

```
| DB_CONNECTION=sqlite
```

We will keep the default SQLite settings.

15.7.4.3 Step 3: Create Product Model, Migration, and Controller

We will create a `Product` model, migration, and controller using the Artisan command line tool. Run the following command in your terminal:

```
| php artisan make:model Product -mcr
```

We have now created the `Product` model, migration, and controller. The migration file creates a `products` table with `name`, `description`, and `price` fields. The `-mcr` flag creates a migration file, a resource controller, and a model.

In the generated migration (`database/migrations/xxxx_xx_xx_create_products_table.php`), update the schema:

```
| public function up()
{
    Schema::create('products', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->text('description')->nullable();
        $table->decimal('price', 10, 2);
        $table->timestamps();
    });
}
```

Run the migration:

```
| php artisan migrate
```

Modify model `app/Models/Product.php` to include the fillable fields:

```
| class Product extends Model
{
    protected $fillable = ['name', 'description', 'price'];
}
```

15.7.4.4 Step 4: Implement ProductController

In the generated `ProductController` (`app/Http/Controllers/ProductController.php`), implement the CRUD methods. Use the `Inertia` facade to render Vue components.

In `app/Http/Controllers/ProductController.php`, implement the CRUD methods:

```
| <?php

namespace App\Http\Controllers;

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

class ProductController extends Controller
{
    /**
     * Display a listing of the resource.
     */
}
```

```
public function index()
{
    return Inertia::render('products/index', [
        'products' => Product::all()
    ]);
}

/**
 * Show the form for creating a new resource.
 */
public function create()
{
    return Inertia::render('products/create');
}

/**
 * Store a newly created resource in storage.
 */
public function store(Request $request)
{
    $request->validate([
        'name' => 'required|string',
        'description' => 'nullable|string',
        'price' => 'required|numeric'
    ]);

    Product::create($request->all());
    return redirect()->route('products.index')->with('success', 'Product created!');
}

/**
 * Display the specified resource.
 */
public function show(Product $product)
{
    return Inertia::render('products/show', ['product' => $product]);
}

/**
 * Show the form for editing the specified resource.
 */
public function edit(Product $product)
{
    return Inertia::render('products/edit', ['product' => $product]);
}

/**
 * Update the specified resource in storage.
 */
public function update(Request $request, Product $product)
{
    $request->validate([
        'name' => 'required|string',
        'description' => 'nullable|string',
        'price' => 'required|numeric'
    ]);

    $product->update($request->all());
    return redirect()->route('products.index')->with('success', 'Product updated!');
}
```

```
    /**
     * Remove the specified resource from storage.
     */
    public function destroy(Product $product)
    {
        $product->delete();
        return redirect()->route('products.index')->with('success', 'Product deleted!');
    }
}
```

This controller handles all CRUD operations for the `Product` model. It uses Inertia to render Vue components for each action.

- The `index` method fetches all products and passes them to the `products/index` Vue component. The `create`, `edit`, and `show` methods render their respective components. The `store`, `update`, and `destroy` methods handle the creation, updating, and deletion of products, respectively.
- The `store` and `update` methods validate the incoming request data before creating or updating a product.
- The `destroy` method deletes the specified product.
- The `ProductController` uses the `Inertia` facade to render Vue components. The `index`, `create`, `edit`, and `show` methods return Inertia responses that load the corresponding Vue components.

15.7.4.5 Step 5: Create API Routes

In `routes/web.php`, define the routes for the `ProductController`:

```
use App\Http\Controllers\ProductController;

Route::middleware(['auth', 'verified'])->group(function () {
    Route::get('/products', [ProductController::class, 'index'])->name('products.index');
    Route::get('/products/create', [ProductController::class, 'create'])->name('products.create');
    Route::get('/products/{product}', [ProductController::class, 'show'])->name('products.show');
    Route::post('/products', [ProductController::class, 'store'])->name('products.store');
    Route::get('/products/{product}/edit', [ProductController::class, 'edit'])->name('products.edit');
    Route::put('/products/{product}', [ProductController::class, 'update'])->name('products.update');
    Route::delete('/products/{product}', [ProductController::class, 'destroy'])->name('products.destroy');
});
```

This sets up the routes for the `ProductController` methods. The routes are protected by authentication middleware, ensuring that only authenticated users can access them.

15.7.4.6 Step 6: Set Up Vue.js Frontend

We will create a simple Vue.js component to manage products. Create a folder named `products` inside `resources/js/pages` and create the following files:

- `index.vue`
- `create.vue`
- `edit.vue`
- `show.vue`

Create `resources/js/pages/products/index.vue` and replace with the following code:

```
<template>
  <AppLayout :breadcrumbs="breadcrumbs">
    <Head title="Products" />

    <div class="container mx-auto p-4">
      <div v-if="flash.success" class="bg-green-100 text-green-800 p-2 mb-4 rounded">
        {{ flash.success }}
      </div>

      <div class="flex justify-between items-center mb-4">
        <h1 class="text-2xl font-bold">Products</h1>
        <router-link
          to="/products/create"
          class="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600">
          Create
        </router-link>
      </div>

      <div class="overflow-x-auto">
        <table class="min-w-full bg-white dark:bg-gray-800 shadow rounded-lg text-sm">
          <thead>
            <tr class="bg-gray-200 dark:bg-gray-700 text-left text-gray-800 dark:text-gray-100">
              <th class="py-2 px-4 border-b border-gray-300 dark:border-gray-600">Name</th>
              <th class="py-2 px-4 border-b border-gray-300 dark:border-gray-600">Price</th>
              <th class="py-2 px-4 border-b border-gray-300 dark:border-gray-600">Description</th>
              <th class="py-2 px-4 border-b border-gray-300 dark:border-gray-600">Actions</th>
            </tr>
          </thead>
          <tbody>
            <tr
              v-for="product in products"
              :key="product.id"
              class="hover:bg-gray-100 dark:hover:bg-gray-700 border-b border-gray-200 dark:border-gray-600 text-gray-800 dark:text-gray-100">
              <td class="py-2 px-4">{{ product.name }}</td>
              <td class="py-2 px-4">${{ product.price }}</td>
              <td class="py-2 px-4">{{ product.description }}</td>
              <td class="py-2 px-4 flex gap-2">
                <router-link
                  :to=`/products/${product.id}`>

```

```

        class="bg-indigo-500 hover:bg-indigo-600 text-white px-3 py-1 rounded
text-xs mr-2"
      >
    Detail
  </router-link>

  <router-link
    :to="`/products/${product.id}/edit`"
    class="bg-green-500 hover:bg-green-600 text-white px-3 py-1 rounded
text-xs"
  >
    Edit
  </router-link>
  <button
    @click="deleteProduct(product.id)"
    class="bg-red-500 hover:bg-red-600 text-white px-3 py-1 rounded text-xs"
  >
    Delete
  </button>
</td>
</tr>
</tbody>
</table>
</div>
</div>
</AppLayout>
</template>

<script>
import { ref } from 'vue';
import { usePage } from '@inertiajs/vue3';
import AppLayout from '@/layouts/AppLayout.vue';

export default {
  components: { AppLayout },
  setup() {
    const { flash, props } = usePage();
    const products = ref(props.products || []);
    const breadcrumbs = ref([{ title: 'Products', href: '/products' }]);

    const deleteProduct = (id) => {
      if (confirm('Are you sure you want to delete this product?')) {
        // Call API to delete product
      }
    };

    return { flash, products, breadcrumbs, deleteProduct };
  },
};
</script>

```

This component displays a list of products in a table format. It uses Vue Router for navigation and state management. The `router-link` component is used to create links to the product creation and editing pages.

Create `resources/js/pages/products/create.vue` and replace with the following code:

```

<template>
<AppLayout :breadcrumbs="breadcrumbs">
  <Head title="Create Product" />
  <div class="container mx-auto p-4 max-w-xl">
    <h1 class="text-2xl font-bold mb-4">Create Product</h1>

```

```

<form @submit.prevent="handleSubmit" class="space-y-4">
  <div>
    <input
      type="text"
      v-model="form.name"
      class="form-input w-full"
      placeholder="Product name"
    />
    <div v-if="errors.name" class="text-red-500 text-sm">{{ errors.name }}</div>
  </div>

  <div>
    <input
      type="text"
      v-model="form.price"
      class="form-input w-full"
      placeholder="Price"
    />
    <div v-if="errors.price" class="text-red-500 text-sm">{{ errors.price }}</div>
  </div>

  <div>
    <textarea
      v-model="form.description"
      class="form-textarea w-full"
      placeholder="Description"
    ></textarea>
    <div v-if="errors.description" class="text-red-500 text-sm">{{ errors.description }}</div>
  </div>

  <button
    type="submit"
    class="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600"
    :disabled="processing"
  >
    Save
  </button>
</div>
</AppLayout>
</template>

<script>
import { ref } from 'vue';
import { useForm } from '@inertiajs/vue3';
import AppLayout from '@/layouts/AppLayout.vue';

export default {
  components: { AppLayout },
  setup() {
    const { data, setData, post, processing, errors } = useForm({
      name: '',
      price: '',
      description: '',
    });

    const breadcrumbs = ref([
      { title: 'Products', href: '/products' },
      { title: 'Create Product', href: '/products/create' },
    ]);

    const handleSubmit = () => {
      post('/products');
    };
  }
}

```

```

    return { data, setData, post, processing, errors, breadcrumbs, handleSubmit };
},
</script>

```

This component provides a form for creating a new product. It uses the `useForm` hook from `Inertia.js` to manage form state and handle submission. The `post` method sends the form data to the backend API.

Create `resources/js/pages/products/edit.vue` and replace with the following code:

```

<template>
  <AppLayout :breadcrumbs="breadcrumbs">
    <Head title="Edit Product" />
    <div class="container mx-auto p-4 max-w-xl">
      <h1 class="text-2xl font-bold mb-4">Edit Product</h1>

      <form @submit.prevent="handleSubmit" class="space-y-4">
        <div>
          <input
            type="text"
            v-model="form.name"
            class="form-input w-full"
            placeholder="Product name"
          />
          <div v-if="errors.name" class="text-red-500 text-sm">{{ errors.name }}</div>
        </div>

        <div>
          <input
            type="text"
            v-model="form.price"
            class="form-input w-full"
            placeholder="Price"
          />
          <div v-if="errors.price" class="text-red-500 text-sm">{{ errors.price }}</div>
        </div>

        <div>
          <textarea
            v-model="form.description"
            class="form-textarea w-full"
            placeholder="Description"
          ></textarea>
          <div v-if="errors.description" class="text-red-500 text-sm">{{ errors.description }}</div>
        </div>

        <button
          type="submit"
          class="bg-yellow-500 text-white px-4 py-2 rounded hover:bg-yellow-600"
          :disabled="processing"
        >
          Update
        </button>
      </form>
    </div>
  </AppLayout>
</template>

<script>
import { ref } from 'vue';
import { useForm } from '@inertiajs/vue3';

```

```

import AppLayout from '@/layouts/AppLayout.vue';

export default {
  components: { AppLayout },
  props: {
    product: Object,
  },
  setup(props) {
    const { data, setData, put, processing, errors } = useForm({
      name: props.product.name,
      price: props.product.price,
      description: props.product.description || '',
    });

    const breadcrumbs = ref([
      { title: 'Products', href: '/products' },
      { title: 'Edit Product', href: `/products/${props.product.id}/edit` },
    ]);

    const handleSubmit = () => {
      put(`/products/${props.product.id}`);
    };

    return { data, setData, put, processing, errors, breadcrumbs, handleSubmit };
  },
};
</script>

```

This component provides a form for editing an existing product. It pre-fills the form fields with the current product data and uses the `put` method to send the updated data to the backend API.

Create `resources/js/pages/products/show.vue` and replace with the following code:

```

<template>
  <AppLayout :breadcrumbs="breadcrumbs">
    <Head :title="`Product: ${product.name}`" />

    <div class="container mx-auto p-4 max-w-xl">
      <h1 class="text-2xl font-bold mb-4">Product Detail</h1>

      <div class="bg-white dark:bg-gray-800 rounded shadow p-4 space-y-4 text-gray-900 dark:text-white">
        <div><strong>Name:</strong> {{ product.name }}</div>
        <div><strong>Price:</strong> ${{ product.price }}</div>
        <div><strong>Description:</strong> {{ product.description }}</div>
        <div class="mt-4">
          <router-link
            to="/products"
            class="text-blue-600 hover:underline"
          >
            ← Back to Products
          </router-link>
        </div>
      </div>
    </AppLayout>
</template>

<script>
import { ref } from 'vue';
import AppLayout from '@/layouts/AppLayout.vue';

```

```
export default {
  components: { AppLayout },
  props: {
    product: Object,
  },
  setup(props) {
    const breadcrumbs = ref([
      { title: 'Products', href: '/products' },
      { title: 'Detail Product', href: `/products/${props.product.id}` },
    ]);
    return { breadcrumbs, product: props.product };
  },
};
```

This component displays the details of a single product. It shows the product name, price, and description. A link is provided to navigate back to the product list.

Finally, update the sidebar menu in `resources/js/components/AppSidebar.vue`:

```
...
const mainNavItems: NavItem[] = [
  {
    title: 'Dashboard',
    href: '/dashboard',
    icon: LayoutGrid,
  },
  {
    title: 'Products',
    href: '/products',
    icon: BookOpen,
  },
];
...
```

This adds a link to the `Products` page in the sidebar navigation. Update the `.env` file to set the `APP_URL` to `http://localhost:8000`:

```
| APP_URL=http://localhost:8000
```

This ensures the Vite development server works correctly with the Laravel backend.

Save all the files.

15.7.4.7 Step 7: Run the App

After completing the above steps, you can run the application. First, we run vite app:

```
| npm run dev
```

You should see the output similar to this:

```
VITE v6.2.0 ready in 408 ms  
→ Local: http://localhost:5173/  
→ Network: use --host to expose  
→ press h + enter to show help  
  
LARAVEL v12.9.2 plugin v1.2.0  
  
→ APP_URL: http://localhost:8000
```

Then, open another terminal window and run:

```
php artisan serve
```

You should see the application running at <http://localhost:8000>. You can register a new user or log in with an existing user. Once logged in, you will be redirected to the dashboard.

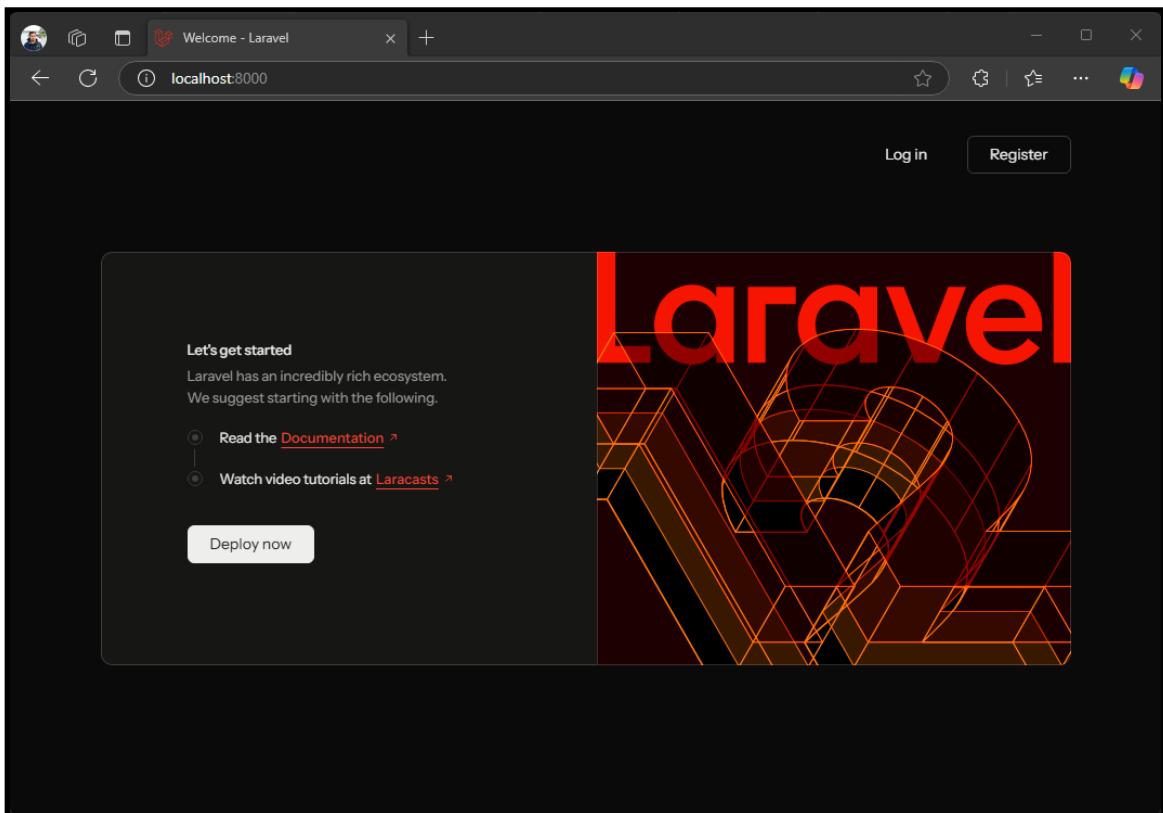


Figure 15.12: Laravel 12 + Vue app running.

Try to register a new user and log in. Click Register and fill in the form. After registering, you will be redirected to the dashboard.

After logging in, you will see the dashboard with the `Products` link in the sidebar.

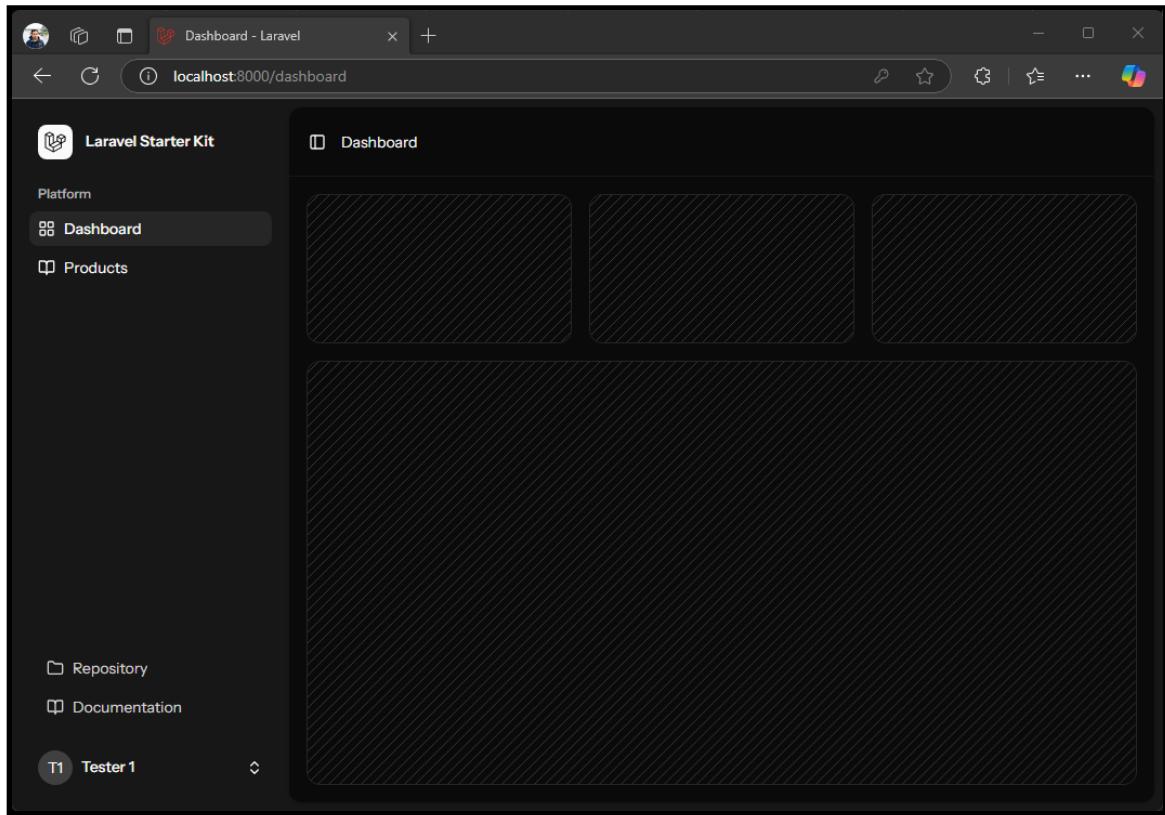


Figure 15.13: Dashboard with Products link.

Click on the `Products` link in the sidebar. You will be redirected to the products page.

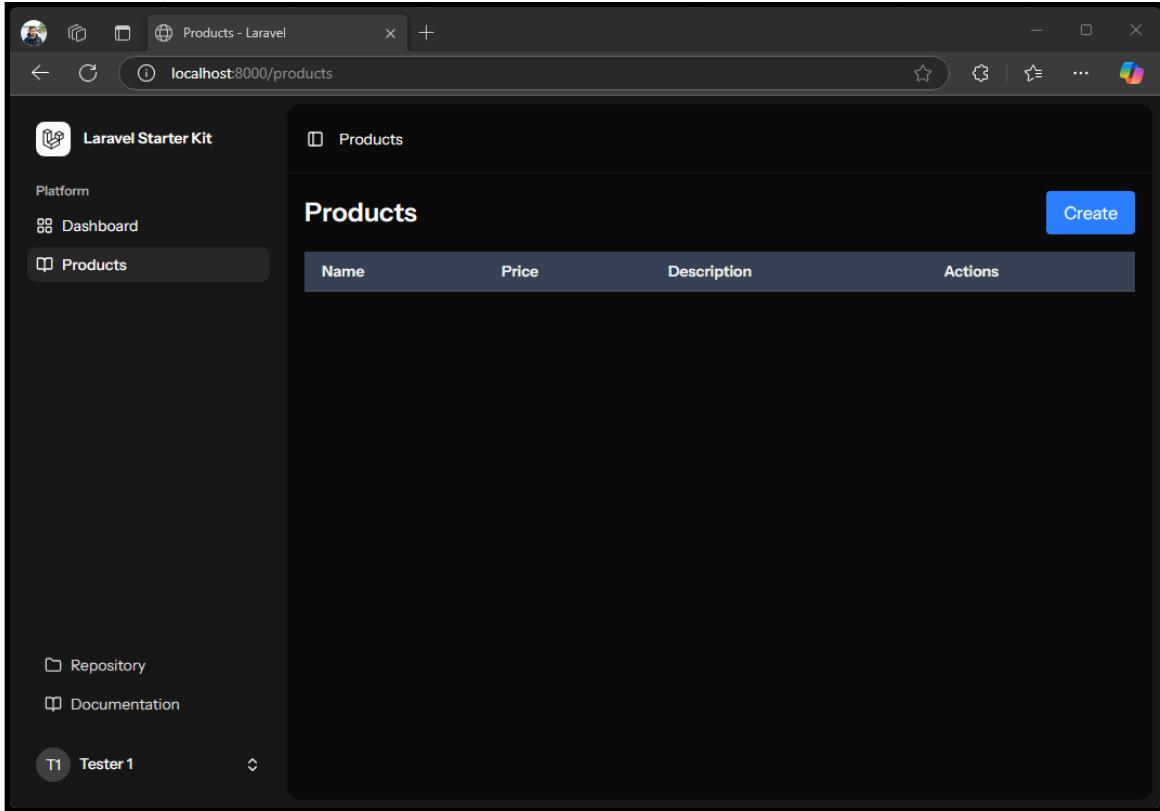


Figure 15.14: Products page.

Now, you can create a new product by clicking the `Create` button. Fill in the form and click `Save`. You will be redirected to the products page, and you should see the newly created product in the list.

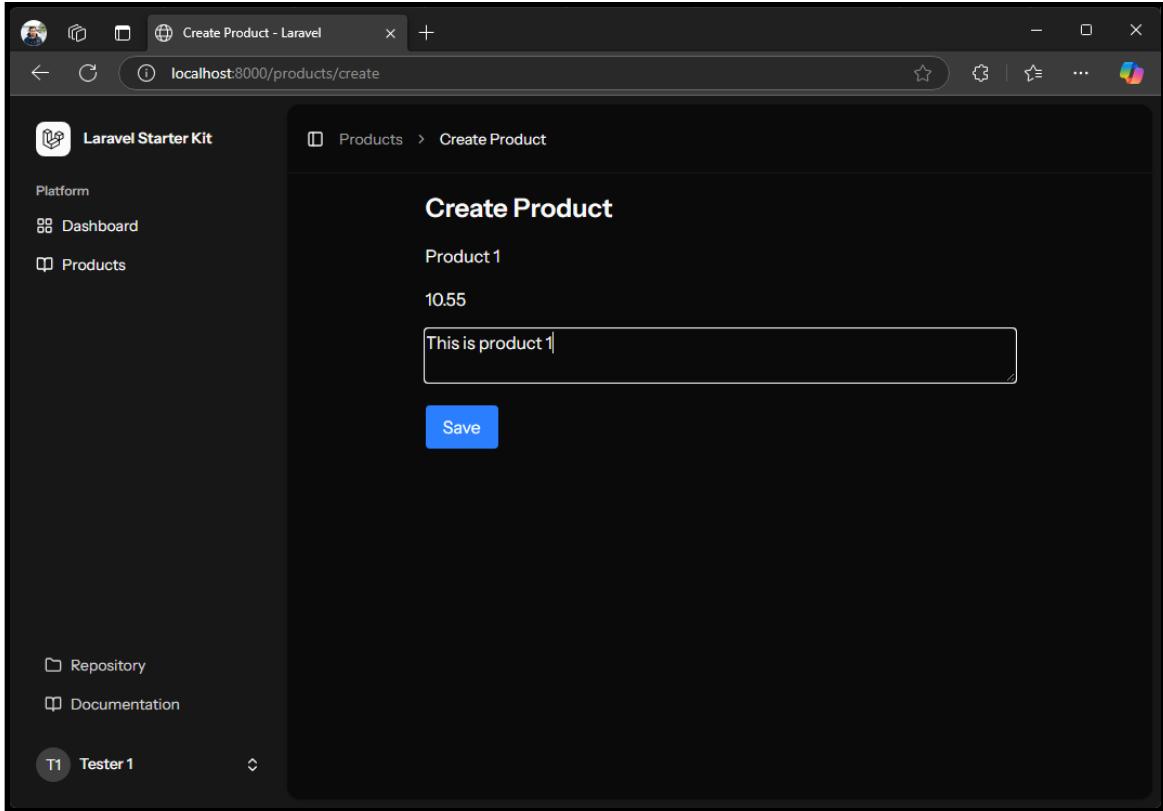


Figure 15.15: Create a new product.

Fill in the form with the product name, price, and description. Click `Save` to create the product.

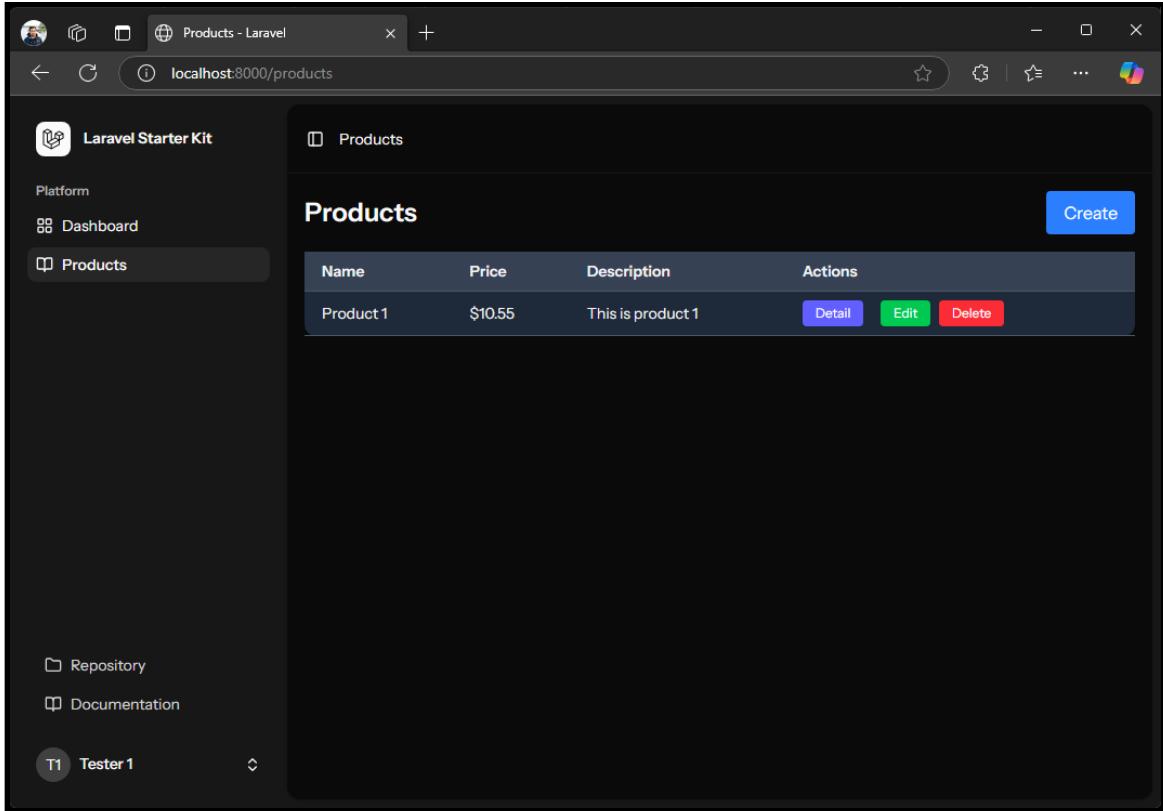


Figure 15.16: Fill in the form.

Try to create more products and see them listed on the products page.

Laravel Starter Kit

Platform

Dashboard

Products

Create

Name	Price	Description	Actions
Product 1	\$10.55	This is product 1	Detail Edit Delete
Product 2	\$12.45	Product 2 description	Detail Edit Delete
Product 3	\$80.45	Product 3 description	Detail Edit Delete
Product 4	\$10.22	Product 4 description	Detail Edit Delete

Repository

Documentation

T1 Tester 1

Figure 15.17: Products list.

To see details of a product, click on the product name. You will be redirected to the product detail page.

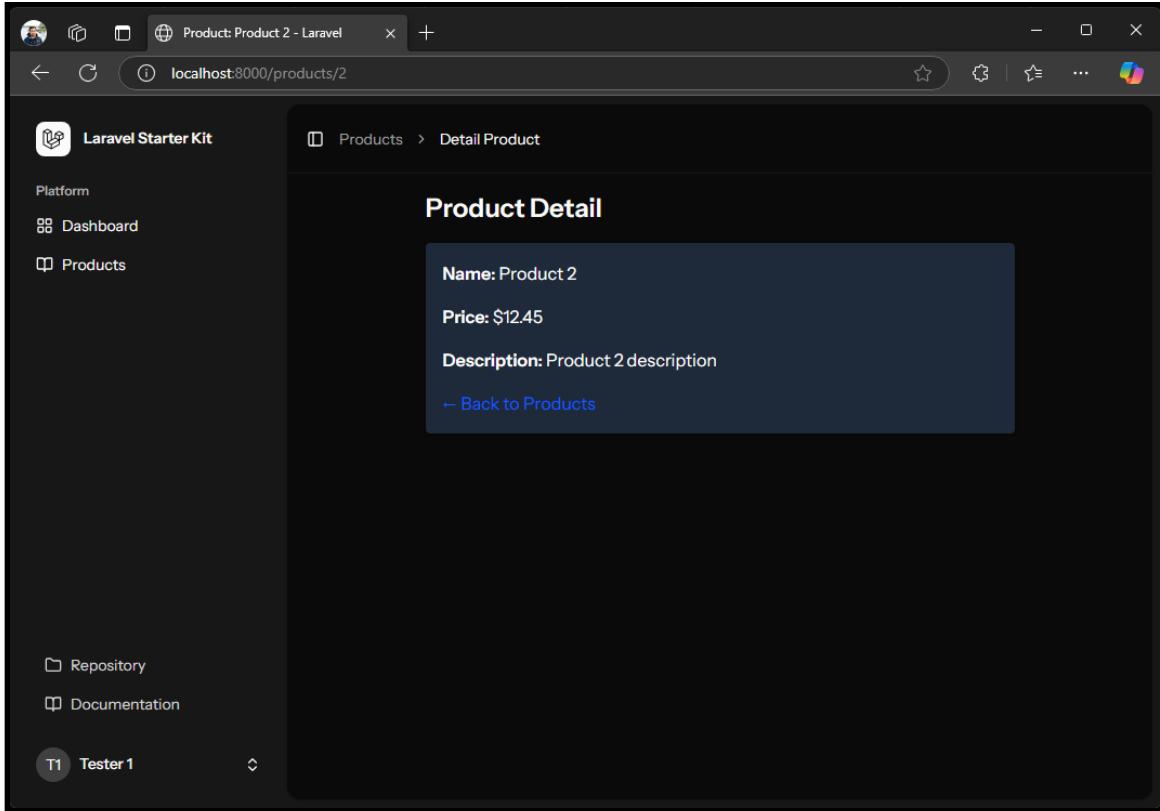


Figure 15.18: Product detail page.

And to edit a product, click the `Edit` button next to the product name. You will be redirected to the edit page.

You also can delete a product by clicking the `Delete` button. A confirmation dialog will appear. Click `OK` to delete the product.

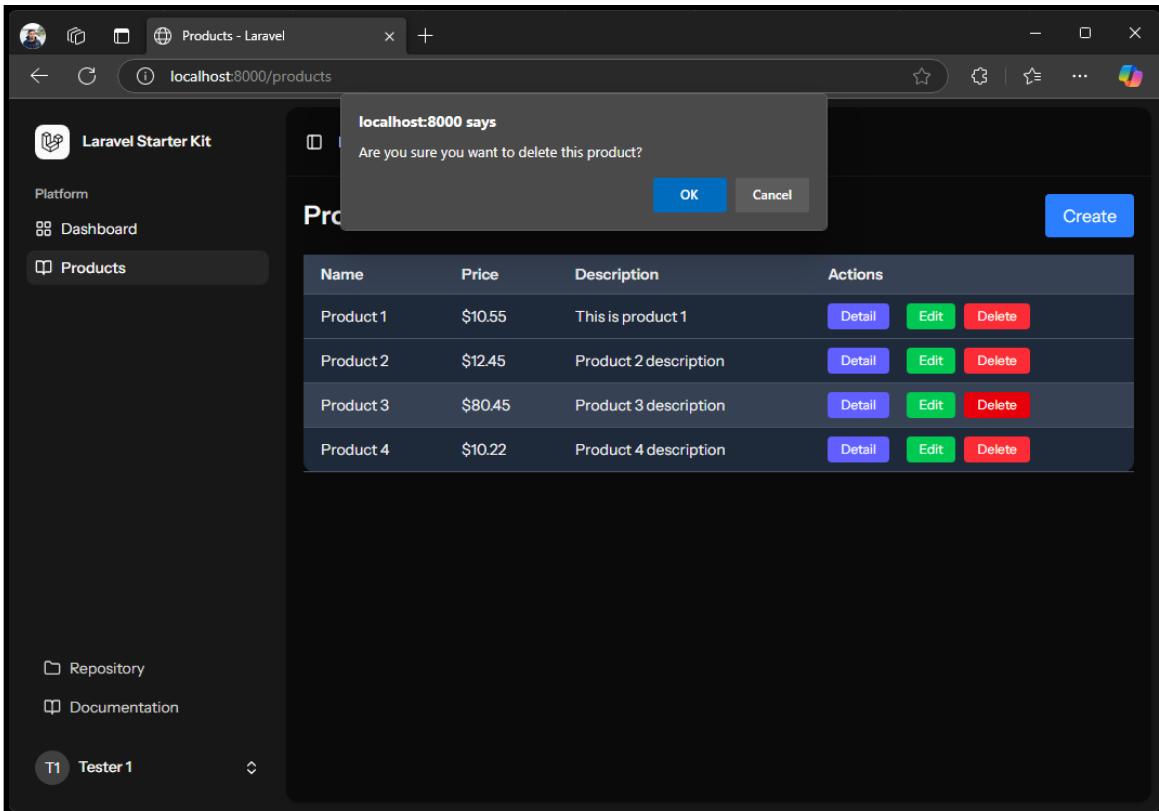


Figure 15.19: Edit product page.

After confirming, the product will be deleted, and you will be redirected to the products page.

15.7.5 Summary

In this hands-on lab, you built a complete full stack CRUD application using Laravel 12 with the Vue.js starter kit. You created a RESTful API for managing products and connected it with a modern Vue.js frontend. SQLite was used as the database for simplicity. This lab provides a solid foundation for full stack Laravel + Vue.js projects with clean architecture and API-driven interaction.

15.8 Conclusion

In this chapter, we explored how to build a full stack web application using Laravel 12 with React and Vue.js as the frontend frameworks. We created a RESTful API for managing products and connected it with modern JavaScript frameworks. This hands-on lab provided a solid foundation for building real-world applications with Laravel 12, showcasing the power of Laravel's starter kits and modern frontend technologies.

OceanofPDF.com

16 Laravel 12 Security – How to Harden the Web App

16.1 Introduction

Web application security is a fundamental aspect of modern development, especially when applications manage sensitive data or are exposed to the public internet. Laravel 12 offers a comprehensive set of built-in security features, but developers must understand and implement best practices to ensure complete protection against threats like SQL injection, XSS, CSRF, and more. This chapter guides you through practical techniques to harden your Laravel application.

16.2 Understanding Common Threats

Before implementing security measures, it's important to understand the types of threats web applications commonly face. The OWASP Top 10, <https://owasp.org/www-project-top-ten> is a globally recognized list of the most critical security risks.

Laravel helps mitigate many of these by default, but developers still play a key role in securing the application through proper usage and configuration.

16.3 Cross-Site Scripting (XSS) Protection

XSS attacks occur when attackers inject malicious scripts into web pages. Laravel protects you by automatically escaping output in Blade templates:

```
| <!-- Safe: Escaped output -->
| {{ $username }}  

| <!-- Unsafe: Raw output -->
| {!! $username !!} 
```

To further secure your application:

- Always use `{} {}` unless raw HTML is absolutely necessary.
- Sanitize user input using libraries like [HTMLPurifier](#) when displaying user-generated content.

16.4 SQL Injection Prevention

Laravel uses prepared statements and parameter binding by default, making SQL injection attacks highly unlikely when using Eloquent or the Query Builder:

```
// Safe with Eloquent
User::where('email', $email)->first();

// Safe with Query Builder
DB::table('users')->where('email', $email)->get();
```

Avoid using raw SQL queries unless absolutely necessary. If you must use `DB::raw()`, ensure all inputs are sanitized properly.

16.5 CSRF Protection

Cross-Site Request Forgery (CSRF) tricks users into performing actions without their consent. Laravel automatically protects forms with CSRF tokens:

```
<form method="POST" action="/submit">
    @csrf
    <!-- form fields -->
</form>
```

For API routes, CSRF protection is typically disabled by default since APIs are stateless. However, you should still:

- Authenticate requests via tokens (e.g., JWT, Sanctum).
- Use HTTPS to protect token transmission.

16.6 Authentication and Authorization Best Practices

Laravel provides robust authentication and authorization systems. Use Laravel Breeze, Jetstream, or Fortify to implement secure login systems

quickly.

Best practices include:

- Use hashed passwords via bcrypt or Argon2id.
- Enforce strong password rules.
- Regenerate session IDs upon login to prevent session fixation.
- Use policies and gates for fine-grained access control.
- Limit login attempts using `ThrottleRequests` middleware.

16.7 Securing Sessions and Cookies

Laravel sessions are encrypted and stored securely by default. To enhance this:

- Set `SESSION_SECURE_COOKIE=true` to transmit cookies only over HTTPS.
- Use `httpOnly` and `secure` flags in cookie settings.
- Store sessions in a database or Redis for better control and scalability.

16.8 Force HTTPS and TLS

Using HTTPS is essential to protect data in transit. Laravel allows you to enforce HTTPS:

- Add `\App\Http\Middleware\EnsureHttps::class` middleware.
- Set `APP_URL=https://yourdomain.com` in `.env`.
- Redirect HTTP traffic to HTTPS via `.htaccess` or web server config.

16.9 Secure File Uploads

File uploads are often a major attack surface. Laravel allows you to handle file uploads securely:

- Always validate file types and sizes.
- Rename uploaded files to avoid guessing.
- Store sensitive files outside `public/` directory.
- Scan uploaded files if they will be executed or viewed.

Example validation:

```
| $request->validate([
|   'file' => 'required|mimes:jpg,png,pdf|max:2048',
| ]);
```

16.10 Set Security Headers

Security headers help browsers mitigate attacks like clickjacking and MIME sniffing. Use middleware or the `bepsvpt/secure-headers` package to add:

- Content-Security-Policy
- X-Frame-Options
- X-Content-Type-Options
- Strict-Transport-Security
- Referrer-Policy

16.11 Rate Limiting and Throttling

Prevent abuse of APIs and forms using Laravel's rate limiting feature:

```
| RateLimiter::for('api', function (Request $request) {
|   return Limit::perMinute(60)->by($request->user()?->id ?: $request->ip());
| });
```

Apply it via middleware in `routes/api.php`:

```
| Route::middleware('throttle:api')->group(function () {
|   // protected routes
| });
```

16.12 Logging and Monitoring

Security also involves knowing what's happening in your system. Laravel uses Monolog and supports multiple channels like:

- Single and daily logs
- Slack or Discord notifications
- External tools (Sentry, Bugsnag, etc.)

Always log:

- Login attempts (failed/successful)
- Authorization denials
- File upload activities
- Database changes

16.13 Security Testing and Audits

Regular audits help discover vulnerabilities early. Recommended tools:

- `composer audit`: Check for known vulnerabilities in dependencies.
- OWASP ZAP, <https://owasp.org/www-project-zap>: Scan your running app.
- Static analysis tools like Larastan or PHPStan.

16.14 Conclusion

Laravel 12 provides a solid foundation for building secure applications, but security is a shared responsibility. By following best practices and leveraging Laravel's built-in features, you can significantly reduce the risk of vulnerabilities in your application. Always stay updated with the latest security trends and continuously monitor your application for potential threats.

17 Monitoring and Deployment

17.1 Introduction

Once an application is feature-complete and tested, the next phase is deployment—delivering your Laravel 12 application to the real world. Deployment is a critical aspect of modern web development, and when done right, it ensures reliability, scalability, and maintainability. In this chapter, we will explore how to monitor a Laravel 12 application, and then walk through deployment using **containers (Docker)**, which is an essential skill in modern DevOps workflows.

17.2 Monitoring Laravel Applications

Monitoring helps track performance, detect issues, and ensure the system is healthy in production. Laravel offers several tools and integrations to make monitoring seamless.

17.2.1 Laravel Telescope (Local Development Debugging)

Laravel Telescope is a powerful debugging assistant for Laravel. While it's not suitable for production, it's invaluable during development.

Features:

- Request and response logging
- Query inspection
- Job monitoring
- Exception tracking
- Mail and notification logs

Installation:

To install Telescope in a Laravel 12 application, follow to run the following commands in your terminal:

```
| composer require laravel/telescope --dev
| php artisan telescope:install
| php artisan migrate
| php artisan serve
```

Visit `/telescope` to access the dashboard.

17.2.2 Laravel Log Monitoring

Laravel writes logs to `storage/logs/laravel.log` by default. These can be monitored using:

- **Log rotation** tools like `logrotate`
- **Log forwarding** to services like ELK Stack (Elasticsearch, Logstash, Kibana)
- **Cloud log tools** like Papertrail or Datadog

17.3 Laravel 12 Deployment Best Practices

When deploying Laravel applications, following best practices ensures reliability:

- Use `.env` files for environment-specific configuration.
- Enable application caching (`php artisan config:cache`, `route:cache`, `view:cache`).
- Set `APP_ENV=production` and `APP_DEBUG=false` for production.
- Use `php artisan migrate --force` for database migrations in CI/CD.

17.4 Exercise 49: Exploring Laravel Telescope in Laravel 12

17.4.1 Description

Laravel Telescope is a powerful debugging and monitoring assistant for Laravel applications. It provides real-time insight into requests, exceptions, database queries, jobs, events, and more. In this lab, you'll integrate Laravel Telescope into a Laravel 12 app using SQLite for simplicity.

17.4.2 Objectives

By the end of this lab, you will be able to:

- Install and configure Laravel Telescope in a Laravel 12 project
- Use Telescope to monitor incoming requests, queries, logs, and exceptions
- Run and test Telescope on a local server using SQLite as the database

17.4.3 Prerequisites

- Laravel 12 project set up locally
- SQLite installed and available (usually pre-installed in most environments)
- PHP 8.1+ and Composer installed
- Basic knowledge of Laravel routing and artisan commands

*This lab uses **PHP 8.4**, **Laravel 12** and **Visual Studio Code** as the primary editor.*

17.4.4 Steps

Here's a step-by-step guide to setting up Laravel Telescope in a Laravel 12 application using SQLite.

17.4.4.1 Step 1: Prepare Laravel Project

If you haven't created a Laravel 12 project, run:

```
| laravel new telescope-demo  
| cd telescope-demo  
| code .
```

Accept all the default options during installation. This will create a new Laravel project named `telescope-demo` and use SQLite as the default database. The last command will open the project in Visual Studio Code.

You should see the project in Visual Studio Code.

17.4.4.2 Step 2: Configure `.env` to Use SQLite

Since we use default SQLite database, we don't do anything. We can verify this by checking the `.env` file. Open the `.env` file and check the following lines:

```
| DB_CONNECTION=mysql
```

17.4.4.3 Step 3: Install Laravel Telescope

Use Composer to install Telescope (dev only):

```
| composer require laravel/telescope --dev
```

Publish Telescope's assets and configuration:

```
| php artisan telescope:install
```

Run database migrations:

```
| php artisan migrate
```

This will create the necessary tables in the SQLite database.

17.4.4.4 Step 4: Serve the Application

Save all changes and run the Laravel server. Then open a new terminal window and navigate to the project directory:

Run the Laravel server:

```
| php artisan serve
```

Visit <http://127.0.0.1:8000/telescope> to access the Telescope dashboard.

By default, Telescope is only accessible in the `local` environment.

You can see the Telescope dashboard, which provides a real-time view of your application's requests, logs, queries, and more.

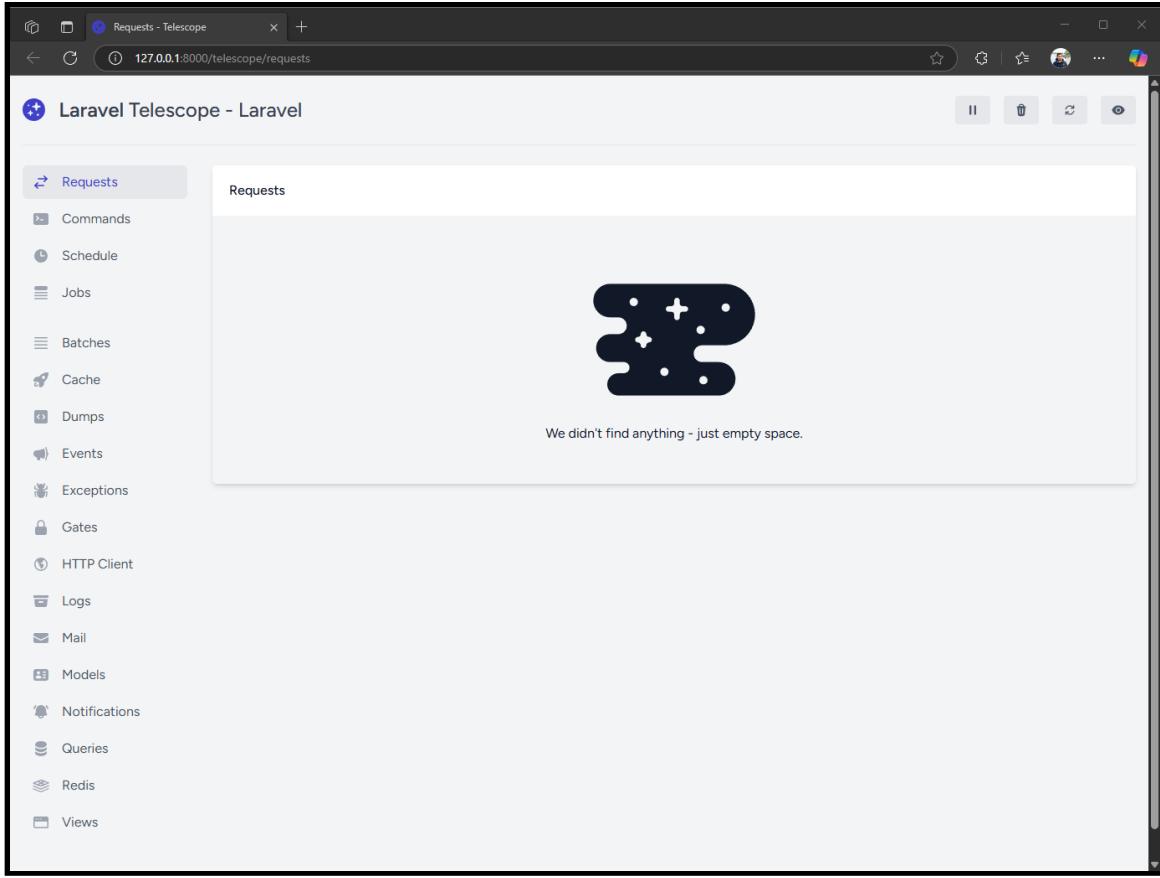


Figure 17.1: Telescope dashboard.

17.4.4.5 Step 5: Try a Simple Route to Generate Telescope Logs

Add a route in `routes/web.php`:

```
use Illuminate\Support\Facades\Log;

Route::get('/hello', function () {
    Log::info('Visited /hello route');
    return 'Hello from Laravel Telescope!';
});
```

Open new tab in your browser and Visit <http://127.0.0.1:8000/hello> in your browser.

Then go back to [Telescope](#) and check the following:

- **Requests** tab → See the `/hello` route logged
- **Logs** tab → See the `info` log you wrote
- **Queries** tab → Observe any database activity (if applicable)

Make hits to the `/hello` route multiple times to see how Telescope tracks requests and logs. You also hit the `/` route to see the Laravel welcome page.

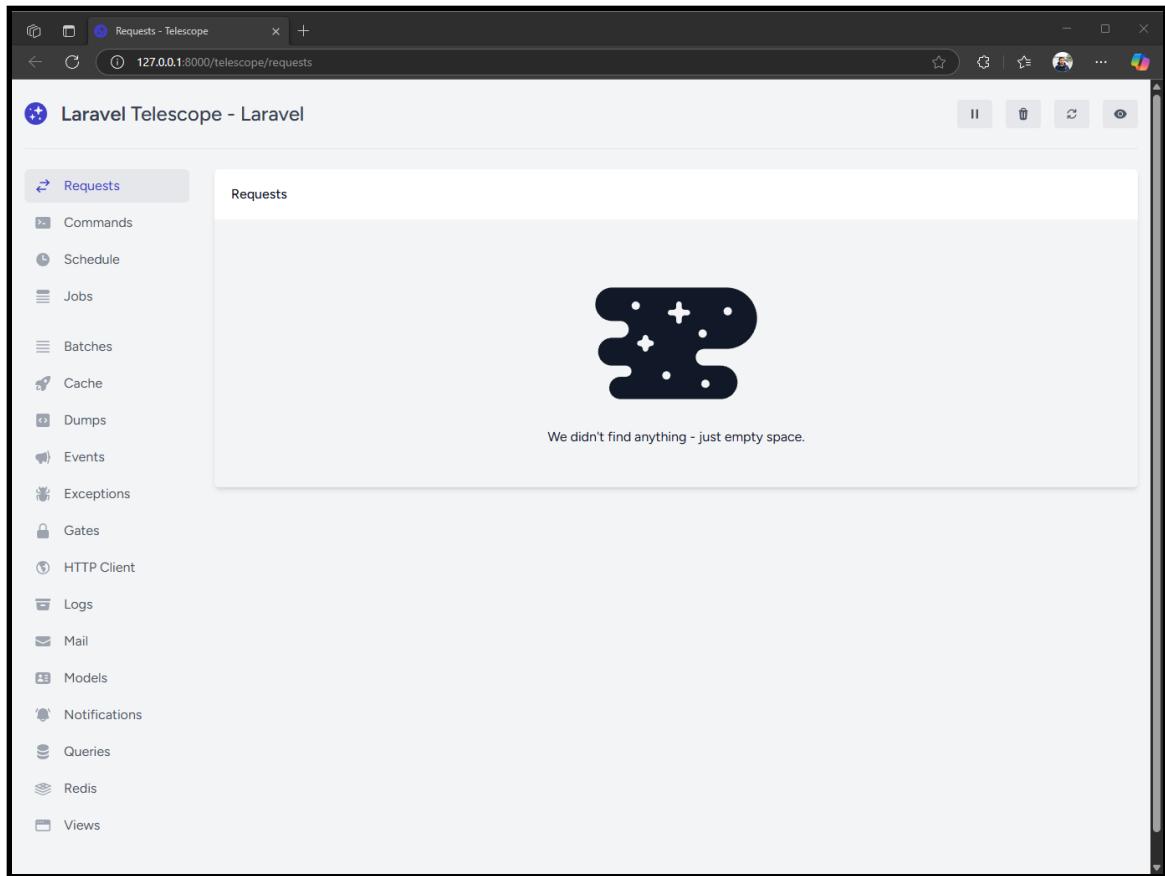


Figure 17.2: TRacing requests on Telescope dashboard.

17.4.4.6 Step 6: (Optional) Restrict Telescope Access in Production

Telescope is intended for development, but if you want to customize access control, open:

```
// App\Providers\TelescopeServiceProvider.php
protected function gate()
{
    Gate::define('viewTelescope', function ($user) {
        return in_array($user->email, [
            'your@email.com',
        ]);
    });
}
```

17.4.5 Summary

In this lab, you learned how to set up **Laravel Telescope** in a Laravel 12 application using **SQLite**. Telescope is a great tool to debug and profile your Laravel applications during development. You've explored how it logs requests, responses, queries, and custom logs in real-time.

This setup helps identify performance bottlenecks and application errors early in the development cycle.

17.5 Laravel Deployment

Laravel applications can be deployed in various environments, including shared hosting, virtual private servers (VPS), and cloud platforms. The deployment process typically involves:

- Setting up the server environment (PHP, web server, database)
- Configuring the web server (Nginx, Apache)
- Deploying the application code
- Setting up environment variables
- Running database migrations
- Configuring caching and queues
- Setting up SSL certificates for HTTPS (optional)
- Monitoring and logging
- Backups and disaster recovery plans

17.6 Containerizing Laravel 12 with Docker

Containers provide a lightweight, portable, and consistent environment for deploying Laravel applications. In this section, we'll cover how to containerize and deploy a Laravel 12 application using Docker.

17.6.1 Why Use Docker?

Docker simplifies the deployment process by allowing you to package your application and its dependencies into a single container. This ensures that your application runs consistently across different environments, from development to production.

Docker provides several benefits:

- Consistent environment from development to production

- Isolation from host machine
- Easy scaling and orchestration with Kubernetes or Docker Compose
- Fast and reproducible deployments

17.6.2 Installing Docker

To install Docker on your local machine or server, follow the official Docker installation guide for your operating system. You can find the instructions here: Docker Installation Guide, <https://docs.docker.com/engine/install/>.

17.7 Exercise 50: Deploying Laravel 12 to Production on Ubuntu Server with PHP 8.4

17.7.1 Description

In this lab, we will deploy a Laravel 12 application on a production-ready environment using a virtual machine running **Ubuntu Server** and **PHP 8.4.x**. You'll configure Nginx, PHP-FPM, MySQL, and deploy Laravel from a GitHub repository.

17.7.2 Objectives

By the end of this lab, you will be able to:

- Set up a production server with PHP 8.4 for Laravel 12
- Install and configure Nginx, PHP-FPM, Composer, and MySQL
- Deploy and configure Laravel application securely on Ubuntu

17.7.3 Prerequisites

- Laravel 12 project (GitHub or ZIP)
- A VM running **Ubuntu Server 22.04 or later**
- SSH access to the server
- (Optional) Domain name pointing to the server

17.7.4 Steps

Here's a step-by-step guide to deploying Laravel 12 on Ubuntu Server with PHP 8.4.

17.7.4.1 Step 1: Add PHP 8.4 Repository and Install Dependencies

We will use the **ondrej/php** PPA to install PHP 8.4 and its extensions.

```
| sudo apt update && sudo apt upgrade -y  
| sudo add-apt-repository ppa:ondrej/php -y  
| sudo apt update  
| sudo apt install nginx php8.4 php8.4-fpm php8.4-mbstring php8.4-xml php8.4-bcmath php8.4-cgi  
|
```

This installs Nginx, PHP 8.4, and the necessary PHP extensions for Laravel. After installation, check the PHP version to confirm:

```
| php -v
```

17.7.4.2 Step 2: Install Composer

Composer is a dependency manager for PHP, essential for Laravel. Install it globally:

```
| cd ~  
| curl -sS https://getcomposer.org/installer | php  
| sudo mv composer.phar /usr/local/bin/composer
```

Make sure Composer is installed correctly:

```
| composer --version
```

You should see the Composer version output.

17.7.4.3 Step 3: Clone Laravel 12 Application

We will clone a sample Laravel 12 application from GitHub. Replace `your-repo` with your actual repository.

```
| cd /var/www  
| sudo git clone https://github.com/your-repo/laravel12-app.git laravel12  
| cd laravel12  
| sudo chown -R www-data:www-data .
```

If your git repository is private, you may need to set up SSH keys or use HTTPS with credentials.

Another option is to upload a ZIP file of your Laravel project and extract it in the `/var/www` directory.

17.7.4.4 Step 4: Configure Laravel Environment

Copy the example environment file and install dependencies:

```
| cp .env.example .env  
| composer install --optimize-autoloader --no-dev  
| php artisan key:generate  
| php artisan config:cache  
| php artisan route:cache  
| php artisan view:cache
```

Edit the `.env` file to set up your database connection and other environment variables.

17.7.4.5 Step 5: Set Up Database

Depending on what database you are using, you can install MySQL, MariaDB, PostgreSQL or SQLite. Make sure to install database server and client. In addition, you should install the PHP extensions for the database you are using.

After configuring the database, you can perform database migrations. Run the migrations to set up the database schema:

```
| php artisan migrate --force
```

You may not perform database migrations if you already restored the database from a backup.

17.7.4.6 Step 6: Configure Nginx for Laravel

We will set up Nginx to serve the Laravel application. First, create a new Nginx configuration file:

Create Nginx config:

```
| sudo nano /etc/nginx/sites-available/laravel12
```

Add the following:

```
| server {  
|   listen 80;
```

```
server_name yourdomain.com; # Replace with your domain or IP
root /var/www/laravel12/public;

index index.php index.html;

location / {
    try_files $uri $uri/ /index.php?$query_string;
}

location ~ \.php$ {
    include snippets/fastcgi-php.conf;
    fastcgi_pass unix:/run/php/php8.4-fpm.sock;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    include fastcgi_params;
}

location ~ /\.ht {
    deny all;
}
}
```

Enable and reload:

```
sudo ln -s /etc/nginx/sites-available/laravel12 /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl reload nginx
```

17.7.4.7 Step 7: Set Permissions

After we copied the files, we need to set the correct permissions for the storage and bootstrap/cache directories:

```
sudo chown -R www-data:www-data /var/www/laravel12
sudo chmod -R 755 /var/www/laravel12/storage /var/www/laravel12/bootstrap/cache
```

Change `laravel12` to your actual project directory. The `www-data` user is the default user for Nginx and PHP-FPM on Ubuntu.

This ensures that the web server can write to these directories.

17.7.4.8 Step 8: Configure Firewall (Optional)

To allow HTTP and HTTPS traffic, you can use UFW (Uncomplicated Firewall):

```
sudo ufw allow OpenSSH
sudo ufw allow 'Nginx Full'
sudo ufw enable
```

If you are using a cloud provider, make sure to allow HTTP and HTTPS traffic in the security group settings.

17.7.4.9 Step 9: Add HTTPS with Let's Encrypt (Optional)

To secure your application with HTTPS, you can use **Certbot** to obtain a free SSL certificate from Let's Encrypt.

You can install Certbot and the Nginx plugin with the following commands:

```
| sudo apt install certbot python3-certbot-nginx -y  
| sudo certbot --nginx -d yourdomain.com
```

Change `yourdomain.com` to your actual domain. Follow the prompts to set up SSL.

Certbot will automatically configure Nginx to use the SSL certificate.

17.7.5 Summary

In this lab, you've deployed a Laravel 12 app using **PHP 8.4** on an **Ubuntu Server VM** with Nginx and MySQL. This step-by-step process ensures a secure and production-ready Laravel deployment with the latest stable PHP features.

17.8 Exercise 51: Deploying Laravel 12 to Production in Docker on Ubuntu Server

17.8.1 Description

In this lab, you'll learn how to deploy a Laravel 12 application inside Docker containers on an Ubuntu Server. The application will run inside containers, but the MySQL database will remain on the host machine (outside of Docker). This hybrid setup is common in many production environments where databases are managed separately from app runtimes.

17.8.2 Objectives

By the end of this lab, you will be able to:

- Install and configure the latest Docker and Docker Compose on Ubuntu Server
- Create a containerized environment for Laravel 12 using PHP-FPM and Nginx

- Connect Laravel containers to a MySQL database running on the host machine
- Run the Laravel 12 application in a production container setup

17.8.3 Prerequisites

- Laravel 12 project (can be pushed to GitHub or zipped)
- Ubuntu Server 22.04+ with root or sudo access
- MySQL installed and running on the host machine (e.g., `localhost:3306`)
- Database created for Laravel (e.g., `laravel12_db`), with user and password
- Port `80` open and accessible

17.8.4 Steps

Here's a step-by-step guide to deploying Laravel 12 in Docker on Ubuntu Server.

17.8.4.1 Step 1: Install Docker and Docker Compose (Latest Version)

This installation method based on the official Docker documentation is the recommended way to install Docker Engine on Ubuntu. It ensures you get the latest version of Docker and its dependencies. Ref:
<https://docs.docker.com/engine/install/ubuntu/>.

Before you install Docker Engine for the first time on a new host machine, you need to set up the Docker apt repository. Afterward, you can install and update Docker from the repository.

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://$(
$(. /etc/os-release && echo "${UBUNTU_CODENAME}-$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

To install the latest version, run:

```
| sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-com
```

To test the installation, run:

```
| sudo docker run hello-world
```

This command downloads a test image and runs it in a container. If everything is set up correctly, you should see a message saying “Hello from Docker!”

`docker` command may require `sudo` if you haven’t added your user to the Docker group. To avoid using `sudo`, add your user to the Docker group:

Based on the official Docker documentation, you can read this here:
<https://docs.docker.com/engine/install/linux-postinstall/>.

17.8.4.2 Step 2: Clone Your Laravel 12 Project

Now we will clone the Laravel 12 project from GitHub. If you have a ZIP file, you can upload it to the server and extract it.

```
| cd /var/www
| sudo git clone https://github.com/your-org/laravel12-app.git laravel12
| cd laravel12
| sudo chown -R $USER:$USER .
```

17.8.4.3 Step 3: Create Laravel Dockerfile

Create a `Dockerfile` inside the Laravel project root:

```
# Dockerfile
FROM php:8.3-fpm

# Install PHP extensions and dependencies
RUN apt-get update && apt-get install -y \
    libpng-dev libjpeg-dev libonig-dev libxml2-dev zip unzip curl git \
    && docker-php-ext-install pdo pdo_mysql mbstring exif pcntl bcmath gd

# Set working directory
WORKDIR /var/www

# Copy files
COPY . .

# Install Composer
COPY --from=composer:latest /usr/bin/composer /usr/bin/composer
RUN composer install --optimize-autoloader --no-dev

# Set file permissions
RUN chown -R www-data:www-data /var/www
```

17.8.4.4 Step 4: Create Docker Compose File

Create a `docker-compose.yml`:

```
version: '3.8'

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: laravel12-app
    restart: unless-stopped
    working_dir: /var/www
    volumes:
      - .:/var/www
    environment:
      - APP_ENV=production
      - DB_CONNECTION=mysql
      - DB_HOST=host.docker.internal
      - DB_PORT=3306
      - DB_DATABASE=laravel12_db
      - DB_USERNAME=laraveluser
      - DB_PASSWORD=securepass
    depends_on:
      - web

  web:
    image: nginx:alpine
    container_name: laravel12-web
    restart: unless-stopped
    ports:
      - "80:80"
    volumes:
      - .:/var/www
      - ./nginx.conf:/etc/nginx/conf.d/default.conf
```

17.8.4.5 Step 5: Create Nginx Configuration File

Create `nginx.conf` in the root of the Laravel project:

```
server {
  listen 80;
  server_name localhost;
  root /var/www/public;

  index index.php index.html;

  location / {
    try_files $uri $uri/ /index.php?$query_string;
  }

  location ~ \.php$ {
    include fastcgi_params;
    fastcgi_pass app:9000;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    fastcgi_index index.php;
  }
}
```

```
|     location ~ /\.ht {  
|         deny all;  
|     }  
| }
```

17.8.4.6 Step 6: Set Environment and Permissions

```
cp .env.example .env  
  
# Update .env for MySQL connection  
nano .env  
# Set:  
# DB_HOST=host.docker.internal  
# DB_PORT=3306  
# DB_DATABASE=laravel12_db  
# DB_USERNAME=laraveluser  
# DB_PASSWORD=securepass  
  
# Ensure correct permissions  
sudo chmod -R 755 storage bootstrap/cache
```

17.8.4.7 Step 7: Build and Run Containers

```
| docker compose up -d --build
```

Access your app via: <http://your-server-ip>

17.8.4.8 Step 8: Run Laravel Setup Commands Inside the Container

```
docker exec -it laravel12-app bash  
  
# Inside container shell  
php artisan key:generate  
php artisan config:cache  
php artisan migrate --force  
exit
```

17.8.5 Summary

In this lab, you've successfully deployed a **Laravel 12 application** in a **Dockerized production environment** on Ubuntu Server, while using a **MySQL database running on the host machine**. You've built containers for PHP-FPM and Nginx, configured Laravel, and accessed the app through your server IP.

This setup provides flexibility, isolation, and a solid foundation for scalable Laravel deployments in real-world environments.

17.9 Conclusion

In this chapter, we explored the importance of monitoring and deployment in Laravel applications. We learned how to monitor Laravel applications using tools like **Laravel Telescope** and **log monitoring**. We also covered best practices for deploying Laravel applications, including setting up a production environment with PHP 8.4, Nginx, and MySQL.

Finally, we delved into containerization using Docker, which simplifies the deployment process and ensures consistency across different environments. By following these practices, you can ensure that your Laravel applications are secure, reliable, and ready for production.

OceanofPDF.com

Appendix A: PHP Cheat Sheet

This cheat sheet provides a quick reference to the most commonly used PHP syntax and functions. It is designed for beginners and intermediate developers who want to refresh their knowledge or learn new concepts.

1. Basic Syntax

```
<?php  
echo "Hello, World!";  
?>
```

2. Variables and Data Types

```
$name = "Agus";           // String  
$age = 25;                 // Integer  
$price = 12.5;             // Float  
$is_active = true;          // Boolean
```

3. Control Structures

- If / Else

```
if ($age >= 18) {  
    echo "Adult";  
} else {  
    echo "Minor";  
}
```

- Switch

```
$day = "Monday";  
switch ($day) {  
    case "Monday":  
        echo "Start of week";  
        break;  
    default:  
        echo "Other day";  
}
```

- For Loop

```
for ($i = 0; $i < 5; $i++) {  
    echo $i;
```

```
| }
```

- While Loop

```
| $i = 0;  
| while ($i < 5) {  
|     echo $i++;  
| }
```

4. Functions

```
| function greet($name) {  
|     return "Hello, $name!";  
| }  
| echo greet("Agus");
```

5. Arrays

- Indexed Array

```
| $fruits = ["Apple", "Banana", "Cherry"];  
| echo $fruits[1]; // Banana
```

- Associative Array

```
| $person = ["name" => "Agus", "age" => 30];  
| echo $person["name"];
```

- Foreach Loop

```
| foreach ($fruits as $fruit) {  
|     echo $fruit . "\n";  
| }
```

6. String Functions

```
| $str = "Hello World";  
| echo strlen($str); // 11  
| echo str_replace("World", "PHP", $str); // Hello PHP
```

7. File Handling

```
| // Write to file  
| file_put_contents("file.txt", "Hello File");  
  
| // Read from file  
| $content = file_get_contents("file.txt");  
| echo $content;
```

8. Forms and \$_POST

```
// HTML
<form method="post">
    <input name="name" />
    <input type="submit" />
</form>

// PHP
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    echo $_POST['name'];
}
```

9. Include and Require

```
include 'header.php';
require 'config.php';
```

10. Sessions

```
session_start();
$_SESSION["user"] = "Agus";
echo $_SESSION["user"];
```

11. Cookies

```
setcookie("user", "Agus", time() + 3600);
echo $_COOKIE["user"];
```

12. MySQLi (MySQL)

```
$conn = new mysqli("localhost", "user", "pass", "db");
$sql = "SELECT * FROM users";
$result = $conn->query($sql);
while ($row = $result->fetch_assoc()) {
    echo $row['name'];
}
$conn->close();
```

13. Error Handling

```
try {
    throw new Exception("Error occurred");
} catch (Exception $e) {
    echo "Caught: " . $e->getMessage();
}
```

14. Sanitize Input

```
| $name = htmlspecialchars($_POST["name"]);
```

OceanofPDF.com

Appendix B: Resources

Enhance Your Learning with Our Udemy Course

For those who've journeyed with us through this book, we have something special to further your understanding — a comprehensive Udemy course titled **“Red Hat NGINX Web Server: Publishing and Deploying Web Apps.”**

Why Choose This Course?

1. **Specialized Knowledge:** Dive deep into the world of Red Hat and NGINX. Understand how to use NGINX on the Red Hat platform, a powerful combination for web server deployments.
2. **Hands-On Approach:** Our course isn't just about theory; we believe in the 'learn by doing' philosophy. With guided tutorials and real-world examples, grasp how to publish and deploy various web applications effectively.
3. **Expert Instructors:** Benefit from the insights and expertise of professionals who are not just educators but industry practitioners with years of experience.
4. **Flexible Learning:** Learn at your own pace. With lifetime access, you can revisit topics anytime and solidify your understanding.

Who Is This Course For? - Web developers looking to understand the deployment process on Red Hat using NGINX. - System administrators aiming to expand their knowledge in server configuration and optimization. - IT professionals transitioning to roles that require knowledge of web server setup and deployment on Red Hat.

Enroll today: *Red Hat NGINX Web Server: Publishing and Deploying Web Apps* <https://www.udemy.com/course/rhel-nginx/?referralCode=C9CFA39AE9E332ADA9FB>

Build Secure PHP APIs Like a Pro with Laravel 12, OAuth2, and JWT

Unlock the full potential of **Laravel 12** for REST API development! This hands-on course on Udemy teaches you how to build **robust, secure, and modern APIs** using Laravel, MySQL, OAuth2, JWT, Sanctum, and Role-Based Access Control (RBAC). Perfect for real-world applications and 2025 standards.

Highlight Topics

- What's New in Laravel 12 for API development
- Build RESTful APIs from scratch (Hello World to full CRUD)
- File upload and user data handling via REST API
- Secure authentication with **Sanctum, JWT, and OAuth2**
- Role-Based Access Control (RBAC) with middleware
- Legacy support: Laravel 8, 7.x, and 6.x projects included
- Real project codebases and testing tutorials

Who Should Enroll?

- Laravel developers aiming to modernize their API skills
- Backend engineers securing APIs with token-based auth
- Teams migrating legacy Laravel APIs to newer standards
- Students and professionals building real-world Laravel apps
- Anyone preparing for backend development roles in 2025

Future-proof your Laravel skills. This course gives you **everything you need** to build secure, scalable, and professional REST APIs in Laravel 12. Learn by doing — with real code, live tests, and full project coverage.

Join now and start building APIs that meet today's security demands.
PHP REST API: Laravel 12, MySQL, OAuth2, JWT, Roles-Based
<https://www.udemy.com/course/phprestapi/?referralCode=2C5B2F14100B499E9845>

Master Real-World Logging & Visualization with the Full ELK Stack

Take control of your logging, search, and monitoring pipeline with this **hands-on Udemy course** covering Elasticsearch, Logstash, Kibana, and Beats. Learn how to set up, ingest, visualize, and scale log data using practical projects — all designed for developers, sysadmins, and DevOps engineers in **real production environments**.

Highlight Topics

- Cross-platform installation: Windows, Ubuntu, macOS, Docker
- Elasticsearch REST API: CRUD, mapping, queries, aggregation, SQL, geo fields
- Real-world API integration: PHP, ASP.NET Core, Node.js, Python
- Logstash ingestion: files, folders, and RDBMS (MySQL)
- Kibana Lens visualizations: charts, maps, dashboards, Canvas
- Beats agents: Filebeat, Winlogbeat, Metricbeat, Packetbeat, Heartbeat, Auditbeat
- High Availability (HA) setup for Elasticsearch and Kibana with Nginx

Who Should Enroll?

- Developers and DevOps engineers building log-driven applications
- System administrators responsible for monitoring and observability
- Backend/API developers seeking integration with Elasticsearch
- Cybersecurity analysts and IT ops engineers using ELK for log auditing
- Teams adopting open-source observability tools for modern infrastructure

Log smarter, visualize better, and scale with confidence. Whether you're just getting started or already managing production systems, this course gives you everything you need to build and operate a **powerful ELK Stack**

pipeline. With real-world use cases, cross-platform setups, and step-by-step guidance, you'll go beyond the basics and into expert territory.

Enroll today to master the ELK Stack and unlock actionable insights from your data! *Practical Full ELK Stack: Elasticsearch, Kibana and Logstash* <https://www.udemy.com/course/elkstack/?referralCode=863C1036F77169C975C5>

OceanofPDF.com

Appendix C: Source Code

You can download the source code files for this book from GitHub at
<https://www.github.com/aguskilmudata-book-laravel12>.

OceanofPDF.com

About

Agus Kurniawan's journey in the field of technology, spanning from 2001, is a remarkable blend of deep technical expertise and a fervent passion for sharing knowledge. As a seasoned professional, Agus has carved a niche in diverse technological domains, including software development, IoT (Internet of Things), Machine Learning, IT infrastructure, and DevOps. His experiences are not just limited to developing cutting-edge solutions but also extend to shaping the future of upcoming technologists through training and workshops.

Agus's career is marked by significant contributions to both technological innovation and community development. His recognition as a Microsoft Most Valuable Professional (MVP) from 2004 to 2022 underlines his proficiency in Microsoft technologies and his dedication to educating others. Agus has been at the forefront of delivering various training sessions and workshops, sharing his insights and helping others grow in the ever-evolving tech industry.

Agus Kurniawan's book, **Laravel 12 Training Kit: A Practical Guide to Modern Web Development**, serves as a thorough resource for developers aiming to master modern web application development with Laravel 12. Drawing from his extensive background in software engineering, Agus presents practical techniques, real-world examples, and best practices to help readers build robust and scalable applications using the latest features of the Laravel framework.

Contact the Author

Agus Kurniawan appreciates feedback, questions, and suggestions from readers. If you have inquiries about Laravel, ideas for future editions, or wish to share your learning experiences, he encourages you to get in touch.

If you are interested in private training or class-based sessions on Laravel, web development, or related technologies, he also offers tailored training

programs for individuals and organizations. Please reach out for more information about available topics, schedules, and formats.

Email: aguskur@hotmail.com, agusk2007@gmail.com

LinkedIn: linkedin.com/in/agusk

Twitter: [@agusk2010](https://twitter.com/@agusk2010)

OceanofPDF.com