Community Experience Distilled

Laravel Design Patterns and Best Practices

Enhance the quality of your web applications by efficiently implementing design patterns in Laravel

Arda Kılıçdağı
H. İbrahim YILMAZ

[PACKT] PUBLISHING open source* community experience distilled

Laravel Design Patterns and Best Practices

Enhance the quality of your web applications by efficiently implementing design patterns in Laravel

Arda Kılıçdağı

H. İbrahim YILMAZ

[PACKT] PUBLISHING

open source community experience distilled

BIRMINGHAM - MUMBAI

Laravel Design Patterns and Best Practices

First published: July 2014

Production reference: 1180714

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-798-7

www.packtpub.com

Cover image by Abhinav Pandey (abhinavphotography30@gmail.com)

Credits

Authors
Arda Kılıçdağı
H. İbrahim YILMAZ

Reviewers
Fabio Alessandro Locati
Brayan Rastelli
Pavel Tkachenko

Commissioning Editor
Taron Pereira

Acquisition Editor
Meeta Rajani

Content Development Editor
Neeshma Ramakrishnan

Technical Editor
Shashank Desai

Copy Editors
Insiya Morbiwala
Sayanee Mukherjee
Aditya Nair

Project Coordinators
Danuta Jones
Harshal Ved

Proofreader
Maria Gould

Indexer
Tejal Soni

Graphics
Valentina D'silva
Yuvraj Mannari

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Authors

Arda Kılıçdağı is a PHP/MySQL/JavaScript/Linux programmer and server administrator from Turkey. He has been developing applications with PHP since 2005. He administers the Turkish national support site of the world famous open source content management script, PHP-Fusion. He's also one of the international developers and a member of the management team of CMS, and he plays an important role in CMS's future. He has worked as a developer and has experience on projects such as Begendy (an exclusive private shopping website) and Futbolkurdu (a local soccer news website). He has experience working with the Facebook API, the Twitter API, and PayPal's Adaptive Payment API (used on crowdfunding websites such as KickStarter). He's also experienced with JavaScript, and he's infusing his applications with JavaScript and jQuery, both on frontend and backend sides.

He has developed applications using CodeIgniter and CakePHP for about 4 years, but these PHP frameworks didn't suit his needs completely, and that's why he decided to use another framework for his projects. After getting introduced to Laravel, he has developed all his applications with it.

He's also interested in Unix and Linux, and he uses Linux on a daily basis. He's administering the world's best-known microcomputer, Raspberry Pi's biggest Turkish community website, Raspberry Pi Türkiye Topluluğu (Raspberry Pi Turkish Community Website).

Before authoring this book, Arda has written two other books. The first book is *Laravel Application Development Blueprints*, *Packt Publishing*, coauthored by H. İbrahim YILMAZ. The second book, *Raspberry Pi*, *Dikeyeksen Consulting & Publishing*, is written in Turkish.

H. İbrahim YILMAZ is a daddy, developer, geek, and an e-commerce consultant from Turkey. After his education at Münster University, Germany, he worked as a developer and software coordinator in over a dozen ventures. During this period, he developed the usage of APIs such as Google, YouTube, Facebook, Twitter, Grooveshark, and PayPal.

Currently, he's focused on creating his company about concurrent computing, Big Data, and game programming. He writes articles on Erlang, Riak, functional programming, and Big Data on his personal blog at `http://blog.drlinux.org`. He is a big Pink Floyd fan, playing bass guitar is his hobby, and he writes poems at `http://okyan.us`.

He has a daughter called İklim. He lives in a house full of Linux boxes in Istanbul, Turkey.

I'd like to thank my daughter İklim and my family for their presence. I'd also like to thank the Gezi Park protesters for their cause to make the world a better place.

I'd like to dedicate this book to Berkin Elvan. Berkin was a 15-year-old boy who was hit on the head by a teargas canister fired by a police officer in Istanbul, while out to buy bread for his family during the June 2013 antigovernment protests in Turkey. He died on March 11, 2014, following a 269-day coma.

About the Reviewers

Fabio Alessandro Locati is an Italian IT external consultant. His main areas of expertise are Linux, networking, security, data centers, and web applications. With more than 10 years of working experience in the field, he has experienced different IT roles, technologies, and languages. Fabio has worked in many different companies, starting from a single-man company to huge companies like Tech Data and Samsung. This has allowed him to consider various technologies from different points of view, helping him develop critical thinking and understand whether a technology is the correct one, in a very short span of time. Since he is always looking for better technologies, he tries new technologies to see their advantages over the old ones. For web development, he often uses PHP with Laravel due to its power and simplicity, ever since he discovered it in the first months of 2012. Fabio has used Laravel for public websites and intranet applications.

I'd like to thank my father who introduced me to computer science even before I could write, and also thank my whole family who has always been supportive.

Brayan Rastelli is involved in web development for more than 5 years now, and he is in constant pursuit of new technologies to work with. Brayan has a passion to make things faster and more efficient. He carries with him an extensive knowledge of PHP, and most notably of the Laravel Framework, having recently created a Laravel course to train Brazilians. In addition, Brayan has also created and maintained both the website and forum for the Laravel community in Brazil in order to try to help them propel and support the knowledge base both nationally and worldwide.

Currently, he works at Speed-to-Contact (`SpeedToContact.com`) on a single page/real-time application using Laravel, AngularJS, WebSockets, telephony, and other cutting-edge proprietary technologies. Brayan's Twitter handle is `@heybrayan`.

Pavel Tkachenko is an inspired self-taught computer wizard. Since childhood, he has had a passion for designing and developing websites, reverse engineering applications, file formats, and APIs. In both areas, he has created a number of original tools such as HTMLki, Sqobot, Lightpath, and ApiHook to tackle many complex computer problems. He is also the founder of the Russian Laravel community (`Laravel.ru`) and an active member of Russian publication networks such as the collaborative blog Habrahabr (`Habrahabr.ru`).

He has been freelancing since 2009, working on e-commerce, entertainment, travel, and all other types of websites built around PHP, JavaScript, and MySQL. Since then, and with over a decade of development experience, he has gathered his own team to create even more challenging and high-quality applications for companies all over the world, with high standards and great support. You can reach Pavel via his page at `http://proger.me`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

This book covers how to develop different applications and solve recurring problems using Laravel 4 design patterns. It will walk you through the widely used design patterns—the Builder (Manager) pattern, the Factory pattern, the Repository pattern, and the Strategy pattern—and will empower you to use these patterns while developing various applications with Laravel. This book will help you find stable and acceptable solutions, thereby improving the quality of your applications.

Throughout the course of the book, you will be introduced to a number of clear, practical examples about PHP design patterns and their usage in various projects. You will also get acquainted with the best practices for Laravel, which will greatly reduce the probability of introducing errors into your web applications.

By the end of this book, you will be accustomed with Laravel best practices and the important design patterns to make a great website.

What this book covers

Chapter 1, *Design and Architectural Pattern Fundamentals*, explains design and architectural pattern terms and explains the classification of these design patterns and their elements. This chapter provides some examples from the Laravel core code, which contains the design patterns used in the framework. At end of this chapter, the Mode-View-Controller (MVC) architectural pattern and its benefits will be explained.

Chapter 2, *Models in MVC*, covers the function of the Model layer in the MVC architectural pattern, its structure, its purpose, its role in the SOLID design pattern, how Laravel uses it, and the advantages of Laravel's Model layers and Eloquent ORM. Laravel classes that handle data are also discussed.

Chapter 3, *Views in MVC*, covers the function of the View layer in the MVC architectural pattern, its structure, its purpose, and the advantages of Laravel's View layer and Blade template engine. The role of View in the MVC pattern and Laravel's approach to that is also covered.

Chapter 4, *Controllers in MVC*, covers the function of the Controller layer in the MVC architectural pattern, its structure, its purpose, and its usage in Laravel's structure.

Chapter 5, *Design Patterns in Laravel*, discusses the design patterns used in Laravel. We will also see how and why they are used, with examples.

Chapter 6, *Best Practices in Laravel*, will cover basic and advanced practices in Laravel, examples of design patterns used in Laravel that we were described in previous chapters, and the reasons these patterns are used.

What you need for this book

The applications written in these chapters are all based on Laravel v4, so you will require what's listed on Laravel v4's standard requirements list, which is available at `http://laravel.com/docs/installation`. The requirements are as follows:

- PHP v5.4 or higher
- The MCrypt PHP extension

Who this book is for

This book is intended for web application developers working with Laravel who want to increase the efficiency of their web applications. It assumes that you have some experience with the Laravel PHP framework and are familiar with coding OOP methods.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `where()` method filters the query with given parameters."

A block of code is set as follows:

```
$users = DB::table('users')->get();
foreach ($users as $user)
{
    var_dump($user->name);
}
```

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Design and Architectural Pattern Fundamentals

Programming is actually a lifestyle rather than a job. It's an intense mental activity. The best developers in the world think about their work 24/7. They come up with their best ideas when they are not at their work desks. Generally, their finest work is done away from the keyboard.

As developers need to look at problems from a different standpoint, software projects cannot be accelerated by spending more time in the office or adding more people to a project. Development is not just about timelines and assigned tasks. If you visit the development centers of world-famous software companies such as Google and IBM, you'll see that there are many opportunities for spending time away from the keyboard for developers. Programming questions have to be thought of in the context of the real world. So, object-oriented programming was invented to make writing software more instinctive for our hunter-gatherer brains; that is, software components took on the properties and behavior of objects in the real world. When looking for a solution to a problem or a way to accomplish what we want, we generally hope to find a way that is reusable, optimized, and cheap. We, as developers, have a few standard ways of approaching some commonly recurring problems in programming, which are called **design patterns**.

When you come across certain problems that recur, you try and find solutions to solve them that can be used by anyone and everyone. This concept is prevalent everywhere – mechanics, architecture, and even human behavior for that matter. Programming is absolutely not an exception.

Programming solutions depend on the needs of the problems and are modified accordingly because each problem has its own unique conditions. Commonly recurring problems exist in both real life and programming life. So, design patterns are given to us to implement our project. These patterns have already been tested and used by many other developers for solving similar problems successfully. Using design patterns also makes it possible to work with clean, standardized, and readable code. Deciding to write a program that does X but using pattern Y is a recipe for disaster. It might work for programs such as `hello world`, fit for demonstrating the code constructs for patterns, but not much else.

Through patterns, we could also find a way to work around the inefficiencies that a language may have. Also, a thing to note here, inefficiency is usually associated with negativity, but it may not necessarily be bad at all times.

In this book, we'll cover PHP design patterns with the Laravel PHP Framework. In the first few chapters, we'll also give examples from the Laravel core code. In the chapters that follow, we'll cover the MVC pattern fundamentals. Then we'll try to examine the differences between an MVC pattern approach to Laravel and a common MVC approach. We hope this book will help you increase your code quality.

Please note that finding the best stable, high-quality solution directly depends on your knowledge of the platform and language. We highly recommend that you be well-versed with the data types and fundamentals of object-oriented programming in PHP and Laravel Framework.

In this chapter, we'll explain design pattern terms and learn about the classification of these design patterns and their elements. We'll give some examples from the Laravel core code, which contains the design patterns used in the framework. Finally, we'll explain the **Mode-View-Controller** (**MVC**) architectural pattern and its benefits.

Design patterns

Design patterns were first introduced by Eric Gamma and his three friends in 1994. A design pattern is basically a pattern of software design that is implemented on multiple projects, and its intended success gives an idea to prove this pattern as a solution of commonly recurring problems.

Design patterns are ways to solve a problem and the way to get your intended result in the best possible manner. So, design patterns are not only ways to create a large and robust system, but they also provide great architectures in a friendly manner.

In software engineering, a design pattern is a general repeatable and optimized solution to a commonly occurring problem within a given context in software design. It is a description or template for how to solve a problem, and the solution can be used in different instances. The following are some of the benefits of using design patterns:

- Maintenance
- Documentation
- Readability
- Ease in finding appropriate objects
- Ease in determining object granularity
- Ease in specifying object interfaces
- Ease in implementing even for large software projects
- Implements the code reusability concept

If you are not familiar with design patterns, the best way to begin understanding is observing the solutions we use for commonly occurring, everyday life problems.

Let's take a look at the following image:

Many different types of power plugs exist in the world. So, we need a solution that is reusable, optimized, and cheaper than buying a new device for different power plug types. In simple words, we need an adapter. Have a look at the following image of an adapter:

In this case, an adapter is the best solution that's reusable, optimized, and cheap. But an adapter does not provide us with a solution when our car's wheel blows out.

In object-oriented languages, we the programmers use the objects to do whatever we want to have the outcome we desire. Hence, we have many types of objects, situations, and problems. That means we need more than just one approach to solving different kinds of problems.

Elements of design patterns

The following are the elements of design patterns:

- **Name**: This is a handle we can use to describe the problem
- **Problem**: This describes when to apply the pattern
- **Solution**: This describes the elements, relationships, responsibilities, and collaborations, in a way that we follow to solve a problem
- **Consequences**: This details the results and trade-offs of applying the pattern

Classification of design patterns

Design patterns are generally divided into three fundamental groups:

- Creational patterns
- Structural patterns
- Behavioral patterns

Let's examine these in the following subsections.

Creational patterns

Creational patterns are a subset of design patterns in the field of software development; they serve to create objects. They decouple the design of an object from its representation. Object creation is encapsulated and outsourced (for example, in a factory) to keep the context of object creation independent from concrete implementation. This is in accordance with the rule: "Program on the interface, not the implementation."

Some of the features of creational patterns are as follows:

- **Generic instantiation**: This allows objects to be created in a system without having to identify a specific class type in code (Abstract Factory and Factory pattern)
- **Simplicity**: Some of the patterns make object creation easier, so callers will not have to write large, complex code to instantiate an object (Builder (Manager) and Prototype pattern)
- **Creation constraints**: Creational patterns can put bounds on who can create objects, how they are created, and when they are created

The following patterns are called creational patterns:

- The Abstract Factory pattern
- The Factory pattern
- The Builder (Manager) pattern
- The Prototype pattern
- The Singleton pattern

Structural patterns

In software engineering, design patterns structure patterns facilitate easy ways for communications between various entities.

Some of the examples of structures of the samples are as follows:

- **Composition**: This composes objects into a tree structure (whole hierarchies). Composition allows customers to be uniformly treated as individual objects according to their composition.
- **Decorator**: This dynamically adds options to an object. A Decorator is a flexible alternative embodiment to extend functionality.
- **Flies**: This is a share of small objects (objects without conditions) that prevent overproduction.
- **Adapter**: This converts the interface of a class into another interface that the clients expect. Adapter lets those classes work together that would normally not be able to because of the different interfaces.
- **Facade**: This provides a unified interface meeting the various interfaces of a subsystem. Facade defines a higher-level interface to the subsystem, which is easier to use.
- **Proxy**: This implements the replacement (surrogate) of another object that controls access to the original object.
- **Bridge**: This separates an abstraction from its implementation, which can then be independently altered.

Behavioral patterns

Behavioral patterns are all about a class' objects' communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects. The following is a list of the behavioral patterns:

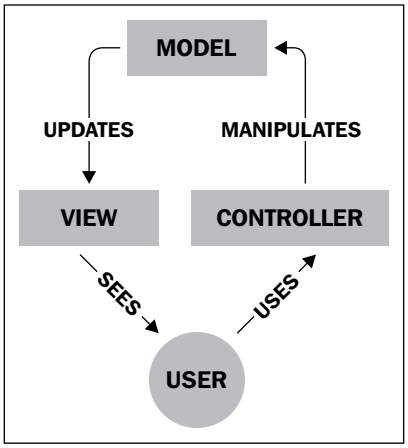
- Chain of Responsibility pattern
- Command pattern
- Interpreter pattern
- Iterator pattern
- Mediator pattern
- Memento pattern
- Observer pattern
- State pattern
- Strategy pattern
- Template pattern
- Visitor pattern

We'll cover these patterns in the following chapters. If you want to check out the usage of some patterns in the Laravel core, have a look at the following list:

- **The Builder (Manager) pattern**: `Illuminate\Auth\AuthManager` and `Illuminate\Session\SessionManager`
- **The Factory pattern**: `Illuminate\Database\DatabaseManager` and `Illuminate\Validation\Factory`
- **The Repository pattern**: `Illuminate\Config\Repository` and `Illuminate\Cache\Repository`
- **The Strategy pattern**: `IIlluminate\Cache\StoreInterface` and `Illuminate\Config\LoaderInterface`
- **The Provider pattern**: `IIlluminate\Auth\AuthServiceProvider` and `Illuminate\Hash\HashServiceProvider`

What is MVC?

The MVC triad of classes were used to build user interfaces in Smalltalk-80 in 1988. MVC is an architectural pattern that is used in software engineering, whose fundamental principle is based on the idea that the logic of an application should be separated from its presentation. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the way that information is presented to or accepted from the user. Refer to the following figure:



In the preceding figure, you'll see the elements of MVC. It shows the general life cycle on an MVC-based application of a request. As you can see, using an MVC architectural pattern in projects allows you to separate different layers of applications, such as the database layer and the UI layer.

The benefits of using the MVC pattern are as follows:

- Different views and controllers can be substituted to provide alternate user interfaces for the same model.
- It provides multiple simultaneous views of the same model.
- The change propagation mechanism ensures that all views simultaneously reflect the current state of the model.
- Changes affecting just the user interface of the application become easier to make.
- It is easier to test the core of the application, as it is encapsulated by the model.
- One of the great benefits of the MVC pattern is that it allows you to recycle the application's logic when you use different templates. For example, when you want to implement an external API inside a part of your application, it will be very helpful to reuse the application's logic. If the MVC approach of Laravel is followed thoroughly, you will only need to modify the controller to render many different templates/views.

Summary

In this chapter, we have explained the fundamentals of design patterns. We've also introduced some design patterns that are used in the Laravel Framework. Finally, we explained the MVC architectural pattern concepts and its benefits.

In the next chapter, we'll cover the MVC concept in depth and its usage in Laravel. Before moving on to learn about design patterns and their usage in Laravel with the actual code, the most important thing is understanding the framework's approach to the MVC concept.

2
Models in MVC

Throughout the chapter, we will be discussing what Model is in the MVC structure, what its purpose is, what its role is in the SOLID design pattern, how Laravel defines it, and the advantages of Laravel's Model layers and Eloquent ORM. We will also discuss Laravel's classes related to handling data.

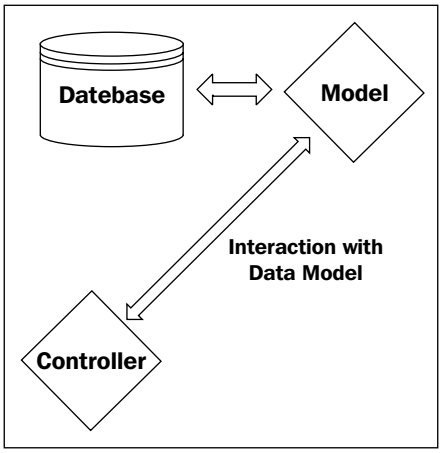
The following is the list of topics that will be covered in this chapter:

- The meaning of the Model
- The roles of the Model in a solid MVC design pattern
- The Model and Model Instances
- How Laravel defines the Model
- The database-related classes of Laravel

What is a Model?

The Model is that part of the Model-View-Controller design pattern that we can simply describe as the layer of the design pattern that handles the management of the data, which is received from the corresponding layers and then sent back to those layers. One thing to note here is that the Model does not know where the data comes from and how it is received.

In simple words, we can say that the Model implements the business logic of the application. The Model is responsible for fetching the data and converting it into more meaningful data that can be managed by other layers of the application and sending it back to corresponding layers. The Model is another name for the domain layer or business layer of an application.



Purposes of the Model

The Model in an application manages all of the dynamic data (anything that's not hardcoded and comes from a database driver) and lets other related components of the application know about the changes. For example, let's say that there is a news article in your database. If you alter it from the database manually, when a route is called—and due to this request—the Controller requests for the data over the Model after the request from the Routing handler, and the Controller receives the updated data from the Model. As a result, it sends this updated data to the View, and the end user sees the changes from the response. All of these data-related interactions are the tasks of the Model. This "data" that the Model handles does not always have to be database related. In some implementations, the Model can also be used to handle some temporary session variables.

The basic purposes of the Model in a general MVC pattern are as follows:

- To fetch the data using a specified (database) driver
- To validate the data
- To store the data
- To update the data

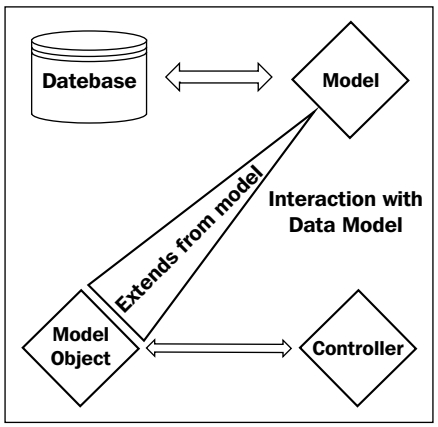
- To delete the data
- To create conditional relations
- To monitor the file I/O
- To interact with third-party web services
- To handle caches and sessions

As you can see, if you follow with an MVC pattern consistently, the Model covers a huge percentage of the application's logic. In modern frameworks, there is a common mistake about the Model that is made by users when learning design patterns. They usually confuse the Model with Model Instances. Although they are quite similar, they have different meanings.

Model instances

In your application, there will usually be more than one data structure to manage. For example, let's say you are running a blog. In a simple blog system, there are authors, blog posts, tags, and comments. Let's say you want to update a blog post; how do you define that the data you want to update is for the blog posts? This is where Model Instances come handy.

Model Instances are simple classes that mostly extend from the Model layer of the application. These instances separate the data logic for each section of your application. In our example, we have four sections to handle (users, posts, tags, and comments). If we are going to handle these using the Model, we have to create at least four instances (we will cover why it is at least four and not exactly four in the *Relationships* section that is under *Eloquent ORM* in this chapter).



As you can see from the diagram, the Controller interacts with the Model Instance to fetch data. Since Model Instances extend from the Model itself, instead of raw Model output, the Controller can customize it or add other layers (such as Validation) to the process.

Let's say you want to fetch the user who has the username `George`. If no Validation layer has been added from the Model Instance to the database, the `username` parameter will go to the database directly, which could be harmful. If you did add a validation layer on your Model Instance (which checks if the username parameter is a clean string), even if the parameter was SQL injection code, it would first be filtered by the validation layer instead of going to the database directly and then be detected as harmful code. Then, the Model Instance will return a message or an exception to the Controller that the parameter is invalid. Then, the Controller will send a corresponding message to the View, and from there, the message will be shown to the end user. In this process, optionally, the application might even fire an Event to log this attempt.

The Model in Laravel

If you recall, we mentioned earlier in this chapter that there are many important jobs that the Model needs to handle. Laravel 4 does not use the MVC pattern directly, but it extends the pattern further. For example, Validation—which is part of the Model in the solid MVC pattern—has its own class, but it's not part of the Model itself. The database connection layer has its own classes for each database driver, but they are not packed in the Model directly. This brings testability, modularity, and extensibility to the Models.

The Laravel Model structure focuses more on the database processes directly, and it is separated from other purposes. The other purposes are categorized as Facades.

To access the database, Laravel has two classes. The first one is Fluent Query Builder and the second one is Eloquent ORM.

Fluent Query Builder

Fluent is the query builder class of Laravel 4. Fluent Query Builder handles base database query actions using PHP Data Objects in the backend, and it can be used with almost any database driver. Let's say that you need to change the database driver from SQLite to MySQL; if you've written your queries using Fluent, then you mostly don't need to rewrite or alter your code unless you've written raw queries using `DB::raw()`. Fluent handles this behind the scenes.

Let's take a quick look at the Eloquent Model of Laravel 4 (which can be found in the `Vendor\Laravel\Framework\src\Illuminate\Database\Query` folder):

```
<?php namespace Illuminate\Database\Query;

use Closure;
use Illuminate\Support\Collection;
use Illuminate\Database\ConnectionInterface;
use Illuminate\Database\Query\Grammars\Grammar;
use Illuminate\Database\Query\Processors\Processor;

class Builder {
    //methods and variables come here
}
```

As you can see, the Eloquent Model uses some classes, such as `Database`, `ConnectionInterface`, `Collection`, `Grammar`, and `Processor`. All of these are required to standardize database queries in the backend, cache the queries if required, and return the output as a collection object.

The following are some basic examples that present how the queries look:

- To get all of the names and show them one by one from a `users` table, use the following code:

  ```
  $users = DB::table('users')->get();
  foreach ($users as $user)
  {
      var_dump($user->name);
  }
  ```

  The `get()` method fetches all of the records from the table in the form of a collection. With a `foreach()` loop, the records are looped, and then we access each name column using `->name` (an object). If the column we want to access is an e-mail, then it'll look like `$user->email`.

- To fetch the first user named `Arda` from the `users` table, use the following code:

  ```
  $user = DB::table('users')->where('name', 'Arda')->first();
  var_dump($user->name);
  ```

  The `where()` method filters the query with given parameters. The `first()` method directly returns the collection object of a single item from the first matched element. If there were two users named `Arda`, only the first one would be caught and set to the `$user` variable.

- If you wanted to use `OR` statements in the `where` clauses, you could use the following code:

```
$user = DB::table('users')
->where('name', 'Arda')
->orWhere('name', 'Ibrahim')
->first();
var_dump($user->name);
```

- To use operators in the `where` clauses, the following third parameter should be added between the column name and variable that is to be filtered:

```
$user = DB::table('users')->where('id', '>', '2')->get();
foreach ($users as $user)
{
    var_dump($user->email);
}
```

- If you are using offsets and limits, execute the following query:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

  This produces `SELECT * FROM` users `LIMIT 10,5` in MySQL. The `skip($integer)` method will set an offset to the query, and `take($ integer)` will limit the output by the natural number that has been set as the parameter.

- You can also limit what is to be fetched using the `select()` method and use the following `join` statements easily in Fluent Query Builder:

```
DB::table('users')
   ->join('contacts', 'users.id', '=', 'contacts.user_id')
   ->join('orders', 'users.id', '=', 'orders.user_id')
   ->select('users.id', 'contacts.phone', 'orders.price');
```

  These methods simply join the `users` table with contacts, then join orders with users, and then get the user ID, the phone column from the `contacts` table, and the price column from the `orders` table.

- You can group queries by parameters easily, using closure functions. This will allow you to write more complicated queries with ease, as follows:

```
DB::table('users')
    ->where('name', '=', 'John')
    ->orWhere(function($query)
    {
        $query->where('votes', '>', 100)
              ->where('title', '<>', 'Admin');
```

```
    })
    ->get();
```

This will produce the following SQL query:

```
select * from users
   where name = 'John'
   or
   (votes > 100 and title <> 'Admin')
```

- You can also use aggregations in the query builder (such as `count`, `max`, `min`, `avg`, and `sum`) as follows:

```
$users = DB::table('users')->count();
$price = DB::table('orders')->max('price');
```

- Sometimes, such builders might not be enough, or you might want to run raw queries. You can also wrap your raw queries inside of Fluent as follows:

```
$users = DB::table('users')
     ->select(
array(
DB::raw('count(*) as user_count'),
'status',
)
)
     ->where('status', '<>', 1)
     ->groupBy('status')
     ->get();
```

- To insert new data into the table, use the `insert()` method:

```
DB::table('users')->insert(
    array('email' => 'me@ardakilicdagi.com', 'points' => 100)
);
```

- To update a row(s) from a table, use the `update()` method:

```
DB::table('users')
->where('id', 1)
->update(array('votes' => 100));
```

- To delete a row(s) from a table, use the `delete()` method:

```
DB::table('users')
->where('last_login', '2013-01-01 00:00:00')
->delete();
```

- Benefiting of `CachingIterator` that's used the `Collection` class, Fluent Query Builder can also cache results upon calling using the method `remember()`:

```
$user = DB::table('users')
->where('name', 'Arda')
->remember(10)
->first();
```

  After this query is called once, it's cached for 10 minutes; if this query is called again, instead of fetching from the database, it'll fetch directly from the cache instead, until 10 minutes have passed.

Eloquent ORM

Eloquent ORM is the Active Record implementation in Laravel. It's simple, powerful, and easy to handle and manage.

For each database table, you'll need a new Model Instance to benefit from Eloquent.

Let's say you have a `posts` table, and you want to benefit from Eloquent; you need to navigate to `app/models` and save this file as `Post.php` (the singular form of the table name):

```
<?php class Post extends Eloquent {}
```

And that's it! You're ready to benefit from Eloquent methods for your table.

Laravel allows you to assign any table to any Eloquent Model Instance. It's not necessary, but it's a good habit to name Model Instances with the singular name of the corresponding table. This name should be a singular form of the table name it represents. If you have to use a name that does not follow this general rule, you can do so by setting a protected `$table` variable inside of the Model Instance.

```
<?php Class Post Extends Eloquent {
   protected $table = 'my_uber_posts_table';
}
```

This way, you can assign a table to any desired Model Instance.

It's not necessary to add the instance into the `models` folder in `app`. As long as you've set an `autoload` path in `composer.json`, you can get rid of this folder completely and add it wherever you like. This will bring flexibility to your architecture during programming.

Let's take a quick look at the following `Model` class of Laravel 4 that we just extended from (which is in the `Vendor\Laravel\Framework\src\Illuminate\Database\Eloquent` folder):

```
<?php namespace Illuminate\Database\Eloquent;

use DateTime;
use ArrayAccess;
use Carbon\Carbon;
use LogicException;
use JsonSerializable;
use Illuminate\Events\Dispatcher;
use Illuminate\Database\Eloquent\Relations\Pivot;
use Illuminate\Database\Eloquent\Relations\HasOne;
use Illuminate\Database\Eloquent\Relations\HasMany;
use Illuminate\Support\Contracts\JsonableInterface;
use Illuminate\Support\Contracts\ArrayableInterface;
use Illuminate\Database\Eloquent\Relations\Relation;
use Illuminate\Database\Eloquent\Relations\MorphOne;
use Illuminate\Database\Eloquent\Relations\MorphMany;
use Illuminate\Database\Eloquent\Relations\BelongsTo;
use Illuminate\Database\Query\Builder as QueryBuilder;
use Illuminate\Database\Eloquent\Relations\MorphToMany;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;
use Illuminate\Database\Eloquent\Relations\HasManyThrough;
use Illuminate\Database\ConnectionResolverInterface as Resolver;

abstract class Model implements ArrayAccess, ArrayableInterface,
JsonableInterface, JsonSerializable {
   //Methods and variables come here
}
```

Eloquent uses `Illuminate\Database\Query\Builder`, which is the Fluent Query Builder that we described earlier, and its methods are defined inside it. Thanks to this, all of the methods that can be defined in Fluent Query Builder can also be used in Eloquent ORM.

As you can see, all of the used classes are split according to their purpose. This brings a better **Abstraction** and **Reusability** to the architecture.

Relationships

Eloquent ORM has other benefits in addition to Fluent Query Builder. The major benefit is Model Instance Relations, which allows Fluent Query Builder to form a relationship with other Model Instances easily. Let's say you have `users` and `posts` tables, and you want to get the posts made by a user with an ID of `5`. After the relationship is set, the collection of these posts can be fetched with this code easily:

```
User::find(5)->posts;
```

This couldn't be easier, could it? There are three major relationship types: one-to-one, one-to-many, and many-to-many. In addition to these, Laravel 4 also has the has-many-through and morph-to-many (many-to-many polymorphic) relationships:

- **One-to-one relationships**: These are used when both Models have only one element of each other. Let's say you have a `User` Model, which should only have one element in your `Phone` Model. In this case, the relationship will be defined as follows:

```
//User.php model
Class User Extends Eloquent {

   public function phone() {
      return $this->hasOne('Phone'); //Phone is the name of
        Model Instance here, not a table or column name
   }

}

//Phone.php model
Class Phone Extends Eloquent {

   public function user() {
      return $this->hasOne('User');
   }

}
```

- **One-to-many relationships**: These are used when a Model has more than one element of another. Let's say you have a news system with categories. A category can have more than one item. In this case, the relationship will be defined as follows:

```
//Category.php model
class Category extends Eloquent {

   public function news() {
```

```
        return $this->hasMany('News'); //News is the name of
          Model Instance here
    }

}

//News.php model
class News extends Eloquent {

    public function categories() {
        return $this->belongsTo('Category');
    }

}
```

- **Many-to-many relationships**: These are used when two Models have more than one element of each other. Let's say you have `Blog` and `Tag` Models. A blog post might have more than one tag, and a tag might be assigned to more than one blog post. For such instances, a pivot table is used along with Many to Many Relationships. The relationship can be defined as follows:

```
//Blog.php Model
Class Blog Extends Eloquent {

    public function tags() {
        return $this->belongsToMany('Tag', 'blog_tag');
          //blog_tag is the name of the pivot table
    }

}

//Tag.php model
Class Tag Extends Eloquent {

    public function blogs() {
        return $this->belongsToMany('Blog', 'blog_tag');
    }

}
```

  Laravel 4 adds some flexibility and additional relationships to these known relationships. They are "has-many-through" and "polymorphic relationships".

- **Has-many-through relationships**: These are more like shortcuts. Let's say you have a `Country` Model, `User` Model, and `Post` Model. A country may have more than one user, and a user may have more than one post. If you want to access all of the posts created by the users of a specific country, you need to define the relationship as follows:

```
//Country.php Model
Class Country Extends Eloquent {

   public function posts() {
      return $this->hasManyThrough('Post', 'User');
   }

}
```

- **Polymorphic relationships**: These are featured in Laravel v4.1. Let's say you have a `News` Model, `Blog` Model, and `Photo` Model. This `Photo` Model holds images for both `News` and `Blog`, but how do you relate this or identify a specific photo that is either for blogs or posts? This can be done easily. It needs to be set as follows:

```
//Photo.php Model
Class Photo Extends Eloquent {

   public function imageable() {
      return $this->morphTo(); //This method doesn't take
       any parameters, Eloquent will understand what will
         be morphed by calling this method
   }

}

//News.php Model
Class News Extends Eloquent {

   public function photos() {
      return $this->morphMany('Photo', 'imageable');
   }

}

//Blog.php Model
Class Blog Extends Eloquent {

   public function photos() {
      return $this->morphMany('Photo', 'imageable');
   }

}
```

The keyword `imageable`, which will describe the owner of the image, is not a must; it could be anything, but you need to set it as a method name and put it as a second parameter into `morphMany` relationship definitions. This helps us understand how we're going to access the owner of the photo. This way, we can call this easily from the `Photo` Model without needing to understand whether its owner is `Blog` or `News`:

```
$photo = Photo::find(1);
$owner = $photo->imageable; //This brings either a blog
  collection or News according to its owner.
```

In this example, you'll need to add two additional columns to your `Photo` Model's table. These columns are `imageable_id` and `imageable_type`. The section `imageable` is the name of the morphing method, and the suffix's ID and type are the keys that will define the exact ID and type of the item that it will be morphed to.

Mass assignment

When creating a new Model Instance (when inserting or updating data), we pass a variable that is set as an array with attribute names and values. These attributes are then assigned to the Model by mass assignment. If we blindly add all of the inputs into mass assignment, this will become a serious security issue. In addition to querying methods, Eloquent ORM also helps us with mass assignment. Let's say you don't want the column e-mail in your `User` Model (`Object`) to be altered in any way (blacklist), or you just want the title and body columns to be altered in your `Post` Model (whitelist). This can be done by setting protected `$fillable` and `$guarded` variables in your Model:

```
//User.php model
Class User Extends Eloquent {
   //Disable the mass assignment of the column email
   protected $guarded = array('email');

}


//Blog.php model
Class User Extends Eloquent {
   //Only allow title and body columns to be mass assigned
   protected $fillable = array('title', 'body');

}
```

Soft deleting

Let's say you have a `posts` table, and let's assume the data inside this table is important. Even if the `delete` command is run from the Model, you want to keep the deleted data inside your database just in case. In such cases, you can use soft deletes with Laravel.

Soft deleting doesn't actually delete the row from the table; instead, it adds a key if the data is actually deleted. When a soft deletion is made, a new column called `deleted_at` is filled with a timestamp.

To enable the soft deletes, you need to first add a timestamp column called `deleted_at` to your table (you could do this by adding `$table->softDeletes()` to your migration), then set a variable called `$softDelete` to `true` in your Model Instance.

The following is an example Model Instance for soft deletes:

```
//Post.php model
Class Post Extends Eloquent {
   //Allow Soft Deletes
   protected $softDelete = true;

}
```

Now, when you run the `delete()` method in this model, instead of actually deleting the column, it will add a `deleted_at` timestamp to it.

Now, when you run the `all()` or `get()` method, the soft-deleted columns won't be listed, like they have actually been deleted.

After such deletes, you might want to get results along with soft-deleted rows. To do this, use the `withTrashed()` method as follows:

```
$allPosts = Post::withTrashed()->get(); //These results will include
both soft-deleted and non-soft-deleted posts.
```

In some cases, you may want to fetch only soft-deleted rows. To do this, use the `onlyTrashed()` method as follows:

```
$onlySoftDeletedPosts = Post::onlyTrashed()->get();
```

To restore the soft-deleted rows, use the `restore()` method. To restore all soft-deleted posts, run a code like the following:

```
$restoreAllSoftDeletedPosts = Post::onlyTrashed()->restore();
```

To hard delete (totally delete) the soft-deleted rows from a table, use the `forceDelete()` method as follows:

```
$forceDeleteSoftDeletedPosts = Post::onlyTrashed()->forceDelete();
```

When fetching rows from a table (including soft deletes), you may want to check whether they have been soft deleted or not. This check is done by running the `trashed()` method on collection rows. This method will return a Boolean value. If true, it means the row has been soft deleted.

```
//Let's fetch a post without the soft-delete checking:
$post = Post::withTrashed()->find(1);
//Then let's check whether it's soft deleted or now
    if($post->trashed()) {
return 'This post is soft-deleted';
  } else {
   return 'This post is not soft-deleted';
}
```

Eager loading

Eloquent ORM also brings a neat solution to the N+1 query problem with **Eager Loading**. Let's assume that you have a query and loop like the following:

```
$blogs = Blog::all();
foreach($blogs as $blog) {
   var_dump($blog->images());
}
```

In this case, to access the images, one more query is executed for each loop in the backend. This will exhaust the database drastically, so to prevent this, we will use the `with()` method on the query. This will fetch all of the blogs and images, relate them in the backend, and serve them as a collection directly. Refer to the following code:

```
$blogs = Blog::with('images')->get();
foreach($blogs as $blog) {
   var_dump($blog->images);
}
```

This way, the querying will be much faster, and fewer resources will be used.

Timestamps

The main benefits of Eloquent ORM are seen when you set `$timestamps` to `true` (which is the default); you will have two columns, the first is `created_at` and the second is `updated_at`. These two columns keep the creation and last update times of data as timestamps and update them automatically on the creation or update of each row.

Query scopes

Let's say that you repeat a `where` condition several times because it's a commonly used clause in your application and this condition means something. Let's say you want to get all of the blog posts that have more than 100 views (we'll call it popular posts). Without using scopes, you'd get the posts in the following format:

```
$popularBlogPosts = Blog::where('views', '>', '100')->get();
```

However, in the example, you'll be repeating this through your application over and over again. So, why not set this as a scope? You can do this easily using Laravel's Query Scope feature.

Add the following code to your `Blog` model:

```
public function scopePopular($query) {
   return $query->where('views', '>', '100');
}
```

After doing this, you can use your scope easily with the following code:

```
$popularBlogPosts = Blog::popular()->get();
```

You can also chain the post as follows:

```
$popularBlogPosts = Blog::recent()->popular()->get();
```

Accessors and mutators

One of the features of Eloquent ORM is accessors and mutators. Let's say you have a column called `name` on your table, and on calling this column, you want to pass PHP's `ucfirst()` method to uppercase its name. This can be done by simply adding the following lines of code to the model:

```
public function getNameAttribute($value) {
    return ucfirst($value);
}
```

Now, let's consider the opposite. Each time you save or update the name column, you want to pass the PHP `strtolower()` function to the column (you want to mutate the input). This can be done by adding the following lines of code to the model:

```
public function setNameAttribute($value) {
    return strtolower($value);
}
```

Note that the method name should be CamelCased even though the column name is `snake_cased`. If your column name is `first_name`, the getter method name should be `getFirstNameAttribute`.

Model events

Model events play an important part in the Laravel design pattern. Using Model events, you can call any method right after the event is fired.

Let's say that you have set a cache for your comments, and you want to flush the cache each time a comment is deleted. How can you catch the comment's deletion event and do something there? Should there be various places in the application that such comments can be deleted? Is there a way to catch exact the "deleting" or "deleted" event? In such case, the Model events come in handy.

Models hold the following methods: `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, and `restored`.

Whenever a new item is being saved for the first time, the creating and created events will fire. If you are updating a current item on the Model, the `updating`/`updated` events will fire. Whether you are creating a new item or updating a current one, the `saving`/`saved` events will fire.

If `false` is returned from the `creating`, `updating`, `saving`, or `deleting` event, the action will be canceled.

For example, let's check whether a user has been created. If its first name is `Hybrid`, we'll cancel the creation. To add this condition, include the following lines of code in your `User` Model:

```
User::creating(function($user){
    if ($user->first_name == 'Hybrid') return false;
});
```

Model observers

Model observers are quite similar to Model Events, but the approach is a little bit different. Instead of defining all events (`creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, and `restored`) inside of the Model, it "abstracts" the logic of the events to a different class and "observes" it with the `observe()` method. Let's assume we have a Model event like the following:

```
User::creating(function($user){
    if ($user->first_name == 'Hybrid') return false;
});
```

To keep the abstraction, it will be much better to wrap all of these events and separate their logic from the Model. In an observer, these events will look like the following:

```
class UserObserver {

   public function creating($model){
       if ($model->first_name == 'Hybrid') return false;
    }

    public function saving($model)
    {
        //Another model event action here
    }

}
```

As you can imagine, you can put this class anywhere in your application. You can even group all of these events in a separate folder for a better architectural pattern.

Now, you need to register this event `Observer` class to a Model. This can be done with the following simple command:

```
User::observe(new UserObserver);
```

The main advantage of this approach is that you can use observers in more than one Model and register more than one observer to a Model this way.

Migrations

Migrations are easy tools to version control your database. Let's say there is a place where you need to add a new column to the table or roll back to the previous state because you did something wrong or the link to your application broke. Without migrations, these are tedious tasks to handle, but with migrations, your life will be much easier.

There are various reasons to use migrations; some of these are as follows:

- You'll benefit from this versioning system. If you made a mistake or need to roll back to a previous state, you can do so with only a single command using migrations.
- The use of migrations for alteration will bring about flexibility. The migrations that are written will work on all supported database drivers, so you won't need to rewrite database code again and again for different drivers. Laravel will handle this in the background.

- They are quite easy to generate. Using the migration commands of the Laravel `php` client, which is called `artisan`, you can manage all of your application's migrations.

The following is what a migration file looks like:

```
<?php

use Illuminate\Database\Migrations\Migration;

class CreateNewsTable extends Migration {

        /**
        * Run the migrations.
        */
        public function up()
        {
                //
        }

        /**
        * Reverse the migrations.
        */
        public function down()
        {
                //
        }

}
```

The `up()` method runs when the migration is run forward (a new migration). The `down()` method runs when the migration is run backward, meaning it reverses or resets (reverses and reruns) the migration.

After these methods are triggered via the `artisan` command, it runs the method up or down, corresponding the parameters of the `artisan` command, and returns the status of the message.

Database seeders

Let's say you've programmed a blog application. You need to show what it's capable of, but there are no example blog posts to show the awesome blog you've programmed. This is where seeders come in handy.

Database seeders are some simple classes that fill random data in a specified table. The seeder class has a simple method called `run()` to make this seeding(s). The following is what a seeder looks like:

```
<?php

class BlogTableSeeder extends Seeder {

  public function run()
  {
    DB::table('blogs')->insert(array(
      array('title' => 'My Title'),
      array('title' => 'My Second Title'),
      array('title' => 'My Third Title')
    ));
  }

}
```

When you call this class from a terminal using the `artisan` command, it connects to the database and fills it with the given data. After this attempt, it returns a command message to the user over the terminal about the status of the seeding.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Summary

In this chapter, we have learned about the role of the Model in the MVC pattern and how Laravel 4 "defines" the Model by extending its roles to various classes. We've also seen what the Model components of Laravel 4 are capable of with examples.

In the next chapter, we'll learn about the role of the View and how it interacts with end users and other aspects of the application using the MVC pattern on Laravel 4.

3
Views in MVC

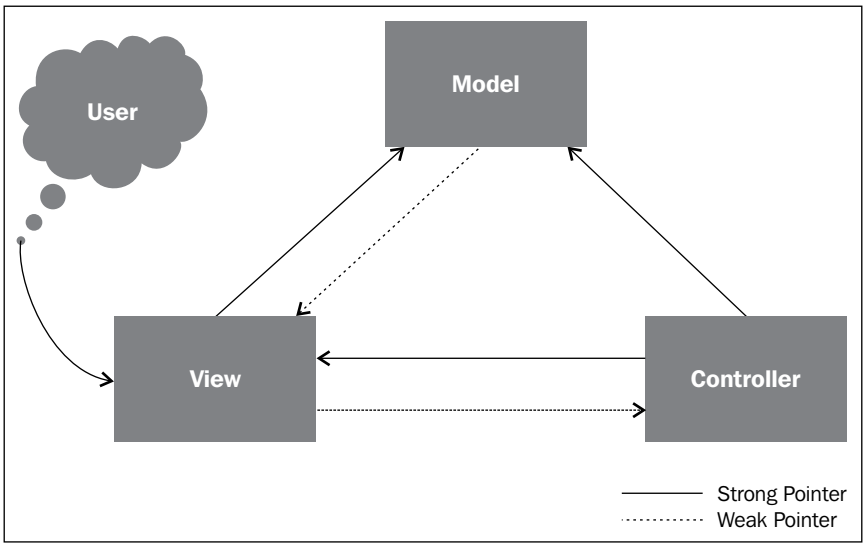
In this chapter, we will discuss what the View is, its structure, its purpose, and the advantages of Laravel's View layer and Blade template engine.

What is a View?

The View in Laravel refers to the V in MVC. The View consists of presentation logic aspects such as templates and caching and the code that involves presentation. Precisely, the View defines exactly what is presented to the user. Usually, Controllers pass data to each View to render in some format. Views collect data from users as well. This is where you're likely to find HTML markup in your MVC application.

Most modern MVCs, such as Laravel Framework, implement a template language that adds a further layer of abstraction from PHP. Added layers mean added overhead. Here, we stick with the speed of PHP within the template, yet all the logic stays outside. This makes it easy for User Interface (UI) designers to develop themes/templates without the need to learn any programming languages.

In many MVC implementations, the View layer speaks with Controllers and Models. The approach is well explained in the following figure:

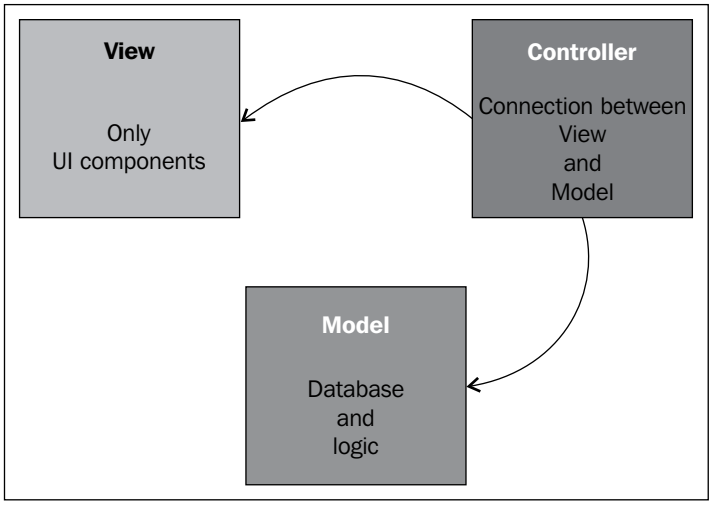


As you can see in the preceding figure, the View communicates with both the Controller and the Model. At first sight, it seems like a very flexible approach to develop an application with OOP languages. Sharing data between all objects of MVC and accessing them in any layer of application sounds very cool. Unfortunately, the method causes some problems that depend on the project's size.

The foremost problems are the complexity of allocating development tasks between teams/developers. If you don't set the development rules, it will lead to chaotic situations such as unmanageable spaghetti code. Also, we have to consider extra costs of development such as training the developers and comparatively long development processes, which directly affect the cost of the project.

As we mentioned at the beginning of the book, development not only involves coding or sharing tasks, it's also the process that includes the planning and marketing of the project development method.

Laravel ships a different kind of approach to MVC. According to Laravel, the View layer should only communicate with the Controller. The Model communicates with the Controller. Let's look at the following figure:



As you can see in the preceding figure, the layers of application are completely separated. Thus, you can get easily manageable code and a development team. Generally, we need at least three files in MVCs: the Model file, the Controller file, and the View file. Let's explain the View file through its objects.

View objects

In your application, there is usually more than one HTML page that contains forms, asset references, and so on; for example, if you are developing an e-commerce application. In a simple e-commerce system, there are product lists, categories, carts, and product detail pages that we need. This means that we need four templates and too much data to present to our users. We can group the objects of the View layer as shown:

- HTML elements (div, header, section, and so on)
- HTML form elements (input, select, and so on)
- Asset and JavaScript references (`.css` and `.js`)

When you work on a project that has dynamic data, separating the template files does not help simplify the problem because you still need programming language functions to process objects. This causes what we don't want to face—spaghetti code. When a project has inline PHP code in HTML documents, you will face problems in keeping the code simple. Let's take a look at the generic template file contents that are not implemented with any template language in the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title><?php echo $title; ?></title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"
  />
<meta http-equiv="x-ua-compatible" content="chrome=1" />
<meta name="description" content="<?php echo $meta_desc; ?>" />
<meta name="keywords" content="<?php echo $meta_keys; ?>" />
<meta name="robots" content="index,follow" />
<meta property="og:title" content="<?php echo $title; ?>" />
<meta property="og:site_name" content="<?php echo $site_name; ?>"
  />
<meta property="og:image" content="http://www.example.com/
images/"<?php echo $thumbnail; ?>
  />
<meta property="og:type" content="product" />
<meta property="og:description" content="<?php echo $meta_desc;
  ?>" />
</head>
<body>
<?php
if($user_name){
?>
<div class="username">Welcome <?php echo strtoupper($user_name);
  ?> !</div>
<?php
}else{

echo '<div class="username">Welcome Guest!</div>';

}
?>
</body>
```

As a UI developer, you need prior knowledge of the PHP language (at least knowledge of the syntax of the language) to understand the code you see. As we mentioned in *Chapter 1*, *Design and Architectural Pattern Fundamentals*, most modern MVC frameworks come with a template language bundled to be used in Views. Laravel ships an implemented template language to be used with Views; this is called the Blade template engine, or simply Blade.

View in Laravel

According to Laravel's MVC approach, the View handles data from the Controller. This means that the View gets the data that is usually already formatted as we need. If the View directly communicates with the Model, we have to format, validate, or filter the data at the View layer, as seen in the previous example code. So, let's see what a Blade template file looks like in the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>{{$title}}</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"
  />
<meta http-equiv="x-ua-compatible" content="chrome=1" />
<meta name="description" content="{{$meta_desc}}" />
<meta name="keywords" content="{{$meta_keys}}" />
<meta name="robots" content="index,follow" />
<meta property="og:title" content="{{$title}}" />
<meta property="og:site_name" content="{{$site_name}}" />
<meta property="og:image" content="http://www.example.com/
images/"{{$thumbnail}} />
<meta property="og:type" content="product" />
<meta property="og:description" content="{{$meta_desc}}" />

</head>

<body>
@if($user_name)

<div class="username">Welcome {{$user_name}} !</div>

@else

<div class="username">Welcome Guest !</div>

@endif
</body>
```

There is neither PHP syntax nor any unclosed brackets problems. So, we have a cleaner template file. Thanks to Blade's built-in features, we can get clearer View files. Generally, the header and footer sections are common to all pages in our applications. There are two ways to add them. The first and unrecommended way is to separate the header, footer, and body sections in three files, similar to something shown in the following example:

```
@include('header')
<body>
@if($user_name)

<div class="username">Welcome {{Str::upper($user_name)}} !</div>

@else

<div class="username">Welcome Guest !</div>

@endif
</body>

@include('footer')
```

This way is not recommended because it requires each page to include both the header and the footer. This also means that if we add a right or left column, we will need to change all the Views of our application. The best way to implement this in Blade is shown as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>{{$title}}</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta http-equiv="x-ua-compatible" content="chrome=1" />
<meta name="description" content="{{$meta_desc}}" />
<meta name="keywords" content="{{$meta_keys}}" />
<meta name="robots" content="index,follow" />
<meta property="og:title" content="{{$title}}" />
<meta property="og:site_name" content="{{$site_name}}" />
<meta property="og:image" content="http://www.example.com/
images/"{{$thumbnail}} />
<meta property="og:type" content="product" />
<meta property="og:description" content="{{$meta_desc}}" />

</head>
```

```
<body>

   @yield('content')

</body>
```

The preceding file is our layout View, for example, `master_layout.blade.php`. As you can see, there is a function for the content that is using the `yield()` function. This is a placeholder; therefore, when any View file extends this file, the section named `content` will be shown instead of the `yield()` function. You can define as many sections in your master layout as you need. So, when we want to use this layout in a View file, we should use it as shown in the following code:

```
@extends('master_layout')

@section(''content')
<body>
@if($user_name)

<div class="username">Welcome {{Str::upper($user_name)}} !</div>

@else

<div class="username">Welcome Guest !</div>

@endif
</body>

@stop
```

That's it! You can extend the master layout in as many Views as you need, and you can create multiple layouts as your application requires.

Summary

In this chapter, we learned the role of Views in the MVC pattern, and what Laravel's approach to Views is. We saw the basics of the Blade template engine functions. For more information, please refer to Laravel's online documentation at `http://laravel.com/`.

In the next chapter, we'll cover the role of the Controller, the maestro in the MVC philosophy of Laravel.

4
Controllers in MVC

Throughout this chapter, we will be discussing what a Controller is, its structure, what its purpose is in the MVC pattern, and its usage in Laravel's extended design pattern and structure.

The topics that will be discussed in this chapter are as follows:

- What is a Controller?
- The Controller's purpose in the MVC design pattern
- The Controller's interaction with other components of the MVC design pattern
- How Laravel handles the Controller in its design pattern

What is a Controller?

The Controller is a part of the Model-View-Controller (MVC) design pattern that we can simply describe as the logical layer of our application. It understands the requests that come from the other end (a user or an API request), makes calls to the corresponding methods, performs primary checks, handles the logic of the request, and then returns the data to the corresponding View or redirects the end user to another route.

The purpose of the Controller

The following are some of the major roles of the Controller in an MVC structure:

- Holding the logic of the application and defining which event should be fired upon actions
- Being the intermediary step between the Model, View, and other components of the application
- Translating actions and responses that come from the View and Model that can be understood by them and sending them to other layers
- Making a bridge between other components of the application and facilitating communication between them
- Making primary permission checks in construct methods before any action

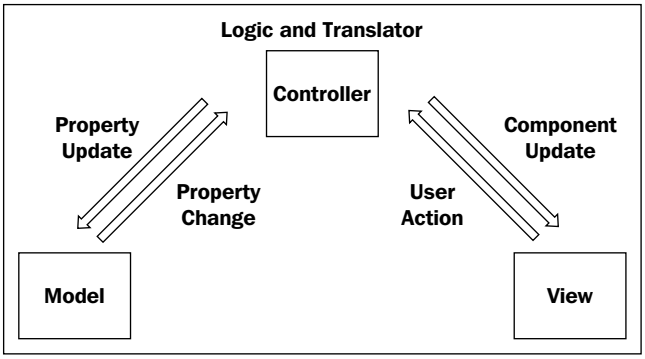
This can best be explained with a real-world example.

Let's assume we have a user management website, and in the administration panel, an administrator tries to delete a user. In design patterns that follow SOLID principles, the following things happen if an admin clicks on the delete user button:

1. From the View, the admin sends a request to the corresponding Controller to delete the news item.
2. The Controller understands this request and performs primary checks. First, it checks if the requester (the administrator, in our case) is really an administrator and has permissions to delete this user.
3. After the primary checks, the Controller tells the Model to delete the user.
4. The Model performs some checks of its own and either deletes the user and tells the Controller that the user is deleted, or tells the Controller that the user is not available (maybe the user is already deleted).
5. After the response comes from the Model, the Controller either tells the View to tell the administrator the user is deleted or redirects to another page with a response like a 404 not found page.

As you can see from the preceding example, for all interactions, the Controller holds the major role for the communication between the components of the application. In an MVC pattern that follows SOLID principles, without the Controller, the View cannot interact with the Model and vice versa. Although there are some derivations to this architectural pattern, like where View interacts with Model(s) directly, in a perfect SOLID design architecture, the Controller should always be the intermediate element for all interactions.

The Controller can also be considered as a translator. It gets input from the View in various ways and converts it to request(s) that can be understood by the Model(s), and vice versa.



Controllers in Laravel

In Laravel 4, the Controllers are simple PHP classes with their filenames and classnames ending with the suffix `Controller` (not forced, but highly recommended; it's a standard between developers), which extends the class `BaseController`, and are stored in the folder `app/controllers` as default. This folder structure is defined in the `composer.json` file's `classmap` key and is not forced. Thanks to the Composer, as long as you define where the Controllers are stored in your application's structure, you can put them in any folder you like.

The following is a very simple Controller for Laravel 4:

```
<?php

class UserController extends BaseController {

    public function showProfile($id)
    {
        $user = User::find($id);

        return View::make('user.profile', array('user' => $user));
    }

}
```

The Controller holds all the action methods for actions that are defined in `routes.php`, in which all the actions (every link that the users interact with) are set in Laravel 4.

In Laravel 3, the developers had to prefix methods with either `GET` or `POST` so as to understand the corresponding request's type. If your request was a `GET` request, your method's name would have to be `get_profile`, and if the request was a `POST` request, it'd have to be like `post_profile`. Thanks to Laravel 4, it's now not forced, and you can name your methods any way you like.

A question now crops up. How do we access this method of the Controller? As we mentioned earlier, we'll be using routes to do this.

Routes

Routes are a set of rules defined in `app/routes.php` that tell Laravel, upon receiving an incoming request, which closure functions and/or Controller methods are being called based on the requested URL. There are various ways to define a route. Three of these are explained as follows:

- You can use closure functions and set the logic for the action directly from `app/routes.php`. Have a look at the following code:

  ```
  Route::get('hello', function(){
     return 'Ahoy, everyone!';
  });
  ```

  Here, we've called the `get()` method because we want this route to be a `GET` request. The first parameter is the path of the action, so if we call `http://ourwebsite.com/hello`, this route action will be called. The second parameter can be from various selections. It can either be an array that holds the name, filter and action, a string that defines the Controller's method for action, or a closure function which holds the logic directly. In our example, we've put a closure function and returned a string directly to the end user. So if the user navigates to `http://ourwebsite.com/hello`, the end user will see the message **Ahoy, everyone!** directly.

- The second way to set a route is to pass a second parameter as a string, define which Controller it is passed to, and the action to be called. Have a look at the following code:

  ```
  Route::get('hello', 'ProfileController@hello');
  ```

  Here, the string `ProfileController@hello` tells Laravel that the method `hello` will be called from the Controller named `ProfileController`. We've separated them using the character `@`.

- The third way is to set an array as a second parameter, which gets various keys and values. Have a look at the following code:

```
Route::get('hello', array(
   'before'   => 'member',
   'as'       => 'our_hello_page'
   'uses'     => 'ProfileController@hello'
));
```

The array can have more than one parameter that define the route's name, the filter that'll be applied before calling the action, and which Controller and its method(s) will be used. The following are the three different keys:

- The `before` key defines the filter before calling the action, so you can set some filtering parameters before calling each action. For example, if you have a members-only area and you don't want guests to access that resource, you can pass filters by using the `before` parameter in your route.
- The `as` key defines the name of the route. This is quite beneficial. Let's say you need to change the URL structure of your application. Classically, if you change the route's action path, you need to change every URL or redirections for this action through your application. Instead, if you set the links and redirections with names, you only need to change the path once, and all links and redirections will be magically fixed.
- The `uses` key has the exact same structure as our second example. It holds the name of the Controller and its method(s) upon calling.

In all these examples, the route did not get any parameters, and we didn't pass any parameters. Think like this: we have accessed the profiles area by using routes, but in these examples, we didn't set a way to access a specific user. How would we set a parameter to these routes? For this, we'll have to set the parameters in curly brackets.

Run the following to set a parameter to a route:

```
Route::get('users/{id}', function($id){
   return 'Hello, the user with ID of '.$id;
});
```

The parameter(s) in curly brackets become a variable name directly in closure methods. This approach also offers us a way to filter these parameters before the Controller's method runs. Using `where()`, you can filter these parameters. Have a look at the following code:

```
Route::get('users/{id}', function($id){
   return 'Hello, the user with ID of '.$id;
})->where(array('id' => '[0-9]+'));
```

The method `where()` is either an array with keys and values, or two parameters, in which the first is the name in curly brackets, and the second is a regular expression to filter the parameter.

In our example, we've filtered the parameter ID with a regular expression to match only numbers, so in this way, we will be distinguishing different kinds of data to overload the endpoint.

Also, there is another benefit to this approach. If a person tries to navigate to `http://ourwebsite.com/users/xssSqlInjection`, Laravel will throw a 404 error even before going to the Controller's method.

If we follow this structure, we need to set each action one by one for each `GET`, `POST`, `PUT`, and `DELETE` requests. If you want to use the RESTful structure for your actions, instead of setting each route one by one, you can use the `controller()` method of the `Route` class.

Certain steps need to be followed to set the routes of a RESTful Controller. For a user Controller, the following are the steps:

1. You first need to make a new Controller named `UserController` and set your RESTful methods such as `index()`, `create()`, `store()`, `show($id)`, `edit($id)`, `update($id)`, and `destroy($id)`.
2. Then you need to set the RESTful Controller in `app/routes.php` by running the following:

   ```
   Route::controller('users', 'UserController');
   ```

Laravel offers a faster way to create Resourceful Controllers. These are called Resource Controllers by Laravel. Follow these steps to set the routes of a Resourceful Controller:

1. You first need to make a new Controller with Resourceful methods using Laravel's PHP client, `artisan`. Have a look at the following code:

   ```
   php artisan controller:make NewsController
   ```

   With this command, a new file named `NewsController.php` is automatically generated in the `app/controllers` folder with all the Resourceful methods already defined inside it.

2. Then you need to set the Resourceful Controller in `app/routes.php` by running the following:

   ```
   Route::resource('news', 'NewsController');
   ```

   While setting the Resourceful Controller, you can set what actions are to be included or excluded by setting a third parameter to this `Route` definition.

3. To include actions that will be defined in the Resourceful Controller (kind of like a whitelist), you use the `only` key as follows:

```
Route::resource(
   'news',
   'NewsController',
   array('only' => array('index', 'show'))
);
```

4. To exclude actions that will be defined in the Resourceful Controller (kind of like a blacklist), you use the `except` key as follows:

```
Route::resource(
   'news',
   'NewsController',
   array('except' => array('create', 'store', 'update',
     'destroy'))
);
```

5. All Resource Controller actions have defined route names. You may also want to override action names in some cases, and this can be done by setting up a third parameter called `names` as follows:

```
Route::resource(
   'news',
   'NewsController',
   array('names' => array('create' =>
     'news.myCoolCreateAction')
);
```

If you have gone through the previous chapter, you may remember we had filters in routes, but we didn't use the key `before` in RESTful and Resourceful Controllers. To use the key `before`, we can perform the steps we the previously followed or set filters in Controllers. These can be set in the `__construct()` method of the Controller (if there is none, create one) as follows:

```
class NewsController extends BaseController {

    public function __construct()
    {

        $this->beforeFilter('csrf', array('on' => 'post'));

        $this->beforeFilter(function(){
           //my custom filter codes in closure function
        });
```

```
            $this->afterFilter('log',
               array('only' => array('fooAction', 'barAction'))
            );
        }

    }
```

As you can see, we've set filters using the `beforeFilter()` and `afterFilter()` methods. These methods either get a closure function or the name of the filter as a first parameter and an optional second parameter, as an array, to define where these filters work.

In this example, we've firstly set CSRF (Cross-site Request Forgery, which is a method to forge a trusted request and to inject malicious code into the application through this forged request) protection filters to all the `POST` actions; after that, we've defined a filter in a closure function and used the `afterFilter` method to log all `fooAction` and `barAction` events' statuses.

The following table is a list of actions that Laravel's Resource Controller handles:

Verb	Path	Action	Route name
`GET`	`/resource`	Index	`resource.index`
`GET`	`/resource/create`	Create	`resource.create`
`POST`	`/resource`	Store	`resource.store`
`GET`	`/resource/{resource}`	Show	`resource.show`
`GET`	`/resource/{resource}/edit`	Edit	`resource.edit`
`PUT/PATCH`	`/resource/{resource}`	Update	`resource.update`
`DELETE`	`/resource/{resource}`	Destroy	`resource.destroy`

Using Controllers inside folders

In some cases, you may want to group Controllers inside a folder, and have them in a more hierarchical structure. There are two ways to achieve this.

Before explaining these methods, let's assume we have a `UserController.php` file inside an `app/controllers/admin` folder, which we've just created for admin-related Controller files. There is a question that crops up here: how do we make both Laravel and the Controller files understand where the Controllers are? Namespaces are used for such requirements. Namespaces are simple ways to encapsulate and group items.

Let's say you have the `app/controllers/UserController.php` and `app/controllers/admin/UserController.php` files. How do we call a specific one? Namespaces come in handy here. Save the following file as `app/controllers/admin/UserController.php`:

```
<?php namespace admin; //The definition of namespace

use View; //What will be used

Class UserController Extends \BaseController {

   public function index() {
      return View::make('hello');
   }

}
```

Now we define the route as follows:

```
Route::get('my/admin/users/index', 'Admin\UserController@index');
```

Here, we've added some new additions to this Controller. They are as follows:

- The first one is `namespace admin;`. This simply defines that this file is inside a folder called `admin`.
- The second one is `use View;`. If our Controller is under a folder or a defined `namespace` unless we import them, all the classes we'll be calling will be like `namespace\class`. If we didn't add this line, the `View::make()` function would throw an error saying `Class admin\View not found`. To understand this better, you can think of this like HTML's assets calling. Let's assume you are browsing `admin/users.html`, and inside it there is an image whose path is defined in this format: `<img src= "assets/img/avatar.png" />`. As you may imagine, the image will be requested from `admin/assets/img/avatar.png` because it is inside a folder called `assets`. This is exactly the same situation.
- We've added a `\` (backslash) character while we're extending from the `BaseController` class. This will signify that it will be called from root. If we didn't add `use View;` to our class and wanted to make `View::make()` work, we should modify it to `\View::make()` (with a leading backslash) so that the correct class will be requested.

If there is a totally new folder structure, it can be defined in two ways. Either add each folder path into the `autoload/classmap` object in the `composer.json` file or define a `psr-0` autoload. Let's assume we have a new `app/myApp` folder, and inside it is a Controller located at `app/myApp/controllers/admin/UserController.php`.

Add a `classmap` object to the Controller as follows:

```
"autoload": {
      "classmap": [
         "app/commands",
         "app/controllers",
          "app/models",
          "app/database/migrations",
          "app/database/seeds",
          "app/tests/TestCase.php",

          "app/myApp/controllers/admin",
      ]
}
```

Now add a `psr-0` autoload to the code as follows:

```
"autoload": {
      "classmap": [
         "app/commands",
         "app/controllers",
         "app/models",
         "app/database/migrations",
         "app/database/seeds",
         "app/tests/TestCase.php",

      ],

       "psr-0": {
         "myApp", "app/"
      }

}
```

Then run `composer dump-autoload` from the terminal to regenerate autoload classes. By this `psr-0` autoload, we've taught our Composer project to autoload everything recursively inside the `myApp` folder, which is inside the `app` folder. Another way is to prefix the classname with the namespace folders and use `underscores (_)` between each folder.

Let's assume we have a Controller, `app/controllers/foo/bar/BazController.php`. Save the following inside this folder:

```
<?php

Class foo_bar_BazController extends BaseController {

   public function index() {
      return View::make('hello');
   }

}
```

Now we define the route as follows:

```
Route::get('foobarbaz', ' foo_bar_BazController@index');
```

Then, we navigate to `http://yourwebsite/foobarbaz`. It will work automatically even without namespacing or including classes using `use`.

Summary

In this chapter, we learned the role of the Controller in the MVC pattern and how you can use Controllers and set routes in Laravel 4. We also learned about filters and RESTful and Resourceful Controllers.

For more information, you can refer to the official documentation page, located at `http://laravel.com/docs/controllers`. In the next chapter, we'll learn about Laravel's unique design pattern, and how it uses Repositories, Facades, and the Factory pattern.

5
Design Patterns in Laravel

In this chapter, we will discuss the design patterns Laravel uses, and how and why are they used, with examples.

The topics that will be discussed in this chapter are as follows:

- Design patterns used in Laravel
- The reasons these patterns are used in Laravel

The Builder (Manager) pattern

This design pattern aims to gain simpler, reusable objects. Its goal is to separate bigger and more convoluted object construction layers from the rest so that the separated layers can be used in different layers of the application.

The need for the Builder (Manager) pattern

In Laravel, the `AuthManager` class needs to create some secure elements to reuse with selected auth storage drivers such as cookie, session, or custom elements. To achieve this, the `AuthManager` class needs to use storage functions such as `callCustomCreator()` and `getDrivers()` from the `Manager` class.

Let's see how the Builder (Manager) pattern is used in Laravel. To see what happens in this pattern, navigate to the `vendor/Illuminate/Support/Manager.php` and `vendor/Illuminate/Auth/AuthManager.php` files, as shown in the following code:

```
    public function driver($driver = null)
    {
       ...

    }



    protected function createDriver($driver)
    {
       $method = 'create'.ucfirst($driver).'Driver';

       ...
    }

    protected function callCustomCreator($driver)
    {
       return $this->customCreators[$driver]($this->app);
    }


    public function extend($driver, Closure $callback)
    {
       $this->customCreators[$driver] = $callback;

       return $this;
    }
    public function getDrivers()
    {
       return $this->drivers;
    }


    public function __call($method, $parameters)
    {
       return call_user_func_array(array($this->driver(), $method),
$parameters);
    }
```

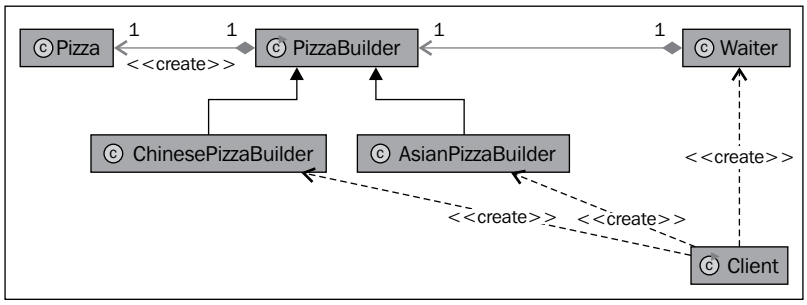
Now, navigate to the `/vendor/Illuminate/Auth/AuthManager.php` file, as shown in the following code:

```
protected function createDriver($driver)
{


   ....
}

protected function callCustomCreator($driver)
{


}

public function createDatabaseDriver()
{


}

protected function createDatabaseProvider()
{


   ....
}


public function createEloquentDriver()
{
   ...


}

protected function createEloquentProvider()
{
   ...

}
```

```
public function getDefaultDriver()
{
   ...
}

public function setDefaultDriver($name)
{
   ...
}
```

As we can see in the preceding code, the `AuthManager` class extends from the `Manager` class. Laravel ships with a basic auth mechanism. So, we need to store auth credentials in a database. First, the class checks our default database configuration with the `AuthManager::setDefaultDriver()` function. This function actually uses the `Manager` class for eloquent operations. All the database and auth options (such as cookie name) are obtained from the application's config file, except the auth model table name.

To understand this Builder (Manager) pattern better, we can take the following presentation as an example:



In the preceding example diagram, we assumed that we want to fetch data, for example, pizza, from the previous illustration. The client ordered two pizzas: an Asian pizza and/or a Chinese pizza. This pizza is requested through the `Waiter` class. The `PizzaBuilder` class (which is the `Manager` class, in our case) made a pizza according to the `AuthRequest` request, in our case, and delivered the pizza through the waiter.

Also, you can navigate to `vendor/Illuminate/Session/SessionManage.php` and check for the use of this pattern in Laravel Framework.

The Factory pattern

In this subsection, we'll examine the Factory pattern and its usage in Laravel Framework. The Factory pattern is based on creating template method objects, which is based on defining the algorithm of a class in a subclass to implement an algorithm. There is a subclass, which is derived from a big superclass, in this pattern structure. The main class, which we may call a superclass, only holds major and generic logic; the subclasses are derived from this superclass. As a result, there may be more than one subclass inherited from this superclass, which are aimed at different purposes.

Unlike other design patterns used in Laravel, the `Factory` method is more customizable. For an extended subclass plus main class, you don't need to set a new class, just a new operation. This method is beneficial if the class or its components usually change, or methods need to be overridden, much like initialization.

While creating a design, developers usually start with using the Factory pattern in their applications. Such a pattern is changed into an abstract Factory, Builder, or Prototype pattern. Unlike the Factory pattern, the Prototype pattern requires initialization once. Due to the pattern's architecture, the methods of the Factory pattern (Factory methods) are usually called inside template methods.

There are some differences between the Factory pattern and the Abstract Factory or Prototype pattern. They are as follows:

- Unlike an Abstract Factory pattern, the Factory pattern can't be implemented using the Prototype pattern.
- Unlike the Prototype pattern, the Factory pattern doesn't need an initialization, but it needs subclassing. This is an advantage when compared with other patterns. Thanks to this approach, the Factory pattern can return an injected subclass instead of an object.
- Since the classes designed with the Factory pattern may return subclasses directly for other components, no other class or component needs to know and access the constructor methods. Due to this, it's recommended that all constructor methods and variables should be protected or private.
- There is another thing to take into consideration. As this pattern might return subclasses aimed for the exact need, it's not recommended to make a new instance of the class using this pattern using the key new.

The need for the Factory pattern

Laravel ships various types of validation rules with the `Validation` class. When we develop applications, we usually need to validate data as we proceed. To do this, a common approach is to set the validation rules in the Model and call them from the Controller. By "rules" here, we mean both validation type and its range.

Sometimes, we need to set custom rules and custom error messages to validate the data. Let's examine how it works and how we are able to extend the `Validation` class to create custom rules. The Controller in the MVC pattern can also be described as a bridge between Model and View. This can best be explained with a live world example.

Let's assume we have a news aggregation website. In the administration panel, an administrator tries to delete the news item. In the SOLID design pattern, this happens if an admin clicks on the **Delete News** button.

First, as an example to check, let's open the `vendor/Illuminate/Validation/Factory.php` file, as shown in the following code:

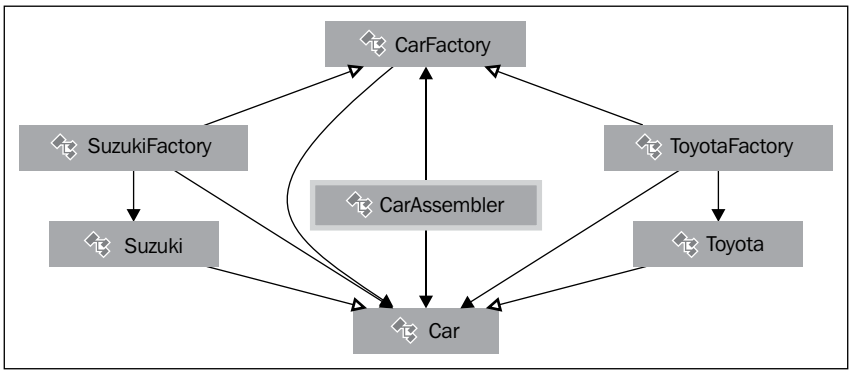
```
<?php namespace Illuminate\Validation;

use Closure;
use Illuminate\Container\Container;
use Symfony\Component\Translation\TranslatorInterface;

class Factory {


   protected $translator;

   protected $verifier;

   protected $container;

   protected $extensions = array();

   protected $implicitExtensions = array();

   protected $replacers = array();

   protected $fallbackMessages = array();

   protected $resolver;

   public function __construct(TranslatorInterface $translator,
     Container $container = null)
```

```
{
   $this->container = $container;
   $this->translator = $translator;
}


public function make(array $data, array $rules, array $messages
  = array(), array $customAttributes = array())
{

   $validator = $this->resolve($data, $rules, $messages,
     $customAttributes);

   if ( ! is_null($this->verifier))
   {
      $validator->setPresenceVerifier($this->verifier);
   }


   if ( ! is_null($this->container))
   {
      $validator->setContainer($this->container);
   }

   $this->addExtensions($validator);

   return $validator;
}


   protected function addExtensions(Validator $validator)
{
   $validator->addExtensions($this->extensions);

   $implicit = $this->implicitExtensions;

   $validator->addImplicitExtensions($implicit);

   $validator->addReplacers($this->replacers);

   $validator->setFallbackMessages($this->fallbackMessages);
}


protected function resolve(array $data, array $rules, array
  $messages, array $customAttributes)
{
```

```
        if (is_null($this->resolver))
        {
            return new Validator($this->translator, $data, $rules,
              $messages, $customAttributes);
        }
        else
        {
            return call_user_func($this->resolver, $this->translator,
              $data, $rules, $messages, $customAttributes);
        }
    }

    public function extend($rule, $extension, $message = null)
    {
        $this->extensions[$rule] = $extension;

        if ($message) $this->fallbackMessages[snake_case($rule)] =
          $message;
    }

    public function extendImplicit($rule, $extension, $message =
      null)
    {
        $this->implicitExtensions[$rule] = $extension;

        if ($message) $this->fallbackMessages[snake_case($rule)] =
          $message;
    }

    public function replacer($rule, $replacer)
    {
        $this->replacers[$rule] = $replacer;
    }

    public function resolver(Closure $resolver)
    {
        $this->resolver = $resolver;
    }

    public function getTranslator()
    {
        return $this->translator;
    }

    public function getPresenceVerifier()
    {
        return $this->verifier;
    }
```

```
    public function setPresenceVerifier(PresenceVerifierInterface
      $presenceVerifier
    {
       $this->verifier = $presenceVerifier;
    }

}
```

As we can see in the preceding code, the `Validation Factory` class is constructed with the `Translator` class and an IoC container. The `addExtensions()` function is set after this. This method includes the user-defined extensions to a `Validator` instance, thus allowing us to write the template (structure) to create the `Validator` class's extensions. The functions, which are public, allow us to implement the `Translator` class, and by this we mean that they allow us to write custom validation rules and messages. Refer to the following **CarFactory** diagram:



In the preceding diagram, you can see that all cars are based on **CarFactory** (the basics of all cars), regardless of the brand. For all brands, the main process is the same (all cars have an engine, tires, brakes, bulbs, gear, and so on). You may want either a **Suzuki** car or a **Toyota** car, and depending on this choice, the **SuzukiFactory** or **ToyotaFactory** creates a **Suzuki** car or a **Toyota** car from the **CarFactory**.

The Repository pattern

The Repository pattern is usually used to create an interface between two distinct layers of an application. In our case, the developers of Laravel use this pattern to create an abstract layer between `NamespaceItemResolver` (the class that resolves the namespaces and understands which file is in which namespace) and `Loader` (a class that requires and loads another class into the application). The `Loader` class simply loads the given namespace's configuration group. As you might know, nearly all of the Laravel Framework code is developed using namespaces.

The need for the Repository pattern

Let's assume you're trying to fetch a product from your database using Eloquent ORM. The method will be something like `Product::find(1)` in your Controller. For abstraction purposes, this approach is not ideal. If you now put a code such as this, your Controller knows you're using Eloquent, which ideally shouldn't happen in a good and abstracted structure. If you want to contain the changes done to the database scheme so that the calls outside of the class do not reference to the fields directly but through a repository, you have to dig all codes one by one.

Now, let's create an `imaginart` repository interface (a list of methods that will be used in the pattern) for the users. Let's call it `UserRepository.php`:

```
<?php namespace Arda\Storage\User;

interface UserRepository {

   public function all();

   public function get();

   public function create($input);

   public function update($input);

   public function delete($input);

   public function find($id);

}
```

Here, you can see that all the methods' names used in the Model are declared one by one. Now, let's create the repository and name it `EloquentUserRepository.php`:

```
<?php namespace Arda\Storage\User;

use User;

class EloquentUserRepository implements UserRepository {

  public function all()
  {
    return User::all();
  }

  public function get()
  {
```

```
        return User::get();
    }

    public function create($input)
    {
        return User::create($input);
    }

    public function update($input)
    {
        return User::update($input);
    }

    public function delete($input)
    {
        return User::delete($input);
    }

    public function find($input)
    {
        return User::find($input);
    }

}
```

As you can see, this repository class implemented our `UserRepository` that we created earlier. Now, you need to bind the two so that when the `UserRepositoryInterface` interface is called, we actually acquire `EloquentUserRepository`.

This can be done either with a service provider or by a simple command, such as the following, in Laravel:

```
App:bind(
    'Arda\Storage\User\UserRepository',
    'Arda\Storage\User\EloquentUserRepository'
);
```

Now, in your Controllers, you can simply use the repositories as `Use Arda\Storage\User\UserRepository as User`.

Every time the controller uses a `User::find($id)` code, it first goes to the interface, and then goes to the binded repository, which is the Eloquent repository in our case. Through this, it goes to the Eloquent ORM. This way, it's impossible for the Controller to know how the data is fetched.

The Strategy pattern

The best approach to describe the Strategy pattern is through a problem.

The need for the Strategy pattern

In this design pattern, the logic is extracted from complex classes into easier components so that they can be replaced easily with simpler methods. For example, you want to show popular blog posts on your website. In a classic approach, you will calculate the popularity, make the pagination, and list the items relative to the current paginated offset and popularity, and make all calculations in a simple class. This pattern aims to separate each algorithm into separate components so that they can be reused or combined in other parts of the application easily. This approach also brings flexibility and makes it easy to change an algorithm system wide.

To understand this better, let's take a look at the following loader interface located at `vendor/Illuminate/Config/LoaderInterface`:

```
<?php namespace Illuminate\Config;

interface LoaderInterface {

   public function load($environment, $group, $namespace = null);

   public function exists($group, $namespace = null);

    public function addNamespace($namespace, $hint);

   public function getNamespaces();

   public function cascadePackage($environment, $package, $group,
$items);

}
```

When we dig the code, the `LoaderInterface` works will follow a certain structure. The `getNamespaces()` function loads all namespaces defined in the `app\config\app.php` file. The `addNamespace()` method passes the namespaces to the `load()` function as grouped. If the `exist()` function returns `true`, there is at least one configuration group that belongs to a given namespace. For the full structure, you can refer to the repository section of this chapter. As a result, you can easily call the method that you need through an interface of the `Loader` class to load various configuration options. If we download a package through the composer, or implement a package to an application that is being authored, the pattern makes all of them available and loads them from their own namespaces without any conflicts, though they are inside different namespaces or have the same filenames.

The Provider pattern

The Provider pattern was formulated by Microsoft for use in the ASP.NET Starter Kits and formalized in .NET Version 2.0 (`http://en.wikipedia.org/wiki/Provider_model`). It is a mid layer between an API class and the Business Logic/Data Abstraction Layer of the application. The provider is the implementation of the API separated from the API itself.

This pattern, its aims, and its usage are quite similar to the Strategy pattern. This is why many developers are already discussing whether to accept this approach as a design pattern.

To understand these patterns better, let's open `vendor/Illuminate/Auth/AuthServiceProvider.php` and `vendor/Illuminate/Hashing/HashServiceProvider.php`:

```
<?php namespace Illuminate\Auth;

use Illuminate\Support\ServiceProvider;

class AuthServiceProvider extends ServiceProvider {

   protected $defer = true;

   public function register()
   {
      $this->app->bindShared('auth', function($app)
      {
           // Once the authentication service has actually been
             requested by the developer
          // we will set a variable in the application indicating
            this, which helps us
```

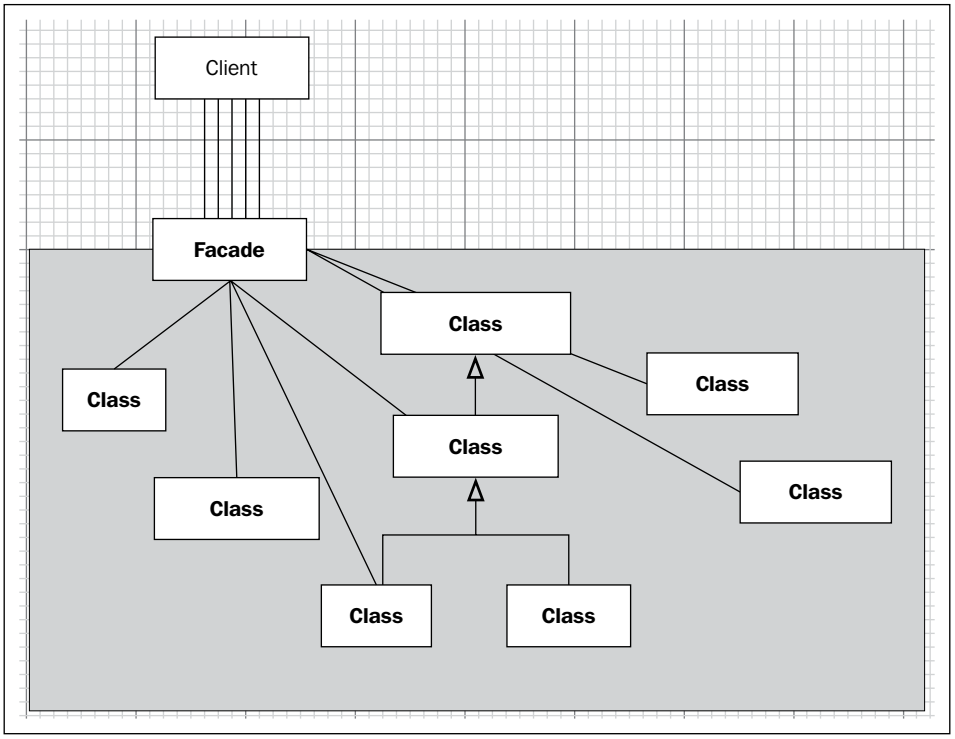
```
            // to know that we need to set any queued cookies in the
              after event later.
            $app['auth.loaded'] = true;

            return new AuthManager($app);
        });
    }

    public function provides()
    {
        return array('auth');
    }

}


<?php namespace Illuminate\Hashing;

use Illuminate\Support\ServiceProvider;

class HashServiceProvider extends ServiceProvider {

    protected $defer = true;

    public function register()
    {
        $this->app->bindShared('hash', function() { return new
          BcryptHasher; });
    }

    public function provides()
    {
        return array('hash');
    }

}
```

As you can see, both the classes extend `ServiceProvider`. The `AuthServiceProvider` class allows us to provide all services to `AuthManager` when an authentication request, such as checking whether a cookie and session is created or whether the content is invalid, is sent. After the authentication service has been requested, the developer can define whether a session or cookie is set through the response through `AuthDriver`.

However, `HashServiceProvider` provides us with the related methods when a secure hash request is done so that we can use, fetch, check, or do other things with these hashes. Both providers return the values as an array.

The Facade pattern

The Facade (façade) pattern allows a developer to unite various complicated interfaces into a single class interface. This pattern also allows you to wrap various methods from various classes into a single structure.



In Laravel 4, as you may already know, almost every method looks like a static method, for example, `Input::has()`, `Config::get()`, `URL::route()`, `View::make()`, and `HTML::style()`. However, they are not static methods. If they were static methods, it would be quite hard to make tests for them all. They are actually the mimics of this behavior. In the background, with the help of the IoC Container (a way to inject dependencies into a class), Laravel actually calls another class(es) through a `Facade` class. The Facade base class benefits from PHP's own `__callStatic()` magic method to call the required methods, such as static methods.

For example, let's assume we have a method called `URL::to('home')`. Let's check what the URL is and what it refers to. First, let's open `app/config/app.php`. In the aliases array, there is a line like the following:

```
'URL' => 'Illuminate\Support\Facades\URL',
```

So, if we call `URL::to('home')`, we actually call `Illuminate\Support\Facades \ URL::to('home')`.

Now, let's check what's inside the file. Open the `vendor/Illuminate/Support/Facades/URL.php` file:

```
<?php namespace Illuminate\Support\Facades;

class URL extends Facade {


   protected static function getFacadeAccessor() { return 'url'; }

}
```

As you can see, the class actually extends from the `Facade` class, and there is no static method called `to()`. Instead, there is a method called `getFacadeAccessor()`, which returns the string `url`. The `getFacadeAccessor()` method's purpose is to define what to inject. This way, Laravel understands that this class is looking for `$app['url']`.

This is defined in `vendor/Illuminate/Routing/RoutingServiceProvider.php`, as follows:

```
protected function registerUrlGenerator()
{
   $this->app['url'] = $this->app->share(function($app)
      {

      $routes = $app['router']->getRoutes();

      return new UrlGenerator($routes, $app->rebinding('request',
        function($app, $request)
      {
         $app['url']->setRequest($request);
      }));
   });
}
```

As you can see, it returns a new instance of the `UrlGenerator` class in the same namespace, which holds the `to()` method we're looking for:

```
//Illuminate/Routing/UrlGenerator.php
public function to($path, $extra = array(), $secure = null)
{
   //...
}
```

So each time you use a method like this, Laravel first goes to and checks the facade, it then checks what's injected through, and then the real method through the `injected` class is called.

Summary

In this chapter, we learned about various design pattern uses in the Laravel PHP framework, how and why they are used, and what problems they can solve.

In the next chapter, we'll learn about best practices to create an application using Laravel using the design patterns in our code in a Laravel project.

6
Best Practices in Laravel

In this chapter, we will see examples of various previously-described design patterns used in Laravel.

The topics that will be discussed in this chapter are as follows:

- Basic and advanced practices
- Real-life examples of design patterns used in Laravel
- The reasons why these design patterns are used in the examples

Basic practices

As a developer, when you are working on an application, there should be a systematic order to things to prevent confusion and allow flexibility. For example, in an MVC architecture, Controller should only hold the logic and Model should only hold dataflow-related stuff. You should not write database queries in View files. This way, anyone working on the project can find what they are looking for easily and can change, fork, or improve it with greater ease. If this is not followed, the project will turn into a mess as it gets bigger and bigger.

A basic good practice would be to avoid repeating yourself. If you're using a code snippet or a condition a number of times, it'd be better for you to prepare a method or a scope for that action. This way, you wouldn't have to repeat yourself over and over. For example, let's say we have an imaginary Controller as follows:

```
<?php

class UserController extends BaseController {

   //An imaginary method that lists all active users
   public function listUsers() {
```

```
        $users = User::where('active', 1)->get();

        return View::make('frontend.users.list')
            ->with('users', $users);
    }


    //An imaginary method that finds a specific user
    public function fetch($id) {

        $user = User::where('active', 1)->find($id);

         return View::make('frontend.users.single')
           ->with('user', $user);

    }

}
```

As you can see, the `where()` condition checks if `active` is repeated twice. In real-world examples, it would be used even more.

To prevent this, in Laravel, you can use query scopes. Query scopes are single functions that help you reuse the logic in Models. Let's define a query scope in Model and change the Controller method as follows:

```
<?php

//Model File
Class User extends Eloquent {

    //We've defined a Query scope called active
    public function scopeActive($query) {
        return $query->where('active', 1);
    }

}




//Controller File
class UserController extends BaseController {

    //An imaginary method that lists all active users
    public function listUsers() {

        $users = User::active()->get();
```

```
        return View::make('frontend.users.list')
            ->with('users', $users);
    }


    //An imaginary method that finds a specific user
    public function fetch($id) {

        $user = User::active()->find($id);

        return View::make('frontend.users.single')
            ->with('user', $user);

    }

}
```

As you can see, we've defined a method called `scopeActive()` in Model, which is prefixed with the word `scope` and CamelCased. This way, Laravel can understand that it's a query scope, and you can use that scope directly. As you can see, the conditions in the Controller have also changed. They have changed from `where('active', 1)` to `active()`.

Design patterns are advanced practices and can be used to keep the code tidy and systematic using various approaches.

Advanced practices

In this subsection, we will see various design patterns' usage in Laravel. If you test the custom classes that include the design patterns, which are provided within the book, they should be autoloaded in your application. This can be done either by adding them to the `ClassLoader::addDirectories()` array of the `global.php` file (which can be found by navigating to `app/start`) or the `start.php` file in the `bootstrap` folder. Alternatively, we can add a `psr-0` autoload in `composer.json`.

To add directories from `app/start/global.php`, first find the following code:

```
ClassLoader::addDirectories(array(

    app_path().'/commands',
    app_path().'/controllers',
    app_path().'/models',
    app_path().'/database/seeds',

));
```

Then add your folders below. The resulting code will look as follows:

```
ClassLoader::addDirectories(array(

   app_path().'/commands',
   app_path().'/controllers',
   app_path().'/models',
   app_path().'/database/seeds',

   //our custom directory that holds classes
   app_path().'/acme',
));
```

If you want to autoload classes or files from the `composer.json` file using the `psr-0` autoload, you have to add the namespace and directory into `composer.json`. The key will be the namespace and the value will be the path of the folder that holds the files and classes to be autoloaded. Have a look at the following code:

```
"autoload": {

    "psr-0": {
        "Acme": "app/lib"
    }
}
```

In this example, if our `composer.json` file doesn't have a `psr-0` object, first we'll create it and then add the namespace and the path values inside. You can see we have a namespace called `Acme`, which is under the folder `app/lib`.

If you don't want to autoload a whole folder but only a few single files, you can also use the `files` object in `composer.json`. It's a single object that only holds the paths of files.

```
"autoload": {
    "files": [
        "app/acme/myFunctions.php"
    ]
},
```

After adding these values, you need to dump the autoload files and make Laravel understand them. To do this, after editing the `composer.json` file, simply run the following command:

```
composer dump-autoload
```

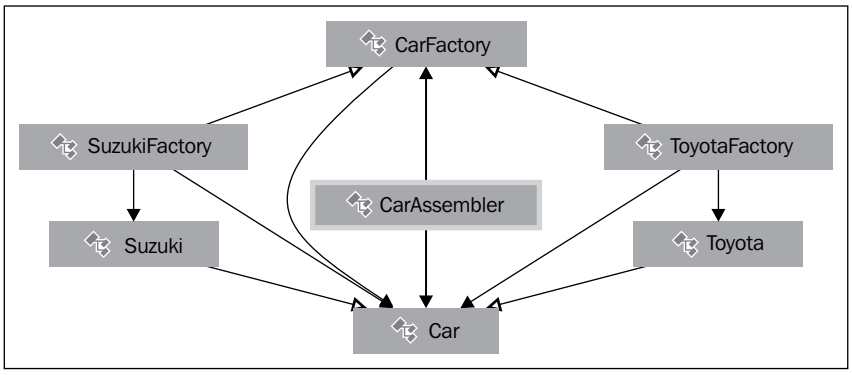
You can also run the following command if the composer is not installed in your environment:

```
php composer.phar dump-autoload
```

After this, the classes or files that you've just added will be autoloaded and available for your project.

The Factory pattern

As you might recollect from *Chapter 5*, *Design Patterns in Laravel*, the Factory pattern is based on creating template method objects to implement an algorithm. Let's assume we are developing the following application:



Let's assume the **Toyota** brand only produces C-class red cars, and the **Suzuki** brand only produces B-class green cars. Let's assume we have a Model for this purpose that is defined as follows:

```
<?php

class CarFactoryModel extends BaseModel{

    public static function createCar($manufacturer)
    {
        switch ($manufacturer)
        {


        }
```

```
            throw new \InvalidArgumentException("Unsupported manufacturer
    [$manufacturer]");
        }

        public static function createCarFromColor($color)
        {
            switch ($color)
            {
                case 'Red':

                return static::createCar('Toyota');

                case 'Green':

                return static::createCar('Suzuki');

            }

            throw new \InvalidArgumentException("Unsupported color
    [$color]");
        }

        public static function createCarFromClass($class)
        {
            switch ($class)
            {
                case 'B':

                return static::createCar('Suzuki');

                case 'C':

                return static::createCar('Toyota');
            }

            throw new \InvalidArgumentException("Unsupported car class
    [$class]");

        }
    }
```

As you can see, in this approach, the same `carFactory()` class is called for both `Suzuki` and `Toyota` because in this example both brands would have the same processes to create the core of the car. The quality-class and color are set after the core of the car is produced. After this is set, for the color and quality-class choices, we can directly call the corresponding class with its brand. Let's say we are going to buy a B-class car. Now, because the code knows which brand produces B-class cars, it will directly call `Suzuki`. This Model can have a Controller like the one seen in the following code:

```
<?php

class CarController extends BaseController{

    public function showCarsByManufacturer($manufacturerName){

        return CarFactory::createCar($manufacturerName);


    }

    public function showCarsByColor($color){

        return CarFactory::createCarFromColor($color);


    }


    public function showCarsByClass($className){

        return CarFactory::createCarFromClass($className);

    }

}
```

The three different routes of this approach are as follows:

```
Route::get(
'cars/{manufacturer}',
array(
'as'      => 'cars_by_manufacturer',
'uses' => 'CarController@showCarsbyManufacturer'
)
);
```

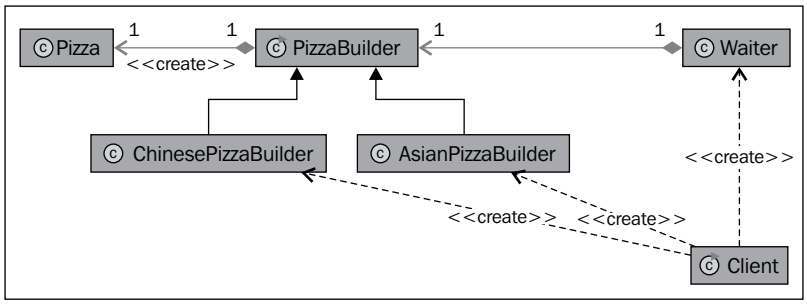
```
Route::get(
'cars/color/{color}',
array(
'as' => 'cars_by_color',
'uses' => 'CarController@showCarsbyColor'
)
);

Route::get(
'cars/class/{class}',
array(
'as' => 'cars_by_class',
'uses' => 'CarController@showCarsbyClass'
)
);
```

In such situations, in addition to the Factory pattern, having an approach like these three routes would be a good practice for URL richness, ease of use, and improved search engine optimization of the website.

The Builder pattern

We discussed in *Chapter 5*, *Design Patterns in Laravel*, that, in a way, the Builder pattern is an approach to separate bigger objects into smaller ones and make them available for reuse.

In this subsection, as in the example in *Chapter 5*, *Design Patterns in Laravel*, let's assume we are baking a pizza that has certain properties, such as Italian and small/big:



Let's assume we have an autoloaded class as follows:

```
<?php

class PizzaDelivery
{
```

```
protected $pizza;


protected $config = array();


public function __construct(array $config)
{
    $this->pizza = new PizzaBuilder();
    $this->setConfig($config);
}

/**
* Process some configuration parameters
*
* @param array $config
*/
protected function setConfig(array $config)
{
    $defaults = array(
        'spice' => true,
        'type' => 'Italian',
        'size' => 'Small',
    );

    $config =  array_replace($defaults, $config);
    $this->config = $config;
}

/**
* Build the pizza using the supplied configuration parameters
* From the constructor that is set using setConfig() method.
*

* @return null
*/
public function build()
{
    foreach ($this->config as $option => $value) {
        $method = sprintf('set%s', ucfirst($option));
        if (method_exists($this->pizza, $method) === true) {
            call_user_func(array($this->pizza, $method),
              $value);
        }
```

```
            }
        }

        /**
        * @return Pizza
        */
        public function getPizza()
        {
            return $this->pizza;
        }
    }
```

As you can see, we have injected a builder, the `PizzaBuilder` class, which includes two chefs: one is an Italian pizza maker and the other is an Asian pizza maker. The `PizzaBuilder` class in this approach can be coded as follows:

```
<?php
class PizzaBuilder
{
    protected $type = '';
    protected $size = '';
    protected $spice = '';
    public function setSpice($spice)
    {
        $this->spice = $spice;
    }


    public function setSize($size)
    {
        $this->size = $size;
    }


    public function setType($type)
    {
        $this->type = $type;
    }
}
```

As you can see, it holds all the basic stuff to bake a pizza, but the properties are defined outside the class, through the Model using methods such as `setType()` and `setSize()`. With this approach, by only defining the properties and without thinking about the rest, we can build and get our pizza directly from the waiter (`PizzaDelivery`). If we need to get an Asian pizza, we'd call the following code anywhere in our application:

```
$myFavoritePizza = new PizzaDelivery(array('type' => 'Asian'));
$myFavoritePizza ->build();
return $myFavoritePizza->get();
```

The Strategy pattern

As you might recall from *Chapter 5*, *Design Patterns in Laravel*, the Strategy pattern is used to divide the logic into smaller parts according to their tasks so that these parts can be reused.

In the sample application we'll code for this approach, we will make a package shipment calculation application for different package carriers. Let's assume we have a class like the following:

```
<?php

interface ShipmentPricingStrategy {

    function shipmentPrice();

}

abstract class ShippingPriceStrategy implements
ShipmentPricingStrategy {
    function __construct() {}
    abstract function shipmentPrice();
}

class FedexPriceStrategy extends ShippingPriceStrategy {

    function shipmentPrice() {
        return 4.95;
    }
}

class UpsPriceStrategy extends ShippingPriceStrategy {
```

```
        function shipmentPrice() {
            return 3.75;
        }
    }

    class Shipping {

        public $shipping_pricing_structure;

        function __construct(ShippingPriceStrategy $shipment_pricing_
    strategy) {

            $this->shipping_pricing_structure = $shipment_pricing_
    strategy;

        }
    }
```

As you can see in the code, in the `Shipping` class, we have injected `ShippingPriceStrategy`. This Strategy pattern has features for each carrier (`shipmentPrice()` in our case). With this approach, for different shipping carriers, we can show different delivery prices and include them in our shipping process, which will be defined in our `Shipping` class. This way, we've used the prices set in the Strategy pattern both when showing the shipping price and counting a sum total for the shipping process.

By assuming that the class is autoloaded, we can use the class that has the Strategy pattern in an example like this:

```
    <?php

    $cart_total = 77.90;

    $fedex_price = new Shipping(new FedexPriceStrategy());

    $ups_price = new Shipping(new UpsPriceStrategy());

    $ups_price_with_cart = $cart_total+$ups_price->shipping_pricing_
    structure->shipmentPrice();

    $fedex_price_with_cart = $cart_total+$fedex_price->shipping_pricing_
    structure->shipmentPrice();

    echo 'The cost of this order with Fedex is: '.$fedex_price_with_
    cart."\n";

    echo 'The cost of this order with UPS is: '.$ups_price_with_cart."\n";
```

As you can see, for the same shipping process (the `Shipping` class), by injecting different shipping strategies for the two brands, we've managed to gain different shipping prices due to the difference in their strategies.

The Repository pattern

The main reason to use a Repository pattern is to provide abstraction and flexibility. For example, let's say you are fetching a product from the database. By default, in Laravel, the usual way is to use Eloquent ORM in Controller and to pass it to the View. This way, your Controller knows that you are using Eloquent ORM to fetch data from the data source / database. For small applications, this should be no problem, but in bigger applications, an issue might occur. In future, for some reason, you might want to drop MySQL using Eloquent ORM and might need to use another ORM in MongoDB. When this happens, because the Controller knows that you are using Eloquent ORM, you'll have to dig each Controller (or any other layer) one by one and change them. Another limitation is that you cannot unit test this code.

This would not happen if you had used repositories. If you do this, the Controller would connect only with the repository, and the repository would handle the other regarding layers. Therefore, the Controller wouldn't know how the data was fetched (abstraction). This way, on bigger applications, managing stuff or testing stuff should be much easier.

To understand this approach, first let's assume we have the `ProductsController` and `Product` models, and we want to fetch a product with a given ID and another method to dump all products. The Controller would look something like this:

```
<?php

Class ProductsController extends \BaseController {

   public function findProduct($id) {
      $product = Product::find($id);
      return View::make('product')
         ->with('product', $product);
   }

   public function allProducts() {
      $products = Product::all();
      return View::make('all_products')
         ->with('products', $products);
   }

}
```

There is a flaw in this approach. If you are testing code that is written like this and there is an error, you can't directly detect what the source of the error is unless `Whoops` (the error handler library used in Laravel) is active. Repositories are helpful in situations like this because they extract the logic. To inject a repository, one way is to define and set it in a constructor method of the Controller.

Let's name this repository `EloquentProductRepository`, which is part of the `\Acme\Repositories` namespace. Our Controller would change into something like the following:

```
<?php

//We use the repository in our class
use Acme\Repositories\EloquentProductRepository;

Class ProductsController extends \BaseController {

   //A protected variable to hold the Repository
   protected $product;

   //Let's define a constructor class, and assigning to the
     variable $product
   public function __construct(EloquentProductRepository $product)
   {
      $this->product = $product;
   }

   public function findProduct($id) {
      //$product = Product::find($id);
      $product = $this->product->find($id);
      return View::make('product')
         ->with('product', $product);
   }

   public function allProducts() {
      //$products = Product::orderBy('id', 'desc')->get();
      //let's give it a unique method name
      $products = $this->product->getNewest();
      return View::make('all_products')
         ->with('products', $products);
   }

}
```

Note that instead of making `orderBy('id', 'desc')->get()`, we have given a new method name, `getNewest()`. Now let's create this repository. Let's assume we have a file called `EloquentProductRepository.php` inside the `namespace Acme\Repositories` folder. Have a look at the following code:

```
<?php namespace Acme\Repositories;

Class EloquentProductRepository {


   public function getNewest() {

      return \Order::orderBy('id', 'desc')->get();

   }

   public function find($id) {
      return \Order::find($id);
   }

}
```

For each method that is used, you need to define functions once you are in the repositories. A major advantage of this approach is that it brings flexibility. Let's say you will be using mocks, or you'll switch from Eloquent ORM to another one in future that has totally different method names. To switch your application's database layer from Eloquent to MongoDB, if you've used repositories, you only need to change the used repository in your Controllers, nothing else. You won't need to dig all of your Controllers, Models, or other components.

There is still a feature lacking here. Our Controller still knows that we are using an Eloquent-specific repository. For a better approach and abstraction, our Controller should not know what kind of repository we are using. To ensure this, we will have to code an interface for this.

Now, let's create an interface in the same namespace path (it is not forced; you may create it anywhere as long as it's loaded) as `ProductInterface`. Our Controller would then look like this:

```
<?php

//We use the interface in our class
use Acme\Repositories\ProductInterface;

Class ProductsController extends \BaseController {
```

```
    //A protected variable to hold the Interface
    protected $product;

    //Let's define a constructor class, and inject the interface as
      $product variable
    public function __construct(ProductInterface $product) {
       $this->product = $product;
    }

    public function findProduct($id) {
       //$product = Product::find($id);
       $product = $this->product->find($id);
       return View::make('product')
          ->with('product', $product);
    }

    public function allProducts() {
       //$products = Product::orderBy('id', 'desc')->get();
       //let's give it a unique method name
       $products = $this->product->getNewest();
       return View::make('all_products')
          ->with('products', $products);
    }

}
```

Instead of the repository, the interface is injected and used. Now let's code the `ProductInterface` interface:

```
<?php namespace Acme\Repositories;

interface ProductInterface {

   public function getNewest();
   public function find();

}
```

As you can see, the interface holds the method names, which are actually the methods available to the implemented repository. Now let's implement this interface to our repository to connect them:

```
<?php namespace Acme\Repositories;

Class EloquentProductRepository implements ProductInterface {


   public function getNewest() {
```

```
        return \Order::orderBy('id', 'desc')->get();

    }

    public function find($id) {
        return \Order::find($id);
    }

}
```

This implementation has an advantage. Let's assume that you've implemented an interface in the repository and it is missing the `getNewest()` custom method. Thanks to this implementation, the interface will directly let you know that it needs that specific method and it's missing.

Lastly, we need to bind the interface to the repository. One of the ways to do this is to use Laravel's built-in `App:bind();` method. To bind the repository that we've just created to the interface, add this line into your `app/routes.php` file or any other file that's autoloaded.

```
App::bind(
'Acme\Repositories\ProductInterface',
'Acme\Repositories\EloquentProductRepository'
);
```

Another way to bind these two is to create a service provider. Let's write a service provider as follows:

```
<?php namespace Acme\Repositories;

use Illuminate\Support\ServiceProvider;

class UserServiceProvider extends ServiceProvider {

    public function register()
    {
        $this->app::bind('Acme\\Repositories\\ProductInterface',
'Acme\Repositories\EloquentProductRepository');
    }
}
```

We've assumed that this provider is in the `namespace Acme\Repositories` folder. We've also used `Illuminate\Support\ServiceProvider` to extend our class from it. In the public method register, we've binded the interface to our product repository.

In future, if you want to switch to MongoDB or any other interface that you've coded, all you have to do is switch `EloquentRepositoryInterface` to the new one. It will be updated everywhere in the application.

Summary

In this chapter, we saw examples of basic and advanced practices of design patterns and architectures that are used both in Laravel and in general development processes. We have learned the advantages of various design patterns while citing real-world examples for each one of them.

Design patterns are there to make your life easier. If development is done without following any pattern or architecture, as the application grows, both refactoring and implementing features would be harder after each refactoring. Also, if another developer joins the project, he or she first needs to understand what's where. This will possibly cause bloating, bad performance, inflexibility, and a variety of errors that are hard to fix. The application would be a time-bomb ready to explode. Design and architectural patterns are there to help you prevent these issues. Not only in your Laravel application, but in anything that you're developing, as the application grows, to keep everything under control, you must use a design pattern or a combination of them. In the end, there will be a day you'll thank yourself for using these patterns.

Index

F

H

L

M

O

P

Q

R

S

T

U

V

Y

Thank you for buying
Laravel Design Patterns and Best Practices

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

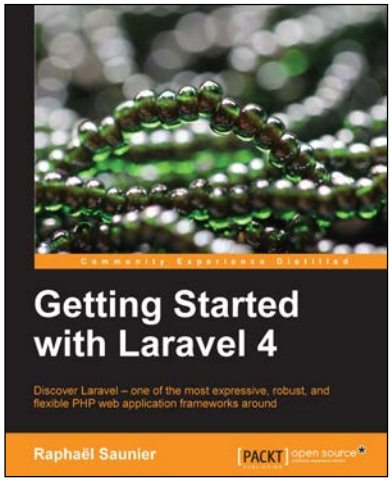
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

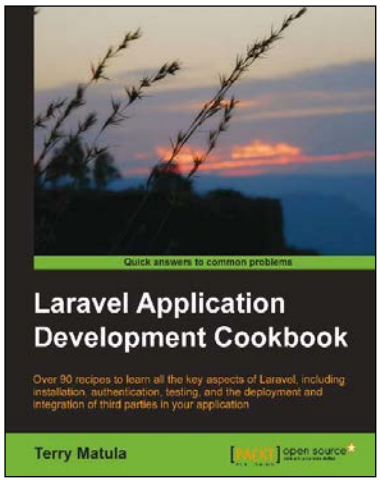


Getting Started with Laravel 4

ISBN: 978-1-78328-703-1 Paperback: 128 pages

Discover Laravel – one of the most expressive, robust, and flexible PHP web application frameworks around

1. Provides a concise introduction to all the concepts needed to get started with Laravel.
2. Walks through the different steps involved in creating a complete Laravel application.
3. Gives an overview of Laravel's advanced features that can be used when applications grow in complexity.
4. Learn how to build structured, more maintainable, and more secure applications with less code by using Laravel.

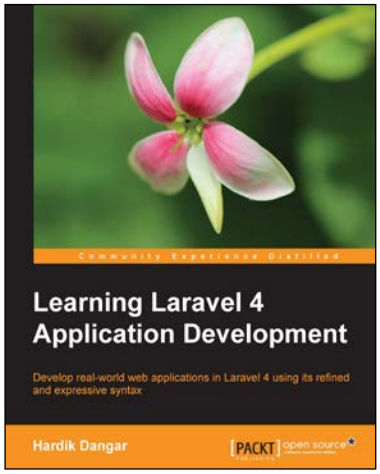


Laravel Application Development Cookbook

ISBN: 978-1-78216-282-7 Paperback: 272 pages

Over 90 recipes to learn all the key aspects of Laravel, including installation, authentication, testing, and the deployment and integration of third parties in your application

1. Install and set up a Laravel application and then deploy and integrate third parties in your application.
2. Create a secure authentication system and build a RESTful API.
3. Build your own Composer Package and incorporate JavaScript and AJAX methods into Laravel.

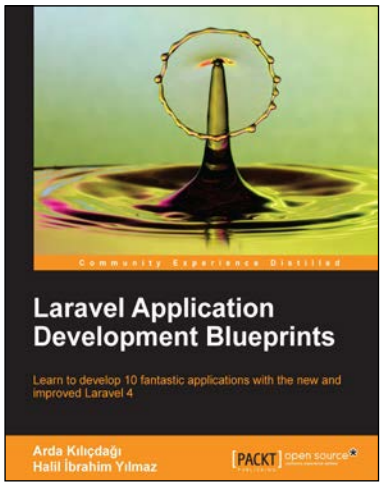


Learning Laravel 4 Application Development

ISBN: 978-1-78328-057-5 Paperback: 256 pages

Develop real-world web applications in Laravel 4 using its refined and expressive syntax

1. Build real-world web applications using the Laravel 4 framework.
2. Learn how to configure, optimize, and deploy Laravel 4 applications.
3. Packed with illustrations along with lots of tips and tricks to help you learn more about one of the most exciting PHP frameworks around.



Laravel Application Development Blueprints

ISBN: 978-1-78328-211-1 Paperback: 260 pages

Learn to develop 10 fantastic applications with the new and improved Laravel 4

1. Learn how to integrate third-party scripts and libraries into your application.
2. With different techniques, learn how to adapt different methods to your needs.
3. Expand your knowledge of Laravel 4 so you can tailor the sample solutions to your requirements.