



Laravel: My First Framework

by Maksim Surguy

Laravel - my first framework

Companion for developers discovering Laravel PHP framework

Maksim Surguy

This book is for sale at <http://leanpub.com/laravel-first-framework>

This version was published on 2014-09-05



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 Maksim Surguy

Tweet This Book!

Please help Maksim Surguy by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#laravelfirst](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

<https://twitter.com/search?q=#laravelfirst>

Also By Maksim Surguy

Integrating Front end Components with Web Applications

Contents

Introduction	i
About the author	i
Prerequisites	ii
Source Code	ii
1. Meeting Laravel	1
1.1 Introducing Laravel 4 PHP framework	1
1.1.1 Laravel's Expressive code	2
1.1.2 Laravel applications use Model-View-Controller pattern	3
1.1.3 Laravel was built by a great community	3
1.2 History of Laravel framework	4
1.2.1 State of PHP frameworks world before Laravel 4	4
1.2.2 Evolution of Laravel framework	4
1.3 Advantages of Using Laravel	5
1.3.1 Convention over configuration	5
1.3.2 Ready out of the box	6
1.3.3 Clear organization of all parts of the application	7
1.3.4 Built-in Authentication	7
1.3.5 Eloquent ORM – Laravel's Active Record implementation	9
1.4 Summary	10
2. Getting started for Development with Laravel	12
2.1 Development overview	12
2.2 Meeting Composer	13
2.3 Installing Laravel	13
2.4 Meeting Artisan: Laravel's command line interface	14
2.5 Application structure overview	14
2.5.1 Files and folders in the root level of the application	15
2.5.2 Contents of the "app" folder	16
2.5.3 Configuration settings	17
2.6 Tutorial: building a website with Laravel	17
2.6.1 The big picture – development overview	18
2.6.2 Creating a new Laravel project	19
2.6.3 Configuring application settings	19
2.6.4 Specifying endpoints (routes) of the website	21
2.6.5 Creating the database structure and the model for the data	22

CONTENTS

2.6.6	Filling the database with data	25
2.6.7	Creating HTML and Blade views	26
2.6.7.1	Building the layout	27
2.6.7.2	Building the template for the home page	28
2.6.7.3	Building the template for the services page	28
2.6.7.4	Building the template for the contact page	29
2.6.8	Displaying view templates from the routes	30
2.6.8.1	Connecting the home page route to the view template	30
2.6.8.2	Connecting the services page route to the view template	31
2.6.8.3	Connecting the contact page route to the view template	32
2.6.9	Adding validation to the contact form	33
2.6.10	Sending HTML email with Laravel	34
2.7	Summary	37
3.	Routing	38
3.1	Introduction to Routing in Laravel	38
3.1.1	Structure of a basic route	39
3.1.2	Types of route requests	41
3.1.3	Routing example: login form	42
3.1.3.1	Building routing structure for the login form example	42
3.1.3.2	Showing HTML form and processing user input	43
3.2	Passing parameters to routes	44
3.2.1	Using route parameters	46
3.2.1.1	Passing more than one parameter	46
3.2.2	Route constraints	46
3.2.3	Making route parameters optional	47
3.3	Route Filters	48
3.3.1	Attaching filters to routes	49
3.3.2	Creating custom filters	50
3.3.3	Adding multiple route filters	52
3.4	Grouping routes	52
3.5	Route responses	53
3.5.1	Showing output	54
3.5.2	Redirects	56
3.5.2.1	Redirecting to other URLs inside of the application	56
3.5.2.2	Redirecting to named routes	57
3.6	Summary	57
4.	Views	58
4.1	Introducing views and templates	58
4.1.1	Views - application's response to a request	60
4.1.2	Templates in Laravel, Blade	61
4.2	Creating and organizing views	62
4.2.0.1	Separating view templates in folders	63
4.3	Passing data to views	64
4.3.1	Using "View::make" arguments to pass data	66

CONTENTS

4.3.2	Using method “with”	67
4.4	Blade template engine	69
4.4.1	Outputting and escaping data	70
4.4.1.1	Escaping Data	71
4.4.2	Using conditional statements and loops	71
4.4.2.1	Using conditional statements in the views with plain php	72
4.4.2.2	Using conditional statements in the views with blade	73
4.4.2.3	Using loops	74
4.5	Blade layouts	76
4.5.1	Creating and using a layout	77
4.5.1.1	Using layouts from views	78
4.5.1.2	Using method “@yield” to specify placeholders	78
4.5.2	Using sections	79
4.5.3	Nesting views by using @include	82
4.6	Summary	83
5.	Understanding Controllers	84
5.1	Introducing controllers, the “C” in MVC	84
5.2	Default Controllers: BaseController and HomeController	86
5.2.1	Base controller (app/controllers/BaseController.php)	87
5.2.2	Home controller (app/controllers/HomeController.php)	88
5.3	Creating controllers	88
5.3.1	Creating a new controller: LoginController	89
5.3.2	Creating logic for the LoginController	90
5.3.3	Defining routing for LoginController	90
5.4	Using Basic, RESTful and Resource controllers	94
5.4.1	Creating and using Basic Controllers	95
5.4.2	Creating and using RESTful Controllers	96
5.4.3	Creating and using Resource Controllers	98
5.4.3.1	Creating resource controller with artisan command	98
5.5	Using controllers with routes	100
5.5.1	Routing to Basic Controllers	100
5.5.2	Routing to RESTful Controllers	102
5.5.3	Routing to Resource Controllers	104
5.6	Passing route parameters to controllers	105
5.6.1	Passing parameters to methods of a Basic Controller	106
5.6.2	Passing parameters to methods of a RESTful Controller	108
5.6.3	Passing parameters to methods of a Resource Controller	109
5.7	Using filters with controllers	111
5.7.1	Attaching filters to a controller	112
5.7.2	Applying filters to specific methods of a controller	113
5.7.2.1	Using the “on” keyword	113
5.7.2.2	Using the “except” and “only” keywords	113
5.8	Summary	114
6.	Database operations	115

CONTENTS

6.1	Introducing Database Operations in Laravel	115
6.1.1	Configuring database settings	117
6.1.2	Introducing Query Builder & Eloquent ORM	119
6.1.2.1	Query Builder	119
6.1.2.2	Eloquent ORM	120
6.2	Using Query Builder	122
6.2.1	Inserting records	123
6.2.1.1	Inserting a single record	123
6.2.1.2	Inserting multiple records	124
6.2.2	Retrieving records	124
6.2.2.1	Retrieving a single record	124
6.2.2.2	Retrieving all records in the table	125
6.2.2.3	Retrieving specific columns of all records in the table	126
6.2.2.4	Retrieving a limited number of records	127
6.2.3	Updating records	127
6.2.3.1	Updating specific records	127
6.2.3.2	Updating all records in the table	128
6.2.4	Deleting records	128
6.2.5	Deleting specific records	128
6.3	Filtering, sorting and grouping data	129
6.3.1	Query Chaining	129
6.3.2	Using “where” operator to filter data	130
6.3.2.1	Simple “where” query	130
6.3.2.2	Using “orWhere” operator	132
6.3.2.3	Using “whereBetween” operator	133
6.3.3	Using “orderBy” to sort data	133
6.3.4	Using “groupBy” to group data	134
6.4	Using Join Statements	135
6.4.1	Using Inner Join	135
6.4.2	Using Left Join	136
6.4.3	Using other types of Joins	137
6.5	Summary	139
7.	Eloquent ORM	140
7.1	Introducing Eloquent ORM	140
7.1.1	Introducing models	142
7.1.2	Understanding conventions	142
7.2	Creating and using models with Eloquent	144
7.2.0.1	Creating models	144
7.2.0.2	Using models	145
7.2.1	Inserting records	146
7.2.2	Retrieving records	147
7.2.2.1	Retrieving specific record by using query builder’s operator find()	148
7.2.2.2	Retrieving first record by using eloquent’s operator first()	149
7.2.2.3	Retrieving all records from a table using eloquent’s operator all()	149

CONTENTS

7.2.2.4	Using operator get()	150
7.2.3	Updating records	151
7.2.4	Deleting records	152
7.3	Using relationships	152
7.3.1	One to one relationships	152
7.3.2	One to Many relationships	153
7.3.3	Many to Many relationships	153
7.3.4	Polymorphic relationships	153
7.3.5	Eager loading relations	153
7.4	Filtering data	153
7.4.1	Using query builder	153
7.4.2	Using query scopes	153
7.5	Working with retrieved data	153
7.5.1	Using collections to iterate through data	153
7.5.2	Converting results	153
7.5.3	Displaying data	153
7.6	Overriding conventions	153
7.7	Summary	154

Introduction

Over the years PHP has grown up into a powerful and popular language for web applications. Without an underlying framework PHP-powered web applications can become messy and nearly impossible to extend or maintain. A framework that has session management, ORM, built-in authentication, and templating saves developers' time by providing the components that are necessary in most web applications. Open source PHP framework called Laravel provides all these components and many more, making you a very efficient developer capable of building better applications a whole lot faster. Laravel allows you to focus on the code that matters instead of constantly reinventing the wheel.

Using well-designed diagrams and source code, this book will serve a great introduction to Laravel PHP framework. It will guide you through usage of Laravel's major components to create powerful web applications. The book begins with an overview of a Laravel-powered web application and its parts, introduces you to concepts about routing, responses and views, then moves onto understanding controllers, dependency injection and database operations. You'll explore Laravel's powerful ORM called Eloquent, simple templating language called Blade and session management. You'll also learn how to implement authentication and protect your applications, learn how to build your own API's applying the concepts learned in the book.

About the author

My name is Maksim Surguy. I am a full time web developer, part time writer and former [breakdancer](#)¹. If you use Laravel PHP framework or Bootstrap, you might have seen some of the websites I created with Laravel:

- [Bootsnipp](#)²
- [Laravel-tricks, open source](#)³
- [Built With Laravel](#)⁴
- [Bookpag.es](#)⁵
- [Panopanda](#)⁶
- [MyMapList](#)⁷
- [Cheatsheetr](#)⁸

I love creating new products and in the process I try to share as much as I possibly can. You can read free web development tutorials on my blog at <http://maxoffsky.com>, my other book (<http://maxoffsky.com/frontend>), and you can follow me on Twitter for various web development tips and tricks at <http://twitter.com/msurguy>

¹https://www.youtube.com/watch?v=wEF_RHL1NFU

²<http://bootsnipp.com>

³<http://laravel-tricks.com>

⁴<http://builtwithlaravel.com>

⁵<http://bookpag.es>

⁶<http://panopanda.co>

⁷<http://mymaplist.com>

⁸<http://cheatsheetr.com>

Prerequisites

In order to use the book correctly, the reader should:

- Have some experience with command line (like creating and deleting files and folders, listing all files in a directory)
- Have heard of MVC but not necessarily have much experience using it
- Have basic understanding of OOP PHP
- Know how to create a basic database with PHPMyAdmin, other GUI tools or with a command line

If you don't have some of the skills necessary to keep up with the book's material, please feel free to ask [me](#)⁹ for pointers to some great resources.

Source Code

The source code for this book is available for each chapter and is located on Github at <https://github.com/msurguy/laravel-first-framework>¹⁰. Feel free to explore it, comment on it and improve it on Github.

⁹<http://twitter.com/msurguy>

¹⁰<https://github.com/msurguy/laravel-first-framework>

1. Meeting Laravel

This chapter covers:

- An introduction to Laravel 4 PHP framework
- History of Laravel framework
- Advantages of using Laravel

Developing a web application from scratch can be a tedious and complicated task. Over time, too many opportunities for bugs and too many revisions can make maintenance of a web application very frustrating for any developer. Because of this, a recently developed framework written in PHP, Laravel, has quickly won the hearts of developers around the world for its clean code and ease of use.

Laravel is a free PHP framework that transforms creation of web applications into a fun and enjoyable process by providing expressive and eloquent code syntax that you can use to build fast, stable, easy to maintain and easy to extend web applications. Laravel consists of many components commonly needed in the process of building web applications. PHP developers use Laravel for a broad range of web applications, from simple blogs and CMSs to online stores and large-scale business applications. It's no wonder Laravel has become a strong contender in the family of package-based web frameworks and is ready for prime time after only a few years of development.

In this chapter you will meet Laravel PHP framework, you will learn about its short but interesting history and its evolution over its lifetime, how by using established conventions and approaches Laravel can speed up development significantly and find out why you, too, will find Laravel a great choice for your applications. Are you ready? Let's meet Laravel!

1.1 Introducing Laravel 4 PHP framework

Laravel 4 is an open source PHP web application framework created by Taylor Otwell and maintained by him and a community of core developers. For over two and a half years web developers around the world have been using Laravel (version 1 to version 4) to build stable and maintainable web applications by using Laravel's powerful features such as built-in authentication, session management, database wrapper and more. Laravel is considered to be a full stack web development framework, meaning it can handle every aspect of a web application architecture – from storing and managing data using the database wrapper to displaying user interfaces using its own templating engine. Majority of the mundane work that a web developer encounters in the process of creating a web application has been thought of and alleviated by Laravel's components. Because of this, an application can be built a lot faster than it would be if the developer made the whole application from scratch.



What's a PHP Web application framework?

A PHP Web application framework is a set of classes, libraries or components written in PHP server-side scripting language that aim to solve common web development problems and promote code reuse. Such components as authentication, session management, caching, routing, database operations wrappers and more are included in Laravel and most popular PHP web application frameworks. Using a web application framework allows developers to save significant amount of time developing a web application.

While being a great time saver, Laravel isn't a magic cure for all your web development problems. Although it provides a solid foundation for the backend of your application, you still need to come up with the application logic, database structure and the HTML, CSS and Javascript for your application. Also, Laravel (and PHP scripting language in general) isn't well suited for real time applications like web chats, video streaming, real time gaming, and instant messaging so you would have to use something else for those kinds of applications.

The architecture of Laravel makes it fitting for building many kinds of web applications. From simple blogs, guestbooks, and single-purpose websites to complex CMSs, forums, APIs, e-Commerce websites and even social networks, Laravel will help you get web application development done faster.

It's not just the features that make Laravel so special. There are many reasons to use Laravel in your next web development project. Web developers choose Laravel for the expressive and clean code that it provides as a basis for web applications. The expressiveness is what distinguishes Laravel from other similar PHP frameworks. The application structure that Laravel provides makes it easy to start working even for a beginner developer that has never used a framework before. The community around Laravel, including the creator of the framework himself, has been very welcoming and supportive. Let's explore these reasons a bit more below. How does Laravel's expressive code help you, the developer, to keep the code of your application maintainable, easy to change and easy to read?

1.1.1 Laravel's Expressive code

The most prominent candidates for code re-use in web development with PHP are managing sessions, authentication and operating with data stored in databases. Laravel has tackled those areas quite gracefully and offers beautiful syntax to help you develop applications. Listing 1.1 shows a handful of examples of Laravel's expressive syntax:

Listing 1.1 Examples of Laravel's expressive syntax

```
$order = array('customer' => 'Prince Caspian', 'product' => 'Laravel: my first framework');  
// Store the order details in the session under name "order"  
Session::put('order',$order);  
...  
// Retrieve the order details from the session using the session element called "order"  
// and create a new record in the database  
$orderInDB = Order::create(Session::get('order'));  
  
// Remove the session element "order"  
Session::forget('order');  
  
// Retrieve all orders for customer "Prince Caspian"  
$ordersForCaspian = Order::where('customer', 'Prince Caspian')->get();  
...  
// Count all orders of "Laravel: my first framework"  
$ordersQuantity = Order::where('product', 'Laravel: my first framework')->count();
```

Laravel promotes usage of clean and expressive code throughout your web application while it still remains PHP code. All of Laravel's method names are logical, sometimes to an extent that you can even guess them without looking up the documentation. For example to detect if there is an item called 'order'

in the session already you would use `Session::has('order')` which might come naturally to your mind. The same could be said about the sensible defaults for all methods that need to return some value: session items form input elements, cache items, and more. As another minimal example, you could return a default value for an input submitted with a form if nothing was provided by the user: `Input::get('name', 'Guest Customer')`.

These conventions might not immediately appeal to every developer, but in the long run they make the framework consistent and unified. Laravel makes many assumptions that accelerate web development tremendously. While developing applications with Laravel you will enjoy those moments when the framework appears to know exactly what you intend to do. Aside from the expressive and clean code, Laravel makes your applications easier to maintain by adopting separation of the application code into files that work with the data, files that control the flow of the application and files that deal with the output that the end user will see.

1.1.2 Laravel applications use Model-View-Controller pattern

When you install Laravel you will notice that inside of the “app” folder there are folders called “models”, “views”, “controllers” amongst other folders. This is because Laravel is a framework for MVC applications. Such applications use the Model-View-Controller software architecture pattern, which enforces separation between the information (model), user’s interaction with information (controller) and visual representation of information (view). As you develop your Laravel applications you will mainly work with files inside of those three folders. Don’t worry if you are not familiar with MVC pattern yet, Laravel is a great framework to learn MVC and there is a section of this book that will give you an introduction to this software pattern and its usage in Laravel applications.

Consider having your whole web application in one PHP script. Mixing SQL statements, business logic and output all inside one file will be very confusing for any developer to work on. Especially if it has been quite some time since you have looked at the code of such application. As the application grows it will become harder or impossible to maintain. The main benefit of using MVC architecture in Laravel applications is that it helps you have the code of your application separated and organized in many small parts that are easier to change and extend. You will come to appreciate this separation when you will want to add a new feature into your application or when you will need to quickly modify existing functionality. Laravel has been around for more than two years. Even though that doesn’t seem like a lot, it has matured and has been road tested by thousands of developers worldwide. The developers that use Laravel communicate to each other through many different online and offline channels, sharing tips, code examples, and helping maintain the framework.

1.1.3 Laravel was built by a great community

Contrary to popular belief, Laravel is not one-man’s product. While Taylor Otwell is the main developer behind the framework – the visionary and the person in charge of saying “yes” or “no” to features, the open source nature of Laravel made it possible to get code contributions and bug fixes from over a hundred developers worldwide, Laravel’s public forums (with over 20,000 registered users) and a thriving IRC channel helped steer the development and testing of the framework. Laravel’s vibrant and welcoming community at <http://laravel.com> is something that makes Laravel special and distinctive from other similar projects.

Laravel would not be in the place where it is now if not for contributions, ideas, suggestions and passion of so many developers and lovers of clean code all around the world. In fact, many Laravel features were inspired by other open source projects like Sinatra, Ruby on Rails, CodeIgniter and others. How did Laravel come such a long way in just over two years? Let’s explore Laravel’s history and see how it evolved over time.

1.2 History of Laravel framework

Rome wasn't built in a day, and so weren't any of the established frameworks. Usually building a solid web development framework that is well tested and ready for production deployment takes a long time. Though interestingly, in case with Laravel its development was progressing a bit differently and at a faster pace than other frameworks.

1.2.1 State of PHP frameworks world before Laravel 4

In August of 2009 PHP 5.3 was released. It featured introduction of namespaces and anonymous functions called closures amongst other updates. The new features allowed developers write better Object-Oriented PHP applications. Even though it provided many benefits and a way to get to a brighter development future, not all frameworks were looking into that future but instead focused on supporting the older versions of PHP. The framework landscape mainly consisted of Symfony, Zend, Slim micro framework, Kohana, Lithium and CodeIgniter.

CodeIgniter was probably the most well known PHP framework at the time. Developers preferred it for its comprehensive documentation and simplicity. Any PHP programmer could quickly start making applications with it. It had large community and great support from its creators. But back in 2011 CodeIgniter was lacking some functionality that Taylor Otwell, the creator of Laravel, considered to be essential in building web applications. For example out of the box authentication (logging users in and out) and closure routing were absent in CodeIgniter. Therefore, Laravel version 1 beta was released on June 9, 2011 to fill in the missing functionality. According to Laravel's creator, Taylor Otwell, Laravel version 1 was released in June 2011 simply to solve the growing pains of using CodeIgniter PHP framework.

1.2.2 Evolution of Laravel framework

Starting with the first release, Laravel featured built-in authentication, Eloquent ORM for database operations, localization, models and relationships, simple routing mechanism, caching, sessions, views, extendibility through modules and libraries, form and HTML helpers and more. Even on the very first release the framework already had some impressive functionality. Laravel went from version 1 to version 2 in less than six months.

The second major release of the framework got some solid upgrades from the creator and the community. Such features were implemented: controller support, "Blade" templating engine, usage of inversion of control container, replacement of modules with "bundles" and more. With the additions of controllers, the framework became a fully qualified MVC framework. Less than two months later new major point release was announced, Laravel 3.

The third release was focused on unit test integration, the Artisan command line interface, database migrations, events, extended session drivers and database drivers, integration for "bundles" and more. Laravel 3 was quickly catching up to the big boys of PHP frameworks such as CodeIgniter and Kohana, a lot of developers started switching from other frameworks to Laravel for its power and expressiveness. Laravel 3 stayed in a stable release for quite some time but about 5 months after it was released the creator of the framework decided to re-write the whole framework from scratch as a set of packages distributed through "Composer" PHP dependency manager. Thus Laravel 4 codenamed "Illuminate" was in the works.

Laravel 4 was re-written from the ground up as a collection of components (or packages) that integrate with each other to make up a framework. The management of these components is done through the best PHP dependency manager available called "Composer". Laravel 4 has an extended set of features that no other

version of Laravel (and even no other PHP framework) has had before: database seeding, message queues, built in mailer, even more powerful Eloquent ORM featuring scopes, soft deletes and more.

Embracing the new features of PHP 5.3 Laravel has come a long way in just over two years appealing to more and more developers worldwide. The visionary behind the framework – Taylor Otwell and the community surrounding Laravel have made enormous progress creating a future-friendly established architecture for PHP web applications in a very short period of time. Holding this book testifies on Laravel’s success and constantly growing community of users and contributors means that Laravel is here to stay. But what makes Laravel so beneficial for developers, and why should you care to use it in your projects instead of hand made code or instead of other available frameworks?

1.3 Advantages of Using Laravel

There are many advantages to using Laravel instead of hand-made code or even instead of other similar frameworks. Laravel is a winner because it incorporates established and proven web development patterns: convention over configuration so it is ready for use out of the box, Model-View-Controller (MVC) application structure and Active Record powering its database wrapper. By using these conventions and patterns Laravel helps you as a developer to build a maintainable web application with easy to understand code separation and in a short time frame.

Without a framework like Laravel maintenance or adding new features to a web application could become a nightmare. Lack of structure will create problems when another developer will want to fix bugs or extend the application. Absence of standards will make it hard to understand application’s functionality even for most experienced developer. Unconventional method and function names might make the code of the application unreadable and low quality. Laravel helps you avoid all of these problems. Using commonly accepted conventions and established software development patterns Laravel helps your applications stay alive and relevant. Over the next few pages I will take you on a tour exploring the advantages of using Laravel framework over hand made code or other frameworks. First destination: “Convention over configuration”.

1.3.1 Convention over configuration

Imagine if you could do a fresh install of the framework and in the next five minutes already be working on how your application behaves when a user visits a certain page and not worry about configuring the things that are common to most applications? Laravel makes many assumptions that make this kind of scenario possible, this is known as “convention over configuration” pattern. When you start writing a new web application you would like to have as minimal initial set up as possible. Such configuration settings as the location where the application will save your sessions and cache, what database provider and what tables in the database it will use, locale of the application and more settings are usually needed for web applications. Thankfully all these configurations are already pre-set for you in a fresh installation of Laravel, but, if needed, any changes to these configurations could be easily made.



Convention over configuration

Convention over configuration is a software design pattern that minimizes the number of decisions that a developer needs to make by implying reasonable configuration defaults (conventions) requiring the developer to provide the configuration for only those parts that deviate from the conventions.

The configuration of new Laravel applications is very minimal and it is all separated for you inside of

one folder: “app/config”. You need to only change the settings that are different from pre-set conventions. For example provide a database name if you are using one, your time zone, and you are good to go. If you are not using a database or don’t care about time zone settings, then you don’t even have to do any configuration and can start coding your application right away! All the other settings are set to reasonable defaults and you need to change them only if your application departs from pre-set conventions. We will explore Laravel’s configuration more in detail in the next chapter. This smart way of minimizing the amount of decisions that a developer has to make to start developing an application makes Laravel very easy to get started with and appealing to developers of all proficiency levels.

1.3.2 Ready out of the box

With some frameworks it could take hours just to get to a point where the developer can start using the framework. Laravel is unique in this sense because you can install and start using Laravel in just few minutes. When you install Laravel for the first time you will notice that you don’t need to do anything else to have your application operational. After starting a server using built in command you can right away visit the root URL of the application and get a response from Laravel signifying that your Laravel application is ready (figure 1.1):

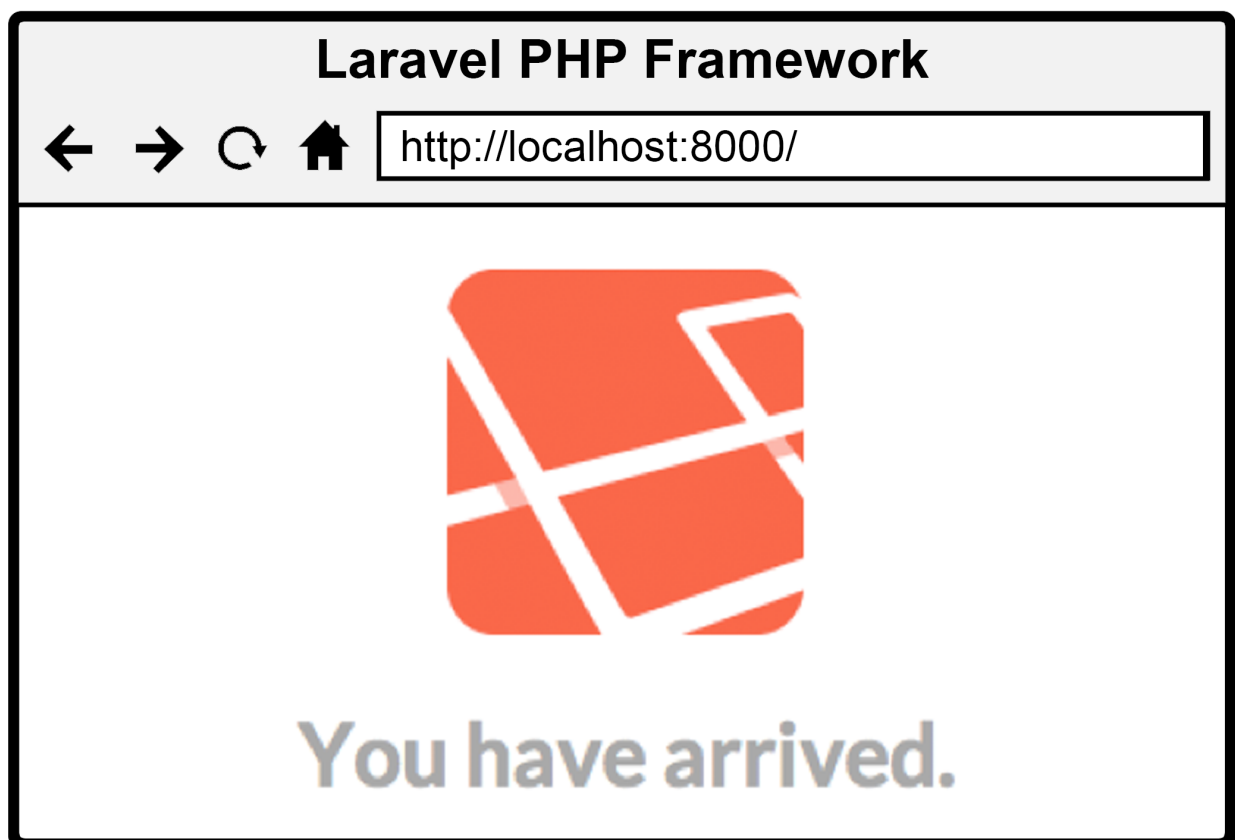


Figure 1.1 Laravel is operational out of the box with zero configuration

Amazingly when this page shows up Laravel already assumes that you are developing locally, it knows

where to save the cached pages, it knows that going to the application's root URL is supposed to display a page with some pre-set content in it. This is great because you can get to the development of your application's logic, models and responses in no time. Not only your application is ready for your coding attention, it already has some great structure to it.

1.3.3 Clear organization of all parts of the application

Everything has its own place in a Laravel application. Because applications built with Laravel follow MVC pattern, the data models live in `app/models` folder, the views reside in `app/views` folder. Can you guess where the controllers are? You got it right, in `app/controllers` folder! This separation makes your code cleaner and easier to find. For example if you have a database table called "orders" you will most likely have a model for it called "order" in your models folder. If you have created a nice login screen for your application it will be somewhere in the views folder. Take a look at figure 1.2 to see which folders your Laravel applications will consist of.

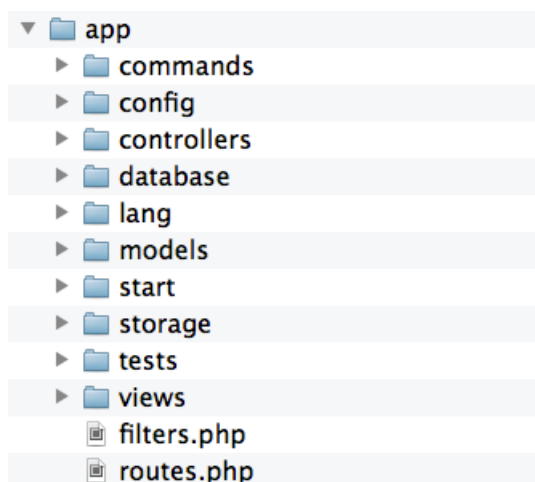


Figure 1.2 Folder structure of a Laravel application

There are a few more folders other than models, views and controllers but don't be afraid, there will be a whole section of the next chapter devoted to all of these folders because together they are the heart and mind of your applications (while Laravel itself is the soul). Each file and folder has its purpose but by conventions they are already pre-filled with something so you can get started quicker and change only the parts that need to break out of the conventions. There are many things that made Laravel so successful. Clean organization is certainly one of those things. Another one is built-in Authentication.

1.3.4 Built-in Authentication

An important feature that has been missing in some other big name frameworks is built-in authentication for web applications.



Authentication

Authentication is the process of identifying a user, usually by some unique user attribute (username or email) and accompanying password, in order to grant access to some specific information.

Since its first release, Laravel featured authentication methods to log users in and log them out and also making it easy for a developer to access properties of the currently logged-in user. Laravel 4 only continues the trend of authentication being pre-baked into the framework and greatly simplifies the process of authentication. In listing 1.2 we will explore some authentication methods that Laravel provides. By default authentication uses a table called “users” in your database and the user model is already in the app/models folder. Please note that the following code is just to demonstrate the simplicity of authentication methods so it is not a part of any particular file (we will get to that in detail later):

Listing 1.2 Demonstration of Laravel’s built-in authentication methods

```
// Provide username and password to check against (could be from input)
$user = array('username' => 'caspien356', 'password' => 'P@s5w0rD');

// Check user's credentials against user's details in the DB table called "users"
// and log the user in
if (Auth::attempt($user))
{
    return 'User has been logged in successfully';
}
else
{
    return 'Login failed';
}
...
// Check if the user's browser has a cookie matching Laravel's session contents
if (Auth::check())
{
    return 'User is logged in';
}
...
// Retrieve some attributes of the currently logged in user
$username = Auth::user()->username;

echo $username;
// Outputs 'caspien356'
...
// Log the user out, by erasing the login cookie from the browser and destroying user's session
Auth::logout();
```

These are just a few examples of Laravel’s incredible authentication functionality. The list above is not a comprehensive list of all authentication methods Laravel 4 has up its sleeve. There are also password resets through a link sent in the email, HTTP Basic authentication (great for APIs) and of course built in methods of protecting areas of your web application. In later chapters we will get to know them better but for now let’s take a look at another reason why Laravel is a great framework to use for your next web development project – methods of working with contents of databases.

1.3.5 Eloquent ORM – Laravel’s Active Record implementation

Working with data is an essential part of all web applications. Usually PHP web applications store data in rows of database tables. Starting with version 5.0 the PHP language introduced a unified interface to query databases and to fetch the results. While those new features were helpful and enough for some basic applications, they made application’s code look like a mess of SQL statements interwoven with PHP code. Also they did not add much convenience to working with data that is related with other data by means of foreign keys. To alleviate that imagine if you could present the data stored in a database table as a class with each row being an object. That would simplify access to the data in database and also would make it possible to relate the data by means of assigning objects to other objects’ properties. This technique of wrapping DB data into objects is called Active Record pattern. Laravel implements Active Record pattern in its “Eloquent ORM”.



ORM

Object Relational Mapping (ORM) enables access to and manipulation of data within a database as though it was a PHP object.

Laravel’s Eloquent ORM allows developers to use Active Record pattern in full extent making application’s code that deal with database look like clean PHP and not like messy SQL. Creation of new entries in database, manipulating and deleting entries, querying the database using a range of different parameters and much more are made easy and readable with Eloquent ORM. Eloquent makes quite a few assumptions about your databases but at the same time all of those assumptions could be overridden and customized (again, convention over configuration). For example it expects the primary key of each table to be called “id” and it expects the name of the tables to be plural of the model’s name.

Let’s say you have a table called “shops” in a database. A model that corresponds to that table would have to be called “Shop”. If you wanted to use Eloquent ORM to create a new row in “shops” table containing the details of a new shop, you would do that by creating a new object of class Shop and operating on that object. This is done with code in listing 1.3:

Listing 1.3 Demonstration of Eloquent ORM creating a row in database

```
// Define the Eloquent model for "shop" in app/models/shop.php file
class Shop extends Eloquent {}
...
// Create a new instance of the model "shop"
$shop = new Shop;

// Assign content to the columns of the database row that will be created
$shop->name = 'Best Coffee Co';
$shop->city = 'Seattle';

// Perform a save of the new row into the database
$shop->save();
```

With Eloquent ORM you are not limited to operating on just one table in the database. Built-in relationships allow you to manipulate related models and in turn related tables. Eloquent assumes that you have a foreign key set up in format “model_id” on the tables that need to be related to a particular model. For

example if you have a table called “customers” containing foreign key called “shop_id” and wanted to relate the newly created shop to a new customer, you would use code that follows in listing 1.4:

Listing 1.4 Using Eloquent relationship to create related model

```
// Define Eloquent model for "customer" in app/models/customer.php file
class Customer extends Eloquent {}

...
// Eloquent model of "shop"
class Shop extends Eloquent {

    // One-to-many Eloquent relationship that relates model "shop" to model "customer"
    public function customers()
    {
        return $this->hasMany('Customer');
    }
}

...
// Create new object of "customer" Eloquent model
$customer = new Customer(array('name' => 'Prince Caspian'));

// Create a new row in table "customers" while setting "shop_id" column
// to the "id" of the shop that was created previously
$customer = $shop->customers()->save($customer);
```

The examples above are just a tiny tip of the iceberg of what’s possible with Eloquent. As you can see, Eloquent makes manipulating database data and related data a breeze while keeping the code of the application clean and easy to read. This book offers a whole chapter devoted to this powerful and unique ORM because being able to work with it will save you countless amounts of time. As many other developers did, you will find it a joy working with Laravel’s Eloquent ORM.

1.4 Summary

You have been introduced to Laravel PHP framework and have learned about its history and importance. Laravel has matured very quickly in just over two years. While being a young and fresh framework, it uses well established software design patterns:

- MVC application structure to keep the code separated while maintaining flexibility
- Active Record in its Eloquent ORM to simplify working with data inside of a database
- Convention over configuration to keep the setup minimal

You have learned that Laravel is a full stack framework. It features built in authentication, it has had a unique and powerful ORM since the first release, it is ready to be used out of the box with minimal configuration. From templating to working with data, Laravel has you covered. Applications built with Laravel feature code that is both expressive and clean at the same time, which makes them easier to maintain.

In the chapters that follow you will become a Laravel expert. Let's begin this journey by installing Laravel and making our first application with it!

2. Getting started for Development with Laravel

This chapter covers

- Installing Laravel with Composer
- Meeting Artisan: the command line interface
- Understanding structure of a Laravel application
- Creating a website with Laravel

Getting started with some PHP frameworks could be a hard and painstaking process with a steep learning curve. Fortunately this is not the case with Laravel. Laravel is beginner-friendly, easy to install and configure, and even somebody who never used another framework before could get started quickly. This chapter will first give you an idea of how development with Laravel looks like in general. Then you will meet a tool underlying Laravel called “Composer” and you will learn how to install Laravel using it. You will meet Laravel’s helper tool called “Artisan” and after that you will learn in detail about Laravel application structure. Finally, you will build a complete and functional website with Laravel. While building the site you will see the following Laravel’s features in action: routing, templating, database operations, sending mail and more. Let’s get started by taking a high level look at the process of developing an application with Laravel.



Prerequisites

This chapter assumes that you have Apache, PHP 5.3.7 or greater with MCRYPT extension installed and database engine like MySQL already installed and set up.

2.1 Development overview

In the previous chapter you have learned that Laravel comes pre-configured to work out of the box. When you have Laravel installed you can get into developing your application right away. The development process for most web applications built with Laravel generally looks like this:

- Configuring database, cache, mail and other settings if necessary
- Creating the end points (routes) of your application
- Creating the models and the database structure for the data
- Creating the controllers and integrating them with the routes and the views
- Creating the view templates that will make up the user-facing side of the application
- Testing the application
- Refining the code of the application

Don't worry if some of these terms are not familiar to you yet, we will be giving proper attention to all of them throughout the book. In fact, you will get to meet most of these concepts in this very chapter!



A note on design patterns

While there are many specific software development methodologies such as BDD (Behavior Driven Development), TDD (Test Driven Development), DDD (Domain Driven Design) and more, this book will try to stay neutral and follow a general simplified development model as described above

Now that you are aware of how the development process of Laravel applications looks like in general, let's get familiar with a very important tool that Laravel uses for its architecture.

2.2 Meeting Composer

Laravel framework consists of many separate components that are downloaded and put together automatically into a framework through a special tool called "Composer". This tool could be downloaded from <http://getcomposer.org/>, it is the holy grail of dependency management for PHP.

In the old days to install a PHP library or a PHP framework you could just download it as a Zip archive from a website. If those PHP libraries or frameworks were to implement third-party libraries or classes, eventually it would get hard or impossible to keep track of dependencies (due to large number of components, new versions, bug fixes or patches coming out). There were a few attempts at solving the dependency management problem in PHP community in the past. Maybe such things as PEAR or PECL sound familiar to you? Those are some of the popular PHP dependency managers but due to their shortcomings they are failing at delivering the promised efficiency when it comes to publishing new dependencies or managing already existing ones inside a library or a framework. Learning from past mistakes and from other software development ecosystems, PHP community eventually came up with a tool that is flexible, powerful and easy to use - Composer.

Being a tool like "npm" for NodeJS or "RubyGems" for Ruby, Composer allows PHP developers, including those who develop with Laravel, to easily specify which other PHP libraries are used in their code or initiate installation and update of those dependencies through the command line. Composer has been around since 2011 but just like Laravel, matured very quickly into a solution that many developers love to use. Before creating your own library be sure to check Composer's website for existing solutions because there are thousands of packages to choose from. Making Composer as the tool of choice for installing Laravel and managing its components allows for great flexibility and extendibility of Laravel applications.

2.3 Installing Laravel

When you have Composer configured (see Appendix for instructions on that), you can install and start using Laravel by just executing one command in the command line replacing "companySite" with the name of your project:

```
composer create-project laravel/laravel companySite --prefer-dist
```

After executing this command from the command line Composer will download all components that Laravel is comprised of and weave them together into a framework ready to become your next web application.



Please note

You will need to run this command for every new Laravel project unless you create some base template that you could use for all of your applications

2.4 Meeting Artisan: Laravel's command line interface

Laravel comes pre-packaged with a powerful command line interface (CLI) called “Artisan”. Many repetitive tasks like model creation, running database migrations, launching development server and more can be executed with Artisan. Besides over 30 built-in commands Laravel allows for creation of custom commands.



Artisan CLI

Artisan Command Line Tool, or Artisan CLI, is a helper tool that comes with Laravel and could be used through operating system's command line to accelerate development of a web application.

You can run Artisan commands through your system's command line (ex. Terminal on Mac OS X). Navigate to your Laravel application's root and run the following command:

```
php artisan
```

You should see the version of the framework your application uses and a list of all commands available through Artisan CLI. Executing commands through Artisan affects only the application that you navigated to in the command line, not all Laravel applications system-wide. You will get to use many of Artisan's commands throughout this book to accelerate development. Now that you have learned about installation and have met the Artisan CLI, let's take a look at the general structure of a web application built with Laravel.

2.5 Application structure overview

A web application built with Laravel consists of certain components that work together to make up a product that functions as the developer intended. Some of these components the developer will need to create: Model-View-Controller code of the application, definition of application's endpoints (routes), database structure (migrations, models), and unit tests. Others, the developer needs to adjust and configure: various application settings, environment settings and third party packages. Figure 2.1 illustrates which components a Laravel web application consists of:



Figure 2.1 Components that make up a Laravel web application

Even though that might look like a lot of parts to an application, don't worry, as you will see throughout this chapter not all of them need to be created or modified by you for each and every application.

All these parts of the application will need to be located in specific places inside of a folder that is created after installation of Laravel following instructions in the "Installing Laravel" section. Let's start discovering the directory structure of a typical Laravel application by looking at the root level of the application.

2.5.1 Files and folders in the root level of the application

The root folder of Laravel application contains Composer configuration, condensed information about the framework, folders containing environment settings, public-facing files, application code and folder with all installed packages including packages that make up Laravel framework itself. Table 2.1 shows a list of all files and folders in application's root directory when Laravel is installed through Composer's "create-project" command:

Table 2.1 List of root-level files and folders of a Laravel application

Folder or file	Purpose
/app	Contains your application's code
/bootstrap	Contains Laravel framework's directory paths, compiled Laravel framework file and application environment settings
/public	This is the public-facing folder. It will contain the CSS, Javascript files and other files or folders that will be accessible to users using your application
/vendor	Third party packages and packages making up Laravel are installed into this folder.
composer.json	This Composer configuration file contains a list of packages that the application is using, version settings for each package and stability settings
CONTRIBUTING.md	Contains instructions on how to contribute to Laravel PHP Framework.

Table 2.1 List of root-level files and folders of a Laravel application

Folder or file	Purpose
readme.md	Provides general information about Laravel and links to documentation, pull requests and the license of the application. If you are making an open source project on top of Laravel be sure to update this file to reflect the purpose and description of your application
server.php	Laravel has a special command that launches the current application on the system's PHP server. This file is a helper file for that functionality
artisan	Provides the built-in command line interface for Laravel,so that a developer can accelerate certain tasks
phpunit.xml	Configuration file for PHPUnit unit testing tool

There will be times when you won't have to touch all of these files and folders during development of your application. For example "vendor" folder is managed by Composer tool so you don't need to modify anything in that folder, you would go into "bootstrap" folder only when you want to add a new environment setting, "phpunit.xml" file will only be modified when you will be using PHPUnit unit testing tool, etc. On the other hand there is one very special folder inside of the root, it is called "app" and it will contain the actual code of your application. Let's take a look inside of that most important folder.

2.5.2 Contents of the "app" folder

When building applications with Laravel, this folder will be the center of your attention. The routes, route filters, various configuration settings and database migrations for your application will reside in this folder. Because Laravel uses Model-View-Controller pattern for your application, you will notice "models", "views" and "controllers" folders among other folders in the "app" folder. These folders store your application's data models, view templates and application controllers respectively. Table below (table 2.2) shows a list of all files and folders inside of the "app" directory:

Table 2.2 Contents of the "app" directory

Folder or file	Contains
/models	Classes that represent the data models
/views	Templates for views and view layouts
/controllers	Application Controllers
/config	Application-specific settings such as database credentials, session and cache driver settings
/database	Database seeds (sample data) and database migrations
/lang	Language strings for validation, pagination and email reminders
/commands	Custom classes containing Artisan commands
/storage	Cache, temporary sessions and compiled views
/start	Files that help you adjust an error handler and,application maintenance mode behavior
/tests	Application's unit tests
filters.php	Logic and definition of filters used in the application
routes.php	List of all registered endpoints of the application (routes)

Knowing the file and folder structure of a Laravel application will help you put things in right places so please take a closer look at each file and folder created after installing Laravel.

2.5.3 Configuration settings

Configuring a Laravel application is pretty simple comparing to other frameworks or hand made code. Configuration files in a Laravel application are stored inside directory “app/config” and have file names corresponding to what settings they affect. In table 2.3 let’s explore responsibilities of each configuration settings file and folder:

Table 2.3 Configuration settings inside of the “app/config” directory

Folder or file	Contains settings for
/packages	Packages that the application is using
/testing	Testing environment
app.php	Application key, timezone and locale, debugger settings, service providers and aliases for classes
auth.php	Authentication driver, model and table used to represent users, password reminder settings
cache.php	Cache driver, cache database connection and table
compile.php	Classes that will be included when application is optimized
database.php	Database connection for MySQL, Sqlite, Postgres, SQLServer and Redis
mail.php	Email driver, SMTP settings, “From” address and name
queue.php	Queue driver and connection
session.php	Session driver and connection, session lifetime and name of the cookie
view.php	View storage
workbench.php	Creation of new packages using Laravel’s Workbench tool

By default the configuration settings are set so that you can use Laravel right away:

- The session driver is set to save sessions on the disk inside of “app/storage/sessions” directory.
- The cache driver is set to save the cache on the disk inside of “app/storage/cache” directory.
- The time zone is set to UTC.
- The language is set to English.
- The database driver is set to MySQL (you only need to provide connection settings).

The provided settings are enough to get started but most likely your application will use a database or have your own time zone, and some settings will have to be different than the provided defaults. The comments inside of each configuration settings file make it easy to understand what the options are for each setting. Besides the default configuration stored in “app/config” folder, Laravel uses a concept called “environments” to manage application settings depending on where the application is being accessed from - a local machine or by going to a URL on the web. We will learn about environments a bit later, but meanwhile let’s apply the knowledge we learned in this chapter to create a fully functional Laravel application, a company website!

2.6 Tutorial: building a website with Laravel

Let’s imagine that you are running a marketing/development company and you would like to build a simple website for your company. The company website will consist of three simple pages: home page with a welcome message, a page that lists all services offered by the company and a contact page with company’s contact information and a functional contact form. The sketches (wireframes) for the company website follow in figure 2.2:

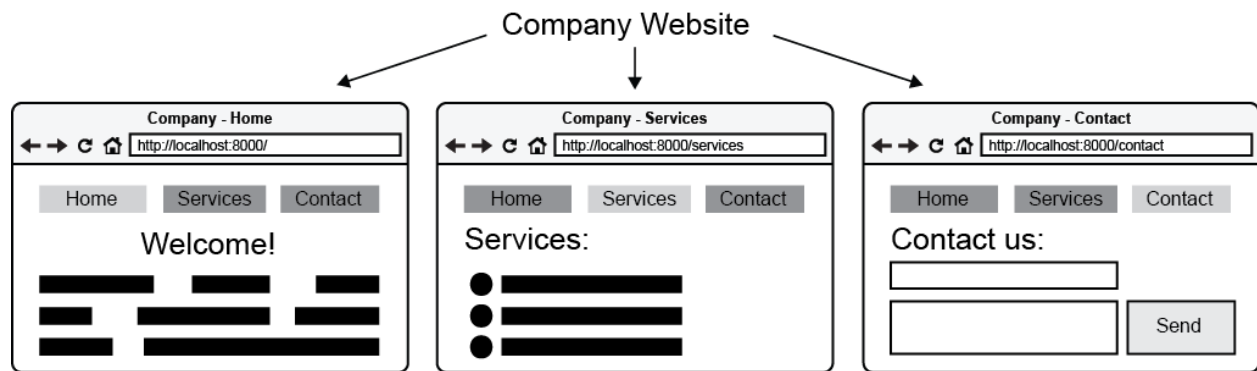


Figure 2.2 A wireframe of company website's pages

Following the development of an application from start to finish will give you an idea of how the development with Laravel looks like in practice. When you finish building the application you will be familiar with the way Laravel applications are structured and you will see how Laravel makes it easy for you to modify your applications in the future. In the process of meeting Laravel's methods you will understand how the Laravel framework alleviates common web development problems and you will experience its advantages over plain PHP code first hand.

So that there is a concrete goal, let's define some specifications for the site's individual pages. These pages will have the following functionality:

- Home page – a page with information about the company and a menu for navigation
- Services page - contains a list of the services that the company offers, this list will be retrieved from the database and embedded into the output HTML of the page
- Contact page - contains a contact form with two fields, the subject and the body of the message. This message will be sent as an HTML email to the site's administrator by using Laravel's mail functions.

While keeping in mind these basic requirements we will create a site that will work according to the specifications. Before we get into development let's first take a high level look at how building a real website with Laravel looks like.

2.6.1 The big picture – development overview

The development of the company website will consist of 9 steps in the following sequence, starting with installing Laravel (figure 2.3):

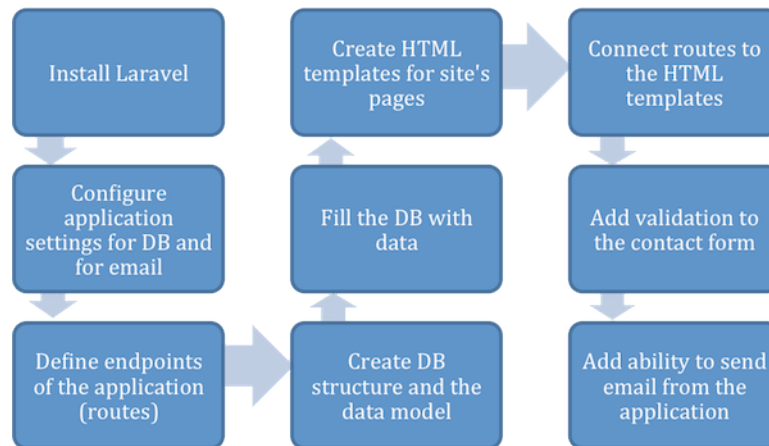


Figure 2.3 Overview of the development steps necessary to build an application

1. Installing Laravel by creating a new Laravel project.
2. Configuring application settings for the DB and for sending of emails
3. Defining application's endpoints (routes)
4. Creating the HTML templates for each of the site's pages
5. Create the DB structure and the data model for the services offered by the company
6. Filling the DB with data that represents the services offered by the company
7. Connecting the HTML templates to the endpoints (routes) of the application
8. Adding validation to the contact form to prevent empty submissions
9. Adding ability to send emails from the application

At the end of the development process you will have a fully functional company website that will work according to the specifications we defined in the introduction to this section. Let's start the development of the company site by installing Laravel!

2.6.2 Creating a new Laravel project

To download and install Laravel we will use Composer command described in section 2.3 of this chapter. Open up your command line and navigate to where you would like the company website created, then execute "create-project" command that will create a new Laravel project:

```
composer create-project laravel/laravel companySite --prefer-dist
```

After Laravel's components are downloaded and put together by Composer, you should see a new folder called "companySite" appear in the current directory. Navigate to the new folder, in particular switch to the "app" folder because the "app" folder will be the heart of your application. You will see a folder structure described in section 2.5.2. Open up your favorite code editor and get ready to write some code.

2.6.3 Configuring application settings

In case with our example website there are only two things to configure, database settings and email driver settings (figure 2.4):

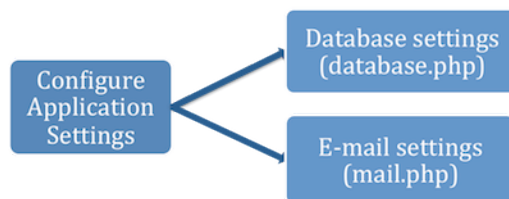


Figure 2.4 Overview of application settings configuration

Since the application will use a database to store the data for the services that the company offers, you will need to create a database first and then tell the Laravel application what settings to use for the database connection. To change the database connection settings open up “database.php” file in the “app/config” folder and set the settings to match your database connection, for example listing 2.1 shows settings for MySQL database engine:

Listing 2.1 Adjusting MySQL database connection settings in app/config/database.php

```
...  
'mysql' => array(  
    'driver'     => 'mysql',  
    'host'       => 'localhost', // Host where MySQL is running  
    'database'   => 'company', // Name of the database to connect to  
    'username'   => 'root', // The user that has access privileges to the MySQL DB  
    'password'   => 'root', // Password of the user  
    'charset'    => 'utf8',  
    'collation'  => 'utf8_unicode_ci',  
    'prefix'     => '',  
)  
...  
...
```



Database engine

This book will use MySQL as the database engine of choice but you are free to use any of the four database engines that Laravel supports: MySQL, Postgres, SQLite, or SQL Server.

Now that the database settings are configured let’s configure the settings for sending email. Laravel supports three different email drivers: PHP’s built in “mail” driver, “smtp” driver and the “sendmail” driver. For this tutorial we will use the “mail” driver that comes with PHP because it needs minimal setup. The mail settings for the Laravel application are located in “app/config/mail.php” file so let’s open it and edit the following lines (listing 2.2), saving the file afterwards:

Listing 2.2 Adjusting email driver settings in app/config/mail.php

```
...
// Set the driver to PHP's mail driver
'driver' => 'mail',
...
// Set the details for the address and the name where the email will appear to be sent from
'from' => array('address' => 'your@email.com', 'name' => 'Admin'),
...
```

That's it! Thanks to adjusting these settings we will be able to send email through the contact form in the application. This concludes configuration of the application, the rest of the settings are set to reasonable defaults (as specified in section 2.5.3). Next up, you will be defining the routes for the application.

2.6.4 Specifying endpoints (routes) of the website

Each website and web application has certain endpoints that are reachable by typing a URL in the browser's address bar. These endpoints are called “routes” (we will look more into the concept of routes in chapter 3). Our example website will have routes for the following URLs, along with descriptions of what they will do:

- “/” – root route, will show the index page of the company site
- “services” – will show the services page
- “contact” – will show the contact page, also route is used to submit the contact form

In Laravel applications, the routing is defined in “app/routes.php” file. Open that file up, you should see the following default code (listing 2.3) that comes with a new Laravel application:

Listing 2.3 Initial contents of app/routes.php file

```
Route::get('/', function()
{
    return View::make('hello');
});
```

We will specify the application's routes by replacing the default route with the following definitions for the application's routes (listing 2.4):

Listing 2.4 Modifying routes in the app/routes.php file

```
// Route's destination is specified as the first parameter of Laravel's Route::get method
Route::get('/', function()
{
    // Returning a string from a route will display the content of the string
    // in the browser when this route is reached
    return 'Home page';
});

Route::get('services', function()
```

```
{  
    return 'Services page';  
});  
  
Route::get('contact', function()  
{  
    return 'Contact page';  
});
```

Having defined these routes we can now test the application in the browser and see the routes working. Let's start Laravel's development server by using an Artisan command in the command line:

```
php artisan serve
```

This command will launch system's PHP server on port 8000 and will make the application reachable by typing the following URL in the browser:

```
http://localhost:8000
```

Go ahead and try the three new routes in the browser, the root route, the services page route and the contact page route. You should see the text output according to what we specified in the "routes.php" file (figure 2.5):



Figure 2.5 The basic routes for three pages set up and displaying text when you visit them

Later in this chapter we will come back to these routes and add ability to display a template file instead of just text strings.

2.6.5 Creating the database structure and the model for the data

Next step in the development of the application will be adding application's models and specifying the structure of the data in the database (figure 2.6):

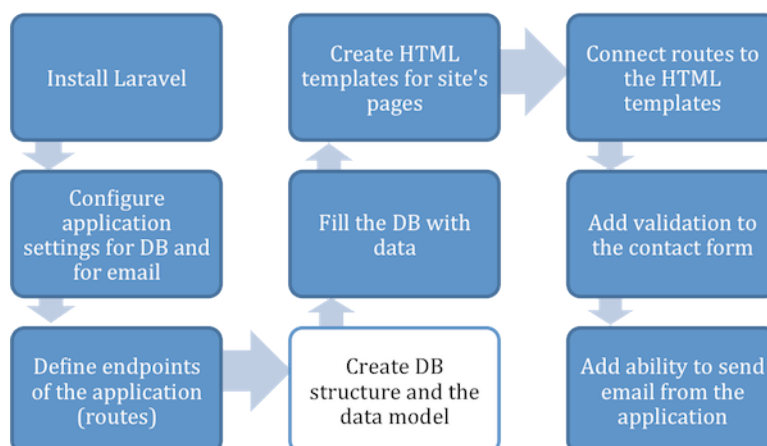


Figure 2.6 When the routes are set up, create the database structure and data model

As we defined in the specifications, the database will be used only for one purpose - to retrieve the list of services that the company offers. Each service offered by the company will have a title and description and a list of these services will be shown on the “services” page of the site. What could the structure be for the table that will store the company’s services? Let’s think this through. First, a unique id is needed; we can use an auto-incrementing integer for that and just call it “id” for simplicity. Second, we will give the title to each service, and third, some text description. Last, let’s give this table a name “services” since that perfectly describes what data it will store. That is all we need for this simple application’s database. Please take a look at the resulting table structure in figure 2.7:

Services		
PK	id	int (11)
	title	varchar (255)
	description	text

Figure 2.7 Database structure for table “services” that will store company’s services

With this database structure in mind, we could create the “services” table through various means, by using PHPMyAdmin, the command line or by using Laravel’s database migrations. What are migrations? Migrations are like blueprints for a database. Migrations allow developers to have an easy way to keep track of changes in database structure; they are very much like version control for a database structure. For the purpose of seeing more of Laravel’s capabilities, let’s use a migration to create our table with Laravel. We will use Laravel’s helper tool Artisan to create the skeleton for the migration, so go back to the command line and execute the following command (make sure you are in application’s root directory):

```
php artisan migrate:make create_services_table
```

After the migration skeleton is created, it will appear in “app/database/migrations” folder, the file will have a name that starts with the current date/time and ends with “create_services_table.php”. Open this file in the code editor. You will need to fill the contents of the “up” and “down” functions of the migration file with the following schema definitions (listing 2.5):

Listing 2.5 Migration schema for “services” table

```
...
public function up()
{
    // Set the name of the table
    Schema::create('services', function($table)
    {
        // Create a column for an auto incrementing primary key
        $table->increments('id');
        // Create a column for the title
        $table->string('title');
        // Create a column for the description
        $table->text('description');
        // Create special fields “created_at” and “updated_at” to store timestamps
        $table->timestamps();
    });
}

public function down()
{
    // If the migration is reversed, drop the table
    Schema::drop('services');
}
...
```

The migration definition has been created but to translate the above migration into a “services” table in the database you need to run the migration by executing artisan’s migrate command from the command line:

```
php artisan migrate
```

Keep in mind that the operations with the database will use the database connection settings you have provided in the database.php settings file. If those settings were set correctly you should see a success message in the terminal, something like “Migration table created successfully. Migrated: timestamp_create_services_table”. Now if you check your database you will see that Laravel created the table “services” with the fields you specified in the migration file.

To use the “services” table in the application we need to make Laravel aware of what DB tables are available for use and the way to do that is to define data models for each separate kind of data that the application will be working with. Laravel comes with a powerful Object-relational mapping called “Eloquent ORM” that makes working with the data in databases very easy. Eloquent ORM has a few conventions that we will make use of.

The first convention is that Laravel uses an auto-incrementing integer with the name “id” as the primary key for the records in tables. Although this could be easily changed if needed, this is exactly what we have already so we won’t have to modify our existing table.

Another convention is that Laravel likes your tables to be plural but the data models for the tables to be singular (this too is flexible). As an example if you have a table called “products” in the database, your model

should be named “product”, if you have table called “users” your model for that table would be named “user”. For this particular application we need to create a data model that will make Eloquent ORM aware of the “services” table and the way to do that is to create a model called “Service” in the “app/models” directory, create a new file called Service.php with the contents of listing 2.6 and place it into “app/models” folder:

Listing 2.6 Data model for services offered by the company

```
<?php
```

```
class Service extends Eloquent {}
```

This is all that is needed to tell Laravel about the “services” table in your database. You do not need to specify the name of the “services” table anywhere in the application. Laravel will assume that there is a table named as a plural of the class “Service” in the “app/models” folder. Now when you need to access the data from the “services” table throughout the application you will be able to use Eloquent’s powerful methods of accessing and managing data.

2.6.6 Filling the database with data

You have the “services” table structure ready but the table is empty and there is no “services” that we can show on the “services” page. You should fill the “services” table with information about the services that the company offers. This can be done in many ways. For example you can use PHPMyAdmin tool or the command line to create the rows for “services” table. But for the purpose of exposing you to Laravel’s power and might we will use one of Laravel’s tools to fill the data. Laravel has a built-in way to populate the data in the database. Populating a database is called “seeding” the database.

We will seed the database by creating a class called “ServiceTableSeeder” with a single function “run” that will simply create three different services that will be later displayed on the site. Please create “ServiceTableSeeder.php” in the “app/database/seeds” folder of the application and place the code from listing 2.7 in it:

Listing 2.7 Database seed for “services” table

```
<?php
```

```
class ServiceTableSeeder extends Seeder {

    public function run()
    {
        Service::create(
            array(
                'title' => 'Web development',
                'description' => 'PHP, MySQL, Javascript and more.'
            )
        );

        Service::create(
            array(
                'title' => 'SEO',
                'description' => 'Get on first page of search engines with our help.'
            )
        );
    }
}
```

```

    )
);

Service::create(
    array(
        'title' => 'Marketing',
        'description' => 'Advertise with us.'
    )
);
}
}

```

After creating the seed class it needs to be executed in order for the data to appear in the database. Laravel's DatabaseSeeder class located in "app/database/seeds" folder is the right place to call the execution of the new ServiceTableSeeder class. Let's open up the "DatabaseSeeder.php" and add a single line to its run function (listing 2.8):

Listing 2.8 Contents of modified app/database/seeds/DatabaseSeeder.php file

```

<?php

class DatabaseSeeder extends Seeder {

    public function run()
    {
        Eloquent::unguard();
        $this->call('ServiceTableSeeder'); // Execute the ServiceTableSeeder
    }
}

```

Now that we have the seed defined and the DatabaseSeeder configured to run it, we need to execute Laravel's seed command in the command line and it will create the data in the database:

```
php artisan db:seed
```

If you see a message that says "Database seeded!" then all is well and the content you have defined in the "ServiceTableSeeder" class will be translated into the data in the "services" table. Check your database to make sure the seeding went well and if it did, let's continue developing our application and start showing the data in the browser!

2.6.7 Creating HTML and Blade views

While we have our application's internals mostly finished, the application so far doesn't have any face to it. Right now all we show to the user is the blank screen with a line of text coming from the application's routes. Let's improve that by creating HTML templates for displaying information on each page and for presenting the navigational menu of the site. Then when we have the view templates all ready we will get routes to display these templates in the browser (figure 2.8):

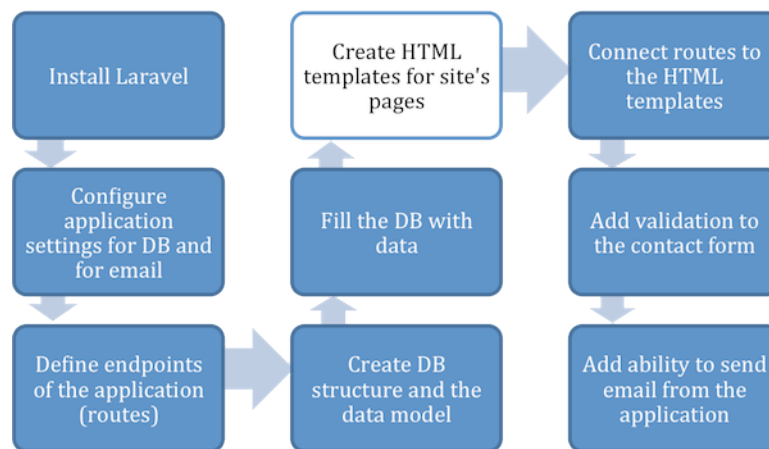


Figure 2.8 Create HTML templates

Laravel comes with a simple yet powerful templating engine called “Blade”. The templates made with Blade look much like plain HTML but with sprinkles of some special statements that make it possible to output PHP data, looping over it, inject other templates, use different layouts and more. Using Blade can definitely save you time and spare from headache when it comes to creating the application’s pages and maintaining the visual aspect of the application.

Application’s view templates are located in the “app/views” directory and files that use Blade syntax end with “.blade.php” in the file name. One of the most useful features of Blade is ability to use a common layout for desired templates of the site and we will make a simple layout that all of the pages of the site will share. Let’s again look at the desired look of the company site in figure 2.9:

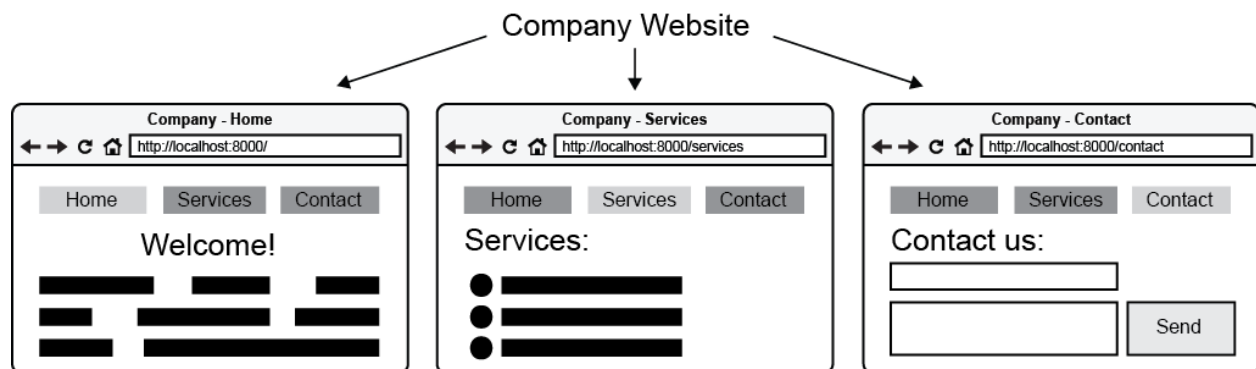


Figure 2.9 A wireframe of company website's pages

2.6.7.1 Building the layout

From this approximate sketch we see that the navigational menu is common to all three pages and the content that goes below the menu is different between the pages. We will build a layout that will allow us to keep the common elements, in this case the site’s navigation in one place and the part that changes, the content, in another. Let’s create the application layout by making a new file in the “app/views” directory, call it “layout.blade.php” and the contents of it will be as follows in listing 2.9:

Listing 2.9 Blade Layout for the company website (layout.blade.php)

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Company website</title>
</head>
<body>
    <ul>
        <li><a href=".">Home</a></li>
        <li><a href="./services">Services</a></li>
        <li><a href="./contact">Contact</a></li>
    </ul>

    @yield('content')

    Company, {{ date('Y') }}
</body>
</html>
```

2.6.7.2 Building the template for the home page

We have the layout for the site ready. Now let's create the templates that will be inserted into this layout. These templates will also go into “app/views” directory and they will be used to display content onto the page. First, let's create the template for the home page as in listing 2.10 and place it in “home.blade.php” file:

Listing 2.10 Blade template for the home page (home.blade.php)

```
@extends('layout')

@section('content')
    <h1>Welcome!</h1>
    <p>We are an established digital agency serving clients that want professional and affordable web development and marketing services.</p>
@stop
```

2.6.7.3 Building the template for the services page

As we specified before, the services page will display the data from the database so the template for this page will need a bit more explanation. One of the best practices of web development is to never do calls to the database from the view templates. Therefore we will do the database calls elsewhere and then pass the retrieved data as objects to the view template. We can then display the data using Blade's clean and easy to read syntax. Let's create “services.blade.php” file in the “app/views” directory and have the contents of listing 2.11 in it:

Listing 2.11 Blade template for the services page (services.blade.php)

```
@extends('layout')
@section('content')
    <h1>Services.</h1>
    <p>The list of the services we offer is below:</p>
    <ul>
        @foreach($services as $service)
            <li>
                <p>{{ $service->title }}</p>
                <p>{{ $service->description }}</p>
            </li>
        @endforeach
    </ul>
@stop
```

That's it for the services page! No messy PHP, no database calls from the view template, this looks nice and clean. Let's build the template for the last page of the site, the contact page!

2.6.7.4 Building the template for the contact page

The contact page will feature the form that will consist of three inputs, the text input for the subject of the message, the text area for the body of the message, and the submit button. While we could hand code the HTML form, it could be faster to use some of the Laravel's built-in methods for building the forms in the templates. As a demonstration, the complete HTML for the form page would look like listing 2.12:

Listing 2.12 HTML code of the contact page without using Blade

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Company website</title>
</head>
<body>
    <ul>
        <li><a href=".">Home</a></li>
        <li><a href="./services">Services</a></li>
        <li><a href="./contact">Contact</a></li>
    </ul>

    <h1>Contact Us.</h1>
    <p>For any inquiries, please send us a message using the form below:</p>
    <form method="POST" action="http://localhost:8000/contact" accept-charset="UTF-8">
        <input name="_token" type="hidden" value="yVsUciVnTw9SqwXZa76vBOUeWuoitZWYIKiUY8Lj">
        <label for="subject">Subject</label>
        <input name="subject" type="text">
        <label for="message">Message</label>
```



```

        <textarea name="message" cols="50" rows="10"></textarea>
        <input type="submit">
    </form>

    Company, 2013</body>
</html>

```

This is quite a bit of code. Let's see how Laravel can make it easier for you to create a form for the contact page. Laravel provides multiple shortcuts for opening and closing forms, creating labels, text inputs, buttons, selects and much more. Using these shortcuts will make the code cleaner and easier to modify. Let's look at this aspect of Laravel in action by creating a template, "contact.blade.php" in "app/views" folder, and by putting the following contents in it (listing 2.13):

Listing 2.13 Blade template for the contact page (contact.blade.php)

```

@extends('layout')
@section('content')
    <h1>Contact Us.</h1>
    <p>For any inquiries, please send us a message using the form below:</p>
    {{ Form::open() }}
    {{ Form::label('subject') }}
    {{ Form::text('subject') }}
    {{ Form::label('message') }}
    {{ Form::textarea('message') }}
    {{ Form::submit() }}
    {{ Form::close() }}
@stop

```

As you can see we have achieved the same HTML output with a lot less code by using Laravel's Blade templating engine. At this point the view templates are complete and we would like to display them in the browser by adjusting our routes to show the templates instead of text strings. Let's do that in the next section!

2.6.8 Displaying view templates from the routes

In section 2.6.4 we left the routes in a state where they are just displaying simple text strings. Of course the time of the blank pages with a single string is now over and we will connect the routes and the view templates together. For simplicity, we can use Laravel's "View::make()" functionality to render the templates when the pages are requested in the browser.

2.6.8.1 Connecting the home page route to the view template

Let's start with the index route, as it is the most basic one, the route for showing the site's homepage. Since all this route is doing is showing a view template, we can call Laravel's "View::make()" function providing it with the name of our view template and it will render the template upon request. This is how it will look like in the "app/routes.php" file after replacing the text string output (listing 2.14):

Listing 2.14 Index route modified to display template for the home page

```
...
Route::get('/', function()
{
    // Return a rendered template for "home.blade.php" file
    return View::make('home');
});
...
```

Returning a rendered view from a route will simply display it in the browser. Now if you visit the homepage of the application in the browser you should see a page that looks similar to figure 2.10:

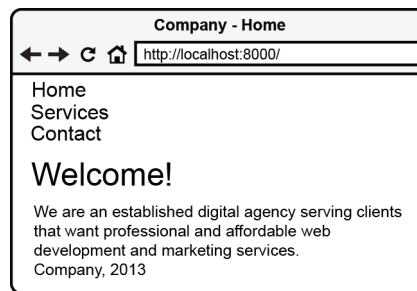


Figure 2.10 Content of the site's homepage displaying from the Blade template

This looks good! Even though there is no CSS to make the pages look pretty, we have a functional home page for the site now. Two pages of the site are left to be made functional, the services page and the contact page. Let's finish up the services page.

2.6.8.2 Connecting the services page route to the view template

As you remember the services that the company offers are stored in the database and we have even created the view template for the services page. That page expects to receive an array of objects that it can then iterate through to get the title and description of each service. We will use Eloquent ORM to retrieve all services from the database and then we will pass the result to the services template. Please look at the listing 2.15 to see how this looks like in action inside of a route and modify your existing "services" route to match this one:

Listing 2.15 Services route modified to display data from services table

```
...
Route::get('services', function()
{
    // Retrieve all services from the services table using the "Service" data model
    $services = Service::all();
    // Pass the variable containing the services to the
    // "services.blade.php" view template
    return View::make('services', array( 'services' => $services));
});
...
```

With these changes the services route should now be fully functional and it should display the page containing the services that we previously inserted into the database (figure 2.11):



Figure 2.11 Content of the services page displaying the list of services from the database

This was easy, wasn't it? In other frameworks or using PDO in plain PHP retrieving records and getting various data attributes would be a bit more involved but Laravel makes development enjoyable by providing logical methods and by forcing solid yet flexible conventions.

2.6.8.3 Connecting the contact page route to the view template

The only page left for us to be developed is the contact form and we are ready to modify the route that leads to it so that it displays the template we created earlier. Replace the existing "contact" route with this one (listing 2.16):

Listing 2.16 Modified route for the contact page

```
...
Route::get('contact', function()
{
    // Return a rendered template for "contact.blade.php" file
    return View::make('contact');
});
...
```

Laravel will convert the Blade template we created for the contact page into HTML along with form we crafted using the "Form" methods. Now going to the "contact" URL in the browser should show a page similar to figure 2.12:



Figure 2.12 Contact page with the form rendered from the view template

We are almost there. The form is on the contact page but what happens when we try to submit it? We will receive an error stating that the requested page is not found. That is because when the form is submitted

it is POSTed to the “contact” route and currently we do not have the route definition for the POST method of “contact” route. Let’s create that. We will add one more route in the “app/routes.php” file, as in listing 2.17:

Listing 2.17 Adding a POST route for the contact page at the end of routes

```
...
Route::get('contact', function(){...});

// Notice the POST method on the route
Route::post('contact', function()
{
    return 'Message sent';
});
...
```

Voila! Now when we submit the form on the contact page we will get a message that says “Message sent”. That is kind of a lie because we don’t send anything yet, but we are optimistic. We would like to receive the values submitted by the form and email the contact request to the administrator of the company site. One problem with that is that the values provided by the user could be empty. How can we prevent submittal of an empty form or a form with invalid values? Dear reader, enter validation!

2.6.9 Adding validation to the contact form

Validation of input tells us if the data entered by the user is valid or not valid. Applying validation to form data is very easy in Laravel. We will discuss it in detail in later chapters but for now we will do the following process to verify that the submitted form data is valid:

- Gather the user’s input
- Define some validation rules for the input fields
- Validate the input according to the validation rules
- If the data is valid, proceed doing something useful, otherwise come back to the form displaying what validation errors have occurred and also preserving user’s input

With this process flow implemented and with validation in action, the “contact” route will look like the code following in listing 2.18:

Listing 2.18 Adding validation to the form input in the “contact” route

```
...
Route::post('contact', function()
{
    // Gather all user’s input
    $input = Input::all();

    // Create an array of validation rules for the input fields
    $rules = array(
        'subject' => 'required',
        'message' => 'required'
    );
});
```

```
// Apply the validation rules to the input
$validator = Validator::make($input, $rules);

// Go back to the contact route preserving the user's input in case of
// failed validation
if($validator->fails()) {
    return Redirect::to('contact')->withErrors($validator)->withInput();
}

return 'Message sent';
});
...
```

When the validation of the input data fails, the application will redirect the user back to the contact page remembering the user's input and also storing validation errors in the session. One small thing that we now need in our view template is to add the display of the error messages that arise when the validation does not pass. We can just display the errors in a string but another way is to loop through the errors using Laravel's HTML helpers, open up the "contact.blade.php" template file from "app/views" folder and add this line above the form methods:

```
{{ HTML::ul($errors->all(), array('class'=>'errors')) }}
```

Please see the listing 2.19 to verify that it is in the right place:

Listing 2.19 Addition of the validation error messages (contact.blade.php)

```
...
<p>For any inquiries, please send us a message using the form below:</p>

{{ HTML::ul($errors->all(), array('class'=>'errors')) }}

{{ Form::open() }}
...
```

Validating user's input and showing error messages will prevent an empty form to be submitted and thus will decrease the number of empty emails the site administrator will be getting. Speaking of emails, as of right now we are not sending anything even when the user has filled the input fields properly. Why don't we fix that right away? Let's learn how to send emails from our application!

2.6.10 Sending HTML email with Laravel

Laravel makes sending HTML emails almost too simple. Let's first look at how we could send an email using plain PHP using the built-in "mail" function and then we will build the same functionality using Laravel's methods to see what benefits there are to using Laravel's email features.

We will send the contact request email to the site admin after the validation of the contact form passes. To do that in plain PHP we could add it to the POST route of the "contact" route as follows in listing 2.20:

Listing 2.20 Sending email in the “contact” POST route using plain PHP

```
Route::post('contact', function(){
    ...
    if($validator->fails()) {
        return Redirect::to('contact')->withErrors($validator)->withInput();
    }

    $to = 'Site admin <my@email.com>';
    $subject = 'Contact Request';

    $emailContent = '
    <!DOCTYPE html>
    <html lang="en-US">
        <head>
            <meta charset="utf-8">
        </head>
        <body>
            <h2>Contact request</h2>
            <div>Somebody sent a contact request, details:</div>

            <div>Subject: '.$input['subject'].'</div>
            <div>Message: '.$input['message'].'</div>
        </body>
    </html>';

    $headers = 'MIME-Version: 1.0' . "\r\n";
    $headers .= 'Content-type: text/html; charset=iso-8859-1' . "\r\n";
    $headers .= 'To: '.$to."\r\n";
    $headers .= 'From: '.$to."\r\n";

    mail($to, $subject, $emailContent, $headers);

    return 'Message sent';
});
```

This is good enough but the code is pretty messy and not easy to maintain. If you need to change the content of the email, recipients, add CC and BCC to the email it will not be an easy task. Laravel alleviates these problems by separating the presentation of the email from the functional part (sending it, specifying recipients, etc). Let’s achieve the same functionality but using Laravel’s easy to read methods.

We already configured the email driver to use PHP’s built in “mail” function and specified the “from” address and name (refer to section 2.6.2 to see how we did that). Now all we need is to specify which HTML template will be used for the emails and what data it will contain. Let’s create a basic view template in “app/views/emails” and call it “contact.blade.php”, the content of it will be as follows in listing 2.21:

Listing 2.21 Template for the HTML email (contact.blade.php in app/views/emails)

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <h2>Contact request</h2>
    <div>Somebody sent a contact request, details:</div>

    <div>Subject: {{ $subject }}</div>
    <div>Message: {{ $request }}</div>
  </body>
</html>
```

This template will be sent out when the contact form is submitted and when the form input passes validation. We will use Laravel’s “Mail” functionality to send the HTML email, let’s go back to our “app/routes.php” file, prepare the data for the email template and add a call to “Mail::send” function (listing 2.22) after the validation:

Listing 2.22 Sending email in the “contact” POST route using Laravel’s mail methods

```
...
    if($validator->fails()) {
        return Redirect::to('contact')->withErrors($validator)->withInput();
    }

    // Prepare an array of data that will be passed to the email template
    $data = array(
        'subject' => $input['subject'],
        'request' => $input['message']
    );

    // Use Laravel’s “Mail::send” method to
    // send “app/views/email/contact.blade.php” template
    Mail::send('emails.contact', $data, function($message)
    {
        // Specify the email address and the name of the recipient
        $message->to('my@email.com', 'Site admin')
            ->subject('Contact Request');           // Specify subject line of the email
    });

    return 'Your request was sent!';
});
...
```

Now when everything is done we can test if the contact form works properly. Go to the “contact” URL of the application, type in a subject and a message body for the contact request and hit “Submit”. If the form passed validation and if you see the message “Your request was sent!”, all went well and your email should be well on its way to the inbox you specified. Check your spam folder if you aren’t seeing the email in the inbox after few minutes, sometimes email providers mark emails sent from “localhost” domain as spam. Congratulations, you just sent a real email from your Laravel application!

By now we have all three pages of the company site working! The home page shows a welcome message. The menu works and links to other pages of the site. The services page shows the list of services from the database and the contact form on the contact page is functional and even prevents the user from submitting an empty form. This concludes the step-by-step guide of making a complete application with Laravel and this is just the beginning of a very fun journey!

2.7 Summary

You have learned a lot in this chapter. You have seen that installation of Laravel through Composer is easy, new Laravel project can be created by executing a single command in the command line. You have met Laravel’s command line interface tool “Artisan” that accelerates development of applications by running various tasks like creating migrations and seeding a database.

Applications built with Laravel consist of many components that the developer has to either adjust or create following a certain convention. You have looked at the contents of a fresh Laravel installation and learned about the responsibilities of each file and folder, including the application folder and the configuration folder. Finally, you have created your own application, complete with routes, database structure, data seed, models, view templates, sending email and more.

In the next chapter we will look into the concept of routing in Laravel and will learn about its importance in each application built with Laravel.

3. Routing

This chapter covers

- Basics of routing
- Passing parameters to routes
- Route filters
- Grouped and named routes
- Redirects

When you come to a coffee shop and ask a barista for a cup of coffee, you wouldn't be too happy if he gave you a bottle of hot sauce or would just quietly stare at you instead not even acknowledging that you are there or that your order has been received. Like the process of ordering coffee at a coffee shop through a barista, every web application should have final and desired destinations in its flow. When visitors come to your website or web application, they expect to receive what they requested. If they go to their user profile page and they see an empty or wrong page instead, frustrated and disappointed they might not come back to your application again.

By now you already have the general idea of how a Laravel application is built and you might be aware of how Laravel directs the flow of the application by the means of routing. Laravel has great routing capabilities, it provides the developer with instruments to create and fine tune application's responses to requests. In this chapter you will learn in detail about Laravel's powerful routing mechanisms and how you can set up those final destinations - routes in your web application to make your users happy. You will also learn about sending very specific requests to your application by means of route parameters and how Laravel can understand and process them. Finally, you will learn about protecting routes with route filters, grouping routes and making routes respond with the expected results.

3.1 Introduction to Routing in Laravel

As users or API's interact with your application they will visit different pages, different destinations of the application, which in web development talk, are called "routes".



What are routes?

Routes define the endpoints of a web application, the final destinations of an incoming request.

Imagine that you are trying to create a login page for your application. Without defined routes your application simply will not know what to do when it receives a request from a user. Where should a request go? How should the application respond to browser's requests? The routes act as virtual roads and pathways that the user's requests can travel through, and your application will need to know where the requests can and cannot go. If your application needs to show a login page it will need to have a route defined for it. Also most likely there will be some processing or validation of the data entered on the login page. That also requires specifying a route. In Figure 3.1 Let's take a look at the result that we would like to achieve:

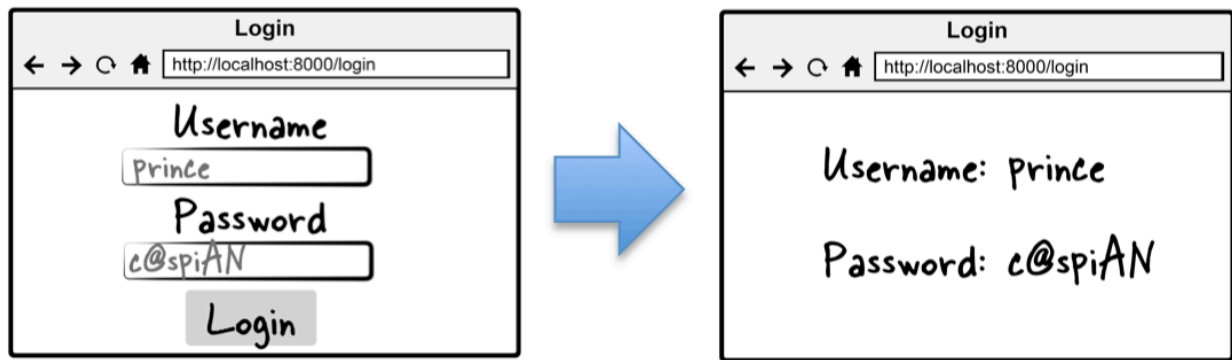


Figure 3.1 Simplified login page flow

In Laravel, the routes for a web application reside in “app/routes.php” file and are evaluated in the order from the bottom to the top. This is the file that you will edit each time you want to add or delete a route to your application. Having a brand new installation of Laravel, open up the “routes.php” file inside “app” folder and you will see that it has a single basic route already set up for the application’s index page (listing 3.1):

Listing 3.1 Contents of routes.php file in a fresh Laravel installation

```
// Route that is executed when the user opens up the index page in the browser
Route::get('/', function()
{
    return View::make('hello');
});
```

Remember when you first installed Laravel and opened the index page of your application in the browser, you saw a simple page with Laravel logo and text that said: “You have arrived”? The route above is the route that your application followed to show that first “hello” page (figure 3.2).

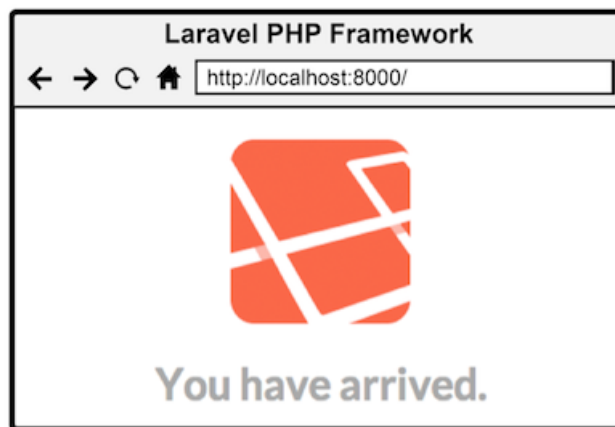


Figure 3.2 Laravel’s response to the index route

3.1.1 Structure of a basic route

A basic route like the index route already present in a fresh Laravel installation consists of three parts, the method of the incoming request (get, post, put, patch or delete), the destination of the route and the function

that will be executed when the route destination is reached. Let's take a look at an overview of route's structure in figure 3.3:

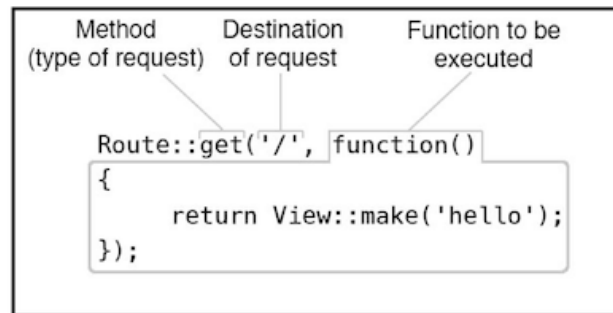


Figure 3.3 Structure of a basic route

The destination of the request could be one of the following:

- A single slash, which is the root of the application
- A single word like “users”, “posts”, “about” and etc.
- Words separated with a slash like “user/profile”, “tags/search”, “pages/company/profile” and etc

In plain English, the route shown in the figure 3.1 could be translated as:

"When I go get the index page create the view "hello" and serve it to me."

The `View::make` displays a Blade template specified in its parameter. Don't worry if this doesn't seem familiar to you. I will talk about Blade templating in later chapters but for now you just need to know that `View::make` shows an HTML page.

If you want to replace the destination with something other than the root of the application, for example with “coffee”, simply place “coffee” in the destination string parameter instead of the forward slash (listing 3.2):

Listing 3.2 Laravel application responding to request for “coffee”

```
// The index route destination ('/') has been replaced with "coffee"
Route::get('coffee', function()
{
    // The response from the route is still the same
    return View::make('hello');
});
```

Now if you go to this modified route in the browser you will still see the default Laravel welcome page but at a new destination - “coffee” (figure 3.4). Knowing this you can now easily define a route for your barista application and be able to process a simple request for “tea”, “latte” or even “beer”!

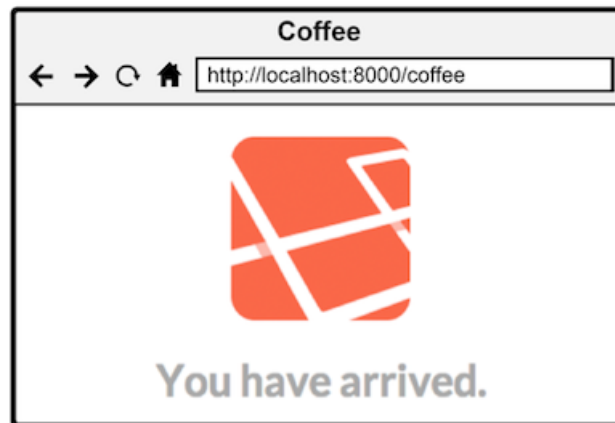


Figure 3.4 The default root route of the application has been replaced with “coffee”

3.1.2 Types of route requests

Web applications can do a lot more than just displaying pages. The routes with the method “GET” are mainly used to request some HTML content from the user’s browser. But what if you wanted to submit a form instead of showing a page? You might know that there are other types of HTTP request methods that are more appropriate. Some of the most popular HTTP methods used in web applications today are:

- GET
- POST
- DELETE
- PUT
- PATCH
- TRACE
- HEAD

Laravel supports all of these HTTP methods in its router. The first five (GET, POST, DELETE, PUT and PATCH) are more than enough for vast majority of web applications. Technically we can even get by with using just the first two methods (GET and POST) for retrieving and submitting data but when you deal with things like APIs or frontend MVC frameworks like Backbone.js or AngularJS, you will see why there is a need for the rest of the HTTP methods. Table 3.1 shows a summary of HTTP request methods that you might use in your Laravel applications and the purpose of each method along with a simplified example.

Table 3.1 Most popular HTTP request methods supported in Laravel

Method	Purpose	Examples
GET	Retrieving data	Showing HTML page or echoing JSON content
POST	Sending data	Submitting contents of a form
PUT	Updating a resource	Updating a blog post or a comment
PATCH	Modifying parts of a resource	Updating a paragraph of existing blog post or a detail about the user
DELETE	Removing a resource	Deleting a blog post, an image or a comment.

3.1.3 Routing example: login form

I imagine that your Laravel applications will not only display static content and HTML pages but will also provide a certain degree of interaction between the user and the web application. For example the user might be presented with a login form and submit his or her username and password to your application. The routing flow to show the login page to the user is shown in figure 3.5:

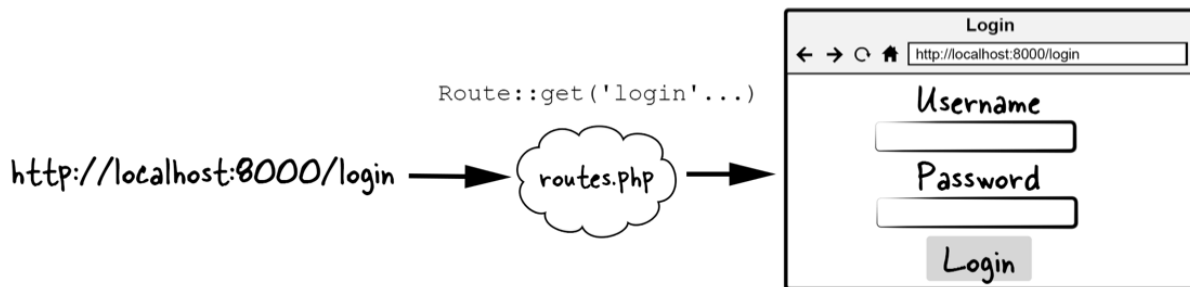


Figure 3.5 Routing flow that shows a login page

Applying the concepts that we have learned in sections 3.1.1 and 3.1.2 let's build routing for such application flow. We know that this simple application will need to show a login form. When the form is submitted by the user, it will send the entered data to the application. For purpose of seeing the routing concept in action, upon submittal of the form we want to just display the submitted data back to the user. Please see figure 3.6 that shows the flow of the form submission:

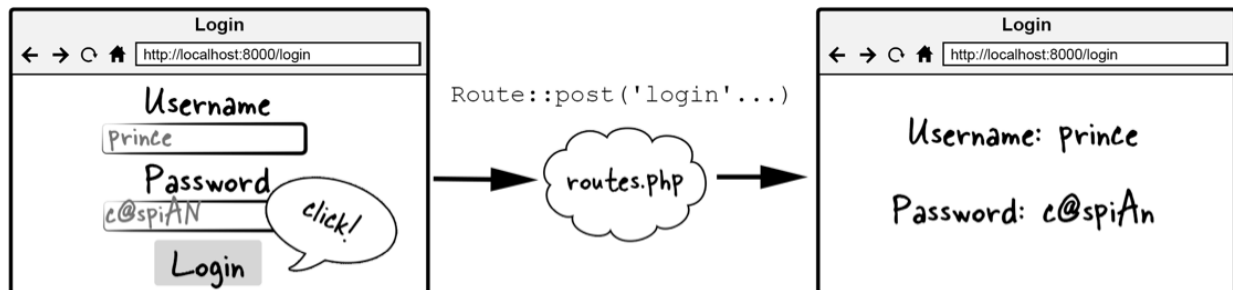


Figure 3.6 Routing flow for login form submission

3.1.3.1 Building routing structure for the login form example

Let's build the routing endpoints of the login form:

- A route that shows the login form
- A route that receives the submitted form and displays data back to the user

Using the proper HTTP methods from the table 3.1 we will need to use “get” method to request the login page and “post” method to process the form submission. First, we will build a skeleton for these two routes that will hold the logic of serving the form and processing the form (listing 3.3):

Listing 3.3 Structure of routes that display and process a login form (“app/routes.php”)

```
// Route definition for showing the login page
Route::get('login', function()
{
    // Display a placeholder text to the user
    return 'login form';
});

// Route definition for processing the login form
Route::post('login', function()
{
    // Display a placeholder text to the user
    return 'processing the login form';
});
```

The code in listing 3.3 will show the text ‘login form’ when you go to the ‘login’ route of your application. If you were to execute a POST request to the same route, you would see the text that says “processing the login form”. This is almost functional but we would like to see an actual form, don’t we?

3.1.3.2 Showing HTML form and processing user input

The simplest way to show HTML from the routes of the application is to just return the HTML as a string of text from the route. While a more proper way of doing this would be to create a view template and use Laravel’s Blade engine to display a form, we will resort to just outputting HTML straight from the route for the sake of simplicity. Let’s look at the code of listing 3.4 that displays the login form to the user.

Listing 3.4 GET Route that displays the login HTML form

```
// The HTML of the login form is returned from the route
Route::get('login', function()
{
    return '<form action="login" method="post">
        Username: <input type="text" name="username"><br>
        Password: <input type="password" name="password">
        <input type="submit" value="Submit">
    </form>';
});
```

When the user submits this form from the login page, a request of type “POST” will be made to the login route, let’s just display entered information back to the user (listing 3.5):

Listing 3.5 POST Route that processes the login form input

```
// Capture submitted data and display it back to the user
Route::post('login', function()
{
    return 'Username: '.$_POST["username"].', Password: '.$_POST["password"];
});
```

Try visiting the “login” route in your browser at “http://localhost:8000/login” and submitting a username and password, you should get back the data that you have entered just like we saw in figure 3.4. Please note, this code might not look pretty because we are not using the proper Laravel instruments to show the form (Blade templates) and to capture the form data (Laravel’s `Input::get` method). In later chapters you will learn how to build the same functionality in a much cleaner way.

Laravel is not limited to just responding to requests. There could be times when you will need to make your routing more dynamic or impose some limits on what the routes will respond to. The most prominent of Laravel’s instruments in its routing arsenal that will help you achieve a degree of protection and dynamism are route filters and route parameters. Route filters allow you to execute a specific function before or after a certain route is executed. This can be incredibly helpful when you are trying to limit certain routes to only logged in users or to users possessing a certain criteria. I will elaborate on the route filters a bit later in this chapter, but for now I would like to introduce you to one of the most often used instruments of Laravel’s routing - passing parameters to your routes.

3.2 Passing parameters to routes

Route parameters allow the routing of your application to become dynamic and adapt to the data that your application is working with. The data in your application most likely will not be all hard coded. Some of it will be stored in a database; some of it will be stored in sessions, some will be passed down from one part of the application to another. When your application becomes alive and filled with information, there will arise a need to access specific bits and pieces of that data.

Referring back to the coffee shop barista example from the introduction to this chapter, let’s say you ask a barista for a specific kind and size of coffee: “I would like a Grande Latte”. How can you make a web application understand specific details about the requested data (size – “Grande”, kind – “Latte”) and respond back to you differently depending on those details?

The answer is: you can do it by using route parameters. You need to somehow tell the application what type of resource you are requesting or provide details about the resource in some way. One way of doing that is to have a form on each page of your application and submit that form that contains all details about the data. That is a very bad, outdated practice and I do not recommend you do that! Instead of having everything in forms, we can utilize something that a server already understands and gets from you as soon as you access a web page – the requested URL. That’s right, the address bar in your browser can provide you web application with a lot of data. Let’s examine a sample URL structure of a request to a barista-like application that we would like to achieve (figure 3.7):

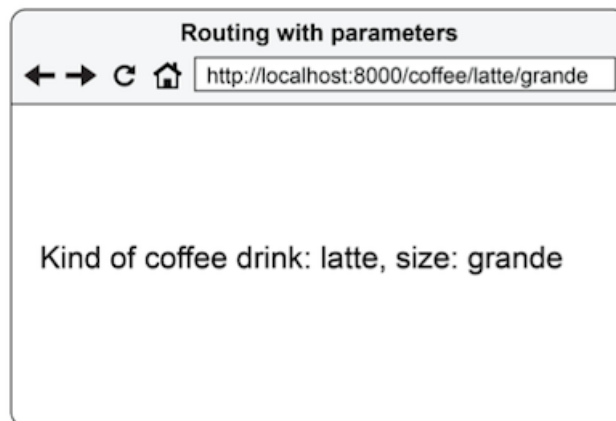


Figure 3.7 URL with route parameters

From a URL like that, “http://localhost:8000/coffee/latte/grande”, we can see what exact resource we would like to request from the barista application. We have provided the details about what type of resource we are requesting (coffee), the kind (latte), and the size (grande). Going to a URL like this without specifying the routing for it in the application will throw a “Not Found” exception because first we need to define the routes for this type of request.

In the previous section you have learned that you can have words separated by a slash in your destination URL for the routes. So one way to achieve routing for specific kind and size of coffee would be to declare a route for each possible kind and size of coffee, something like the code of listing 3.6:

Listing 3.6 Inefficient example of routing to specific kinds and sizes of coffee

```
// Route to coffee/latte/grande
Route::get('coffee/latte/grande', function()
{
    return 'Grande Latte';
});

// Route to coffee/latte/venti
Route::get('coffee/latte/venti', function()
{
    return 'Venti Latte';
});

... // 57 more routes (!)

// Route to coffee/espresso/grande
Route::get('coffee/espresso/grande', function()
{
    return 'Grande Espresso';
});
```

This type of routing would quickly get out of hand and become nearly impossible to manage if you have 20 different kinds of coffee drinks and 3 different sizes for each kind. That would be 60 routes in total! Isn’t

there a better way of doing this in Laravel? There sure is, and it is route parameters that you could use to make this routing better.

3.2.1 Using route parameters

It would be nice if you could just give that function inside of the route definition some variable along with some sort of template in the route destination that defines where that variable can occur and Laravel would automatically make that into a dynamic route that responds to different URLs but has the same behavior for all of those URLs. Guess what, this is exactly what Laravel can do. The code in listing 3.7 shows how to use the route parameter to display any kind of coffee drink:

Listing 3.7 Route with a parameter

```
// Passing a parameter to the route "coffee" to dynamically
// specify a kind of coffee.
Route::get('coffee/{kind}', function($kind)
{
    // Displaying the variable back to the user
    return 'Requested kind of coffee drink: '.$kind;
});
```

Now if you go to any URL like coffee/espresso or coffee/latte or coffee/cappuccino (relative to application's URL), your application will tell you which kind of coffee you have requested. No more statically declared routes that know only one thing. This is so powerful!

3.2.1.1 Passing more than one parameter

Even though we have achieved some degree of dynamics in our barista application, we have not accomplished our goal, we still cannot ask for a specific size of our coffee drink. We are not limited to passing just one parameter however, in the same way of passing a single parameter we can pass two, three or as many as we would like by defining them in the destination string template and passing them as the arguments to the function inside of the route, like in listing 3.8:

Listing 3.8 Route with multiple parameters

```
// Passing two parameters to a single route
Route::get('coffee/{kind}/{size}', function($kind, $size)
{
    // Displaying the variables back to the user
    return 'Kind of coffee drink: '.$kind.', size: '.$size;
});
```

Voila, now you can request a coffee/espresso/tall, or coffee/latte/grande or coffee/cappuccino/small or any other size or kind of coffee and your application will know what to do with it based on our route destination template.

3.2.2 Route constraints

In some cases you will want to have only certain types of destinations be reachable. For example you might want the size of the coffee to only be a number symbolizing how many ounces of coffee was requested.

Laravel gives you quite a bit of flexibility about limiting the route parameters to be conforming to only specific patterns. One way of constraining what the route parameter can be is to use Regular Expression constraint attached to the end of the route (listing 3.9):

Listing 3.9 Route with multiple parameters and with a constraint

```
// Constraining route parameter with a regular expression pattern
Route::get('coffee/{kind}/{size}', function($kind, $size)
{
    return 'Kind of coffee drink: '.$kind.', size: '.$size.'oz';
})->where('size', '[0-9]+');
```

Now if you try to go to this URL : coffee/espresso/tall you will encounter a “route not found” error. That is because you are using a route constraint limiting the “size” parameter to only numbers. Try reaching a different URL, coffee/espresso/24, that should work and you should see route response like in figure 3.8:

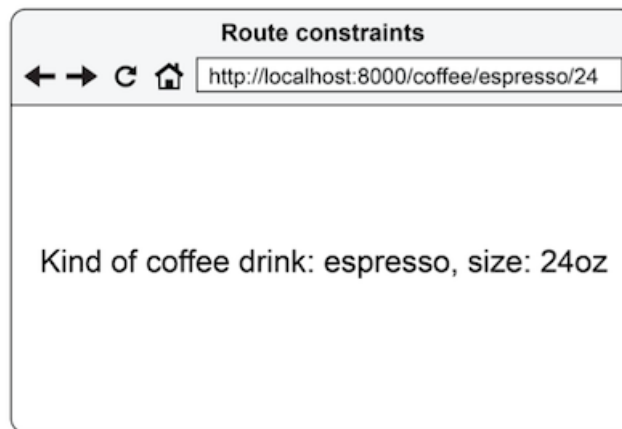


Figure 3.8 Reaching route with a constraint

This can be helpful to filter out unwanted types of route parameters to be reached. Patterns like only numbers, only letters, combination of the two, and many more can be created with simple regular expression constraints.

3.2.3 Making route parameters optional

Not all route parameters are always necessary. There might be a case when a route parameter is optional. For example you would like to have the size of the coffee drink to be optional and not required. You can place a question mark (“?”) after the route parameter and provide the default value for the matching variable to make that parameter optional (listing 3.10):

Listing 3.10 Route with an optional parameter for the size of the coffee

```
// Making a parameter optional by adding "?" after it and setting
// a default value for it in the function arguments
Route::get('coffee/{kind}/{size?}', function($kind, $size = null)
{
    return 'Kind of coffee drink: '.$kind.', size: '.$size;
});
```

The optional parameter's default value does not have to be a null value. You can set it to be anything, for example 'tall' or '12'. This way your application will not break when the user goes to URLs like:

- coffee/espresso
- coffee/espresso/grande
- coffee/espresso/16
- coffee/cappuccino
- coffee/cappuccino/verylargesize

All of these URLs will work and the application will respond with the kind and the size of the coffee that the user has requested.

Using route parameters with constraints can be very powerful, flexible and dynamic. You can't do without these instruments when you are building applications that have different types of data, APIs or some sort of interaction between the user and your application. What if you wanted to make your routing even more flexible, say you wanted to check that the size of the coffee drink can be only between 12 to 24 oz? Or if you wanted for some routes to be accessible only if the user is logged in? Another routing device up the Laravel's sleeve is to the rescue in those cases. Dear reader, please welcome: Route Filters.

3.3 Route Filters

Route filters serve a purpose of filtering anything that deals with the application's routes, data coming to and from the routes, route parameters, user permissions and more. They are small functions that when attached to routes allow you to fine tune what can be executed in different parts of your application. Route filters are conveniently located in "app/filters.php" file.

Route filters are great for building a basic implementation of an access control list, a technique of managing user's access permissions. In fact, Laravel already comes with a few basic filters that can help you distinguish between users that are authenticated and logged in into your application, and the ones that are not authenticated - "guests".



Please note

Unless the filters are attached to the routes, they will not be executed.

Let's look through the filters that Laravel provides out of the box and in the next section we will learn how to activate them. You can see these predefined route filters when you open up a fresh "app/filters.php" file of a new Laravel installation (listing 3.11):

Listing 3.11 Contents of fresh app/filters.php file

```
<?php

// Global filters that are executed at the beginning and at the end of
// every request to the application
App::before(function($request) {});
App::after(function($request, $response) {});

// Filter that prevents users that are not logged in from accessing the
// content of the application
Route::filter('auth', function()
{
    if (Auth::guest()) return Redirect::guest('login');
});

// Filter that enables the use of basic authentication (great for building API's)
Route::filter('auth.basic', function()
{
    return Auth::basic();
});

// Filter that prevents logged in users from seeing pages that are
// strictly meant for logged in users
Route::filter('guest', function()
{
    if (Auth::check()) return Redirect::to('/');
});

// Filter that compares CSRF tokens on form submission, a useful security feature
Route::filter('csrf', function()
{
    if (Session::token() != Input::get('_token'))
    {
        throw new Illuminate\Session\TokenMismatchException;
    }
});
```

These filters might serve as the basis for your application's security implementations. In most cases these default filters are enough to protect certain application routes from unauthorized use, but as you will see soon, you are not limited to using only what Laravel provides. First though, filters need to be attached to the routes to be activated, let's look at how that could be done.

3.3.1 Attaching filters to routes

You can activate route filters by attaching them to the application's routes. Filters can be executed before route's execution (for example to prevent unauthorized access) or after (for example to log something). The

diagram in figure 3.9 shows in which order the execution will take place:

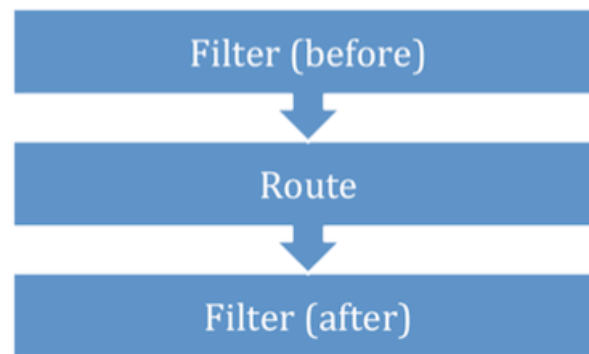


Figure 3.9 Order of execution of route filters relative to the route

The process of attaching a filter to any route in your application consists of two decisions:

1. Deciding whether the filter should be executed before or after route's execution
2. Specifying which filter from “app/filters.php” you want to attach

To better understand the concept of attaching the filters to routes let's look at simple real life example that shows this concept in action. Let's imagine that you want only logged-in users to be able to access application's route for 'coffee', while the guest users trying to access that same route will be taken to the login page. To attach the “auth” filter to the route that leads to “coffee” for the purpose of allowing to view the page for logged in users only, you would do it as follows in listing 3.12:

Listing 3.12 Attaching ‘auth’ filter to route ‘coffee’

```
// The 'auth' filter will be executed before the contents of the route is executed
Route::get('coffee', array('before' => 'auth', function()
{
    // Hello page will be only visible to logged in users
    return View::make('hello');
}));
```

Now if you try to reach the “coffee” route in the browser you will be redirected to the “login” route instead because the “auth” filter in “app/filters.php” file checks if the user is logged in, and if that condition is not met, it redirects the user to the “login” route and is not executing the contents of the “coffee” route.

You might have noticed that the structure for the route that has a filter attached has changed a bit comparing to the basic route structure you saw in figure 3.2. Route's closure function is now inside of an array which is the second parameter of Laravel's “Route::get” function.

We will learn more details about securing your application in later chapters of the book, but this basic concept of attaching route filters such as “auth” is enough to introduce you to the next concept of route filters – creating custom filters!

3.3.2 Creating custom filters

Remember, in our barista application from the section 3.2.3 we wanted to somehow check if the parameter passed to the route as the size of the drink (“coffee/espresso/24”) was within certain limits? In this section we

will explore how you can enforce that limit and in the process you will learn how to create your own route filters.

Laravel allows you to create custom route filters painlessly. In order to attach a custom filter to any of your routes, first you have to define the new filter in “app/filters.php” file. Then after that filter is defined, you would attach it to a route as any of the Laravel’s built in filters.

Let’s create a filter that will check if the third route parameter is within defined limits, call it ‘checksize’, and if the parameter is not within limit of 12 - 24, we will tell the user that the passed parameter is out of allowed range. Add the following filter to “app/filters.php” (listing 3.13):

Listing 3.13 Custom filter added to filters.php

```
Route::filter('csrf', function(){ ... });  
  
...  
// Define a new 'checksize' filter  
Route::filter('checksize', function()  
{  
    // Retrieve the third parameter of the route representing the size of the drink  
    $size = Request::segment(3);  
    if($size < 12 || $size > 24)  
    {  
        // Display an error in case the parameter is out of range  
        return 'Size is not in the allowed range of 12 - 24';  
    }  
});
```

Now that the filter is created, the only thing left is to attach it to the route from listing 3.10. We can do that by specifying the ‘checksize’ filter as the ‘before’ filter (listing 3.14):

Listing 3.14 Attaching custom filter to a route

```
// Filter 'checksize' attached to be executed before the route is executed  
Route::get('coffee/{kind}/{size?}', array('before'=>'checksize',  
    function($kind, $size = null)  
    {  
        return 'Kind of coffee drink: '.$kind.', size: '.$size;  
    }  
));
```

Try going to this route in the browser and make the size parameter (third parameter) bigger than our imposed limit of 24, you should see a page with the filter-specified error message (figure 3.10):

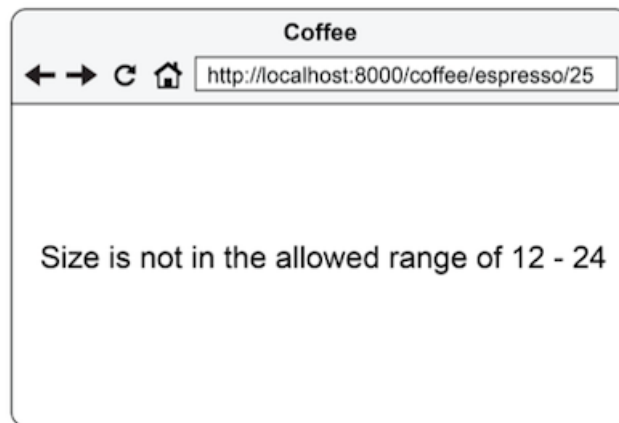


Figure 3.10 Showing an error message from a filter

With this custom filter in place, the user will only be able to access the drinks that have sizes in range 12-24, thus our goal of limiting the passed route parameters to a certain range has been achieved.

3.3.3 Adding multiple route filters

Laravel doesn't stop you from using more than one filter on any of your routes. For example you want to apply the size limit AND make the route accessible only to the users that are logged in. You can easily attach two, three or many more route filters by separating them with a pipe character "|" like in listing 3.15:

Listing 3.15 Attaching multiple filters to a route

```
// Multiple route filters are separated with "|"
Route::get('coffee/{kind}/{size?}', array('before' => 'checksize|auth',
    function($kind, $size = null)
    {
        return 'Kind of coffee drink: '.$kind.', size: '.$size;
    }
));
```

This way our coffee shop barista application will first check the size provided by the user using the 'checksize' filter and if the size is within allowed range, it will check if the user is logged in or not by executing the 'auth' filter. Implementing all sorts of access limiting couldn't get easier than this by means of using multiple route filters.

3.4 Grouping routes

Sometimes when your application grows in size the routing file could also expand significantly. To prevent yourself from cluttering the routing file you could optimize the routing by means of grouping and prefixing routes that start with a common pattern. Imagine that you are trying to build routes for an administration panel of your website. You might have multiple routes all of which start with 'admin' like in listing 3.16:

Listing 3.16 Routing for an admin panel without grouping

```
Route::get('admin/users', function(){ ... });
Route::get('admin/posts', function(){ ... });
Route::get('admin/comments', function(){ ... });
```

When you have some routes that share a common pattern like the example in the listing above, it is a good idea to separate these routes as a group. Grouping routes is be done by taking a pattern common to some of the routes in your routing file and making them into a separate group. Laravel’s “Route::group” method with a ‘prefix’ parameter specifying the prefix for the ‘admin’ route group will accomplish the same task as the routing in listing above. Let’s see how this is done in practice in listing 3.17:

Listing 3.17 Routing for an admin panel with grouping

```
// Prefixing the route group with 'admin'
Route::group(array('prefix' => 'admin'), function()
{
    Route::get('users', function(){ ... });
    Route::get('posts', function(){ ... });
    Route::get('comments', function(){ ... });
});
```

Now all routes that start with ‘admin’ will be automatically mapped to the route group and going to the ‘admin/posts’ will still work the same way as it did in listing 3.16. The biggest advantage of using route groups is the ability to attach filters to the whole group in the same way you would do it with a single route. Let’s say you wanted the user to be logged in to see the admin side of the site, you could attach the ‘auth’ filter to the whole group of routes like in the listing 3.18:

Listing 3.18 Attaching a filter to route group

```
// Attaching 'auth' filter to the whole group of routes
Route::group(array('prefix' => 'admin', 'before' => 'auth'), function(){
    ...
});
```

This Laravel’s instrument makes it very convenient to manage multiple routes that share a same prefix and/or need the same route filters for a group of routes.

3.5 Route responses

As you have learned in the beginning of this chapter, all routes should lead to some sort of a destination in your application. These destinations could be one of the following:

- Displaying something back to the user (HTML, text, etc)
- Directing the flow of execution to another route (also called “redirecting”)
- Passing the execution to a function of another class in the application (also called “Controller routing”)

In Laravel applications, it is absolutely necessary for the route to have a definite conclusion, otherwise your application will not execute properly. You might remember the diagram showing the structure of a basic route (provided again in figure 3.11):

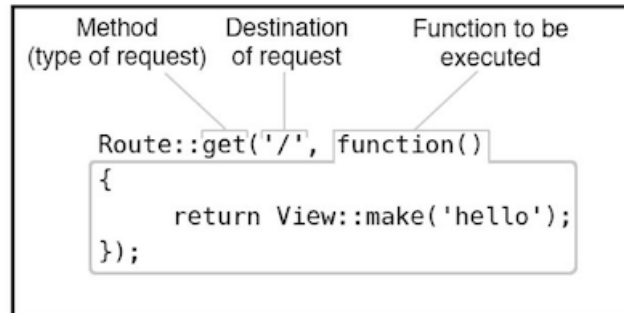


Figure 3.11 Structure of a basic route

As you see, the second parameter of a basic route is a function. As with any other PHP functions, to get a result from executing a function, there has to be a ‘return’ statement. In case with Laravel, this ‘return’ statement could look like the following (listing 3.19):

Listing 3.19 Examples of possible route responses

```
// Responding by showing a Blade template
return View::make('contact');
// OR
// Responding by displaying text
return 'Route response';
// OR
// Responding by redirecting to another route
return Redirect::to('tea');
```

There is another type or response not listed here – responding by passing the execution to controllers. We will look at that type of response in the chapter titled “Controllers”. But first, let’s look at each of these possible responses in more detail. We will look at the first of the possible resolutions of a route, displaying something back to the user.

3.5.1 Showing output

One of the simplest ways to show output to the user is to simply return a string of text. Let’s say you want to if your routes are working properly, you could do that easily by returning a string from the route, listing 3.20 provides an example of that:

3.20 Route showing a string of text to the user

```
Route::get('text', function()
{
    // When the route returns a string, it will be shown to the user
    return 'This is route response';
});
```

You don't have to stop here. You can even display HTML pages by returning an HTML string instead of plain text (listing 3.21).

3.21 Route showing a HTML to the user

```
Route::get('html', function()
{
    // Display HTML to the user
    return '<b>This text is bold!</b>';
});
```

Displaying HTML pages like that is not the cleanest way to render web pages. Laravel provides much more convenient methods of building that HTML string with its “View::make” capabilities that keep your routes clean. We will look more into the “View” methods later, but as an example you can display HTML pages by first creating the view template (and placing it into “app/views”) folder and then calling “View::make” from your route response. Let's look at listing 3.22 to see how it is done:

3.22 Displaying HTML by responding with “View::make” method

```
// Contents of app/views/page.blade.php
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>HTML test</title>
</head>
<body>
    <b>This text is bold!</b>
    <p>It is inside of a template</p>
</body>
</html>

// Inside of app/routes.php
Route::get('html', function()
{
    // Display HTML from a template by calling Laravel View methods
    return View::make('page');
});
```

Now going to the ‘html’ route of your application will display a plain html page with the contents you specified in the view template.

These are the basic ways to show something back to the user, but what if you wanted to take the user to another page inside of your application, for example to the login page if the user is not logged in? Laravel provides you with a nice feature called “redirects” that can do exactly that.

3.5.2 Redirects

Redirects are convenient when you want to direct the user to another route within your application. This technique is often used to create a desired flow in an application. We have discussed route filters in section 3.3 of this chapter and you might have noticed redirects being used in some of the filters that Laravel is shipped with in the “app/filters.php” file. One of the filters we used was the “auth” filter and here is a reminder how it looked like (listing 3.23):

Listing 3.23 Auth filter is implementing a redirect

```
Route::filter('auth', function()
{
    // Redirecting the user to login page if not logged in
    if (Auth::guest()) return Redirect::guest('login');
});
```

To the user redirects simply make it look like they just went straight to the destination of the redirect. Laravel provides a few ways of using this simple yet efficient technique:

- Redirecting to another URL inside of your application
- Redirecting to a named route
- Redirecting to a function inside of a class (controller)

We will look at the first two ways of redirects here and the third one will be discussed in the “Controllers” chapter.

3.5.2.1 Redirecting to other URLs inside of the application

Redirecting to any URL inside of your application could be done by returning a “Redirect::to” method, for example in listing 3.24 we will redirect the user to the page with the URL of “tea” when they access the “coffee” URL:

Listing 3.24 Redirecting the user to another URL inside of the application

```
Route::get('coffee', function()
{
    // Redirecting the user to URL “tea” relative to your application
    return Redirect::to('tea');
});
```

Now if you try to access “coffee” URL you will immediately see your browser change the URL to “tea” which is what you would expect. You are not limited to just simple URLs, you can do more complex URLs just as successfully (listing 3.25):

Listing 3.25 Redirecting the user to a long URL inside of the application

```
Route::get('coffee', function()
{
    return Redirect::to('tea/black/earl-grey');
});
```

As long as you have your routing defined for the target URLs, the user will be redirected to them. Another way to redirect to URLs inside the application is to redirect to named routes.

3.5.2.2 Redirecting to named routes

Laravel allows you to name your routes for convenience. If you had named any of your routes, you could redirect to them using Laravel’s “Redirect::route” method providing the name of the route to redirect to. For example, the code in listing 3.26 upon going to “coffee” URL will redirect the user to a route named as ‘tea’ and will show a text string telling which route it is:

Listing 3.26 Redirecting the user to a named route

```
Route::get('coffee', function()
{
    // Redirecting to a named route
    return Redirect::route('tea');
});

// Naming the 'this-is-tea' route as 'tea'
Route::get('this-is-tea', array('as' => 'tea', function()
{
    return 'I am the tea route';
}));
```

This could be useful when you have many routes that don’t have expressive names and you would like to keep the structure of the routing cleaner.

3.6 Summary

The importance of this chapter cannot be underestimated. Routing defines the underlying behavior of your applications and without it Laravel applications simply would not function. You have learned how to create basic routes and how to get something out of them. You have explored the different types of routes and have applied the learned concepts by creating a simple HTML form that submits data to your Laravel application. Now you know how to pass parameters to the routes, how protect them using route filters and how to construct the desired behavior for your application.

In the next chapter you will learn in detail how to work with Laravel’s view templates and how to apply the knowledge you have already gained to make your pages more informative and management of them easier and enjoyable.

4. Views

This chapter covers

- Basics of views and templates
- Passing data to views
- Blade template engine
- View layouts
- HTML helpers

Most modern web applications have a user interface component to them. As the user goes to various pages of the application, he or she will see web pages that look different due to changes in content or changes in layout of the pages. For example a typical blog would have a page that lists all posts, pages that display an individual blog post and a page that shows an archive of all blog posts. A social network usually has some information feed, a profile page for each user, a page showing all private messages, etc. In modern web applications the user interfaces are becoming increasingly dynamic and complex. MVC frameworks like Laravel simplify creation of user interfaces by the use of “Views”, the V in MVC.

Laravel provides the developer with convenient separation between application’s data and the way the data is displayed in the browser, making the application’s presentation layer maintainable. In this chapter you will learn in detail about Laravel’s approach to templates and outputting data to the user. You will understand how to show the data from within the application and how to define the look of different parts of the application. You will meet Laravel’s powerful template engine called “Blade” and its methods of using conditional statements and loops. Then you will learn about Blade layouts and how they can help the developer eliminate repetition of code. You will also be introduced to Laravel’s shortcut methods of generating some of the HTML tags and form elements that will shorten the code that you need to produce to create good looking user interfaces.

4.1 Introducing views and templates

What are “views”? If you are not familiar with the concept of views, they are the presentation layer, the user interface of the web application. From small pages like a login page to blog layouts and even e-Commerce websites, views are used to build up the resulting page that the user will see in their browser when they click on a link or submit a form.

Consider an example of a Laravel-powered MVC application that displays a login page. The client’s browser sends a request to a “login” URL relative to application’s root. As this request is processed by the application running on the server, it is routed to a controller that might operate with application’s data (models) and load a template (view) containing HTML, Javascript, CSS, placeholders for data and/or other markup. This template is then compiled by Laravel filling the placeholders with actual data and returned back to the controller. The result of the application’s execution will be an HTML document that is then displayed in the user’s browser (figure 4.1).

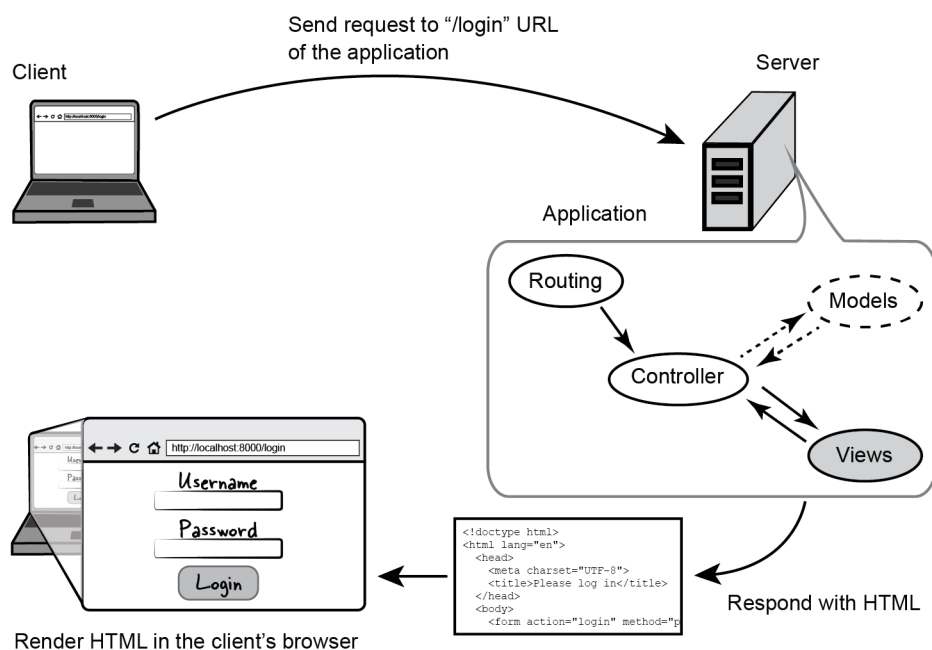


Figure 4.1 Flow of a client's request to a Laravel MVC application



Definition

Views define visual representation of a web application and enable the user to interact with the application. Typically a view is the HTML response returned by the application to the browser initiating the request.

Using views without controllers

At this point of the book you might not be familiar with controllers (they are covered in the next chapter). Views could still be used in your application even without the use of controllers. In this case the views will be loaded directly from the routing, compiled by Laravel and the resulting HTML is then given to the user as a response to a request.

Examples in this chapter will operate with views directly from the routing, not involving controllers.

Why use views? Keeping the presentation of the application separate from the application's logic (routing and controllers) makes it easy for the developer to create and maintain user interfaces that application displays to the user. Laravel uses views to give the developer flexibility and maintainability when it comes to building the presentation layer of the application.



Info

Methods that deal with views in Laravel are under “View” class. To understand which part the views play in a Laravel web application and how the concept of using views benefits you as a developer, let’s define the purpose of views.

4.1.1 Views - application’s response to a request

A view is the final step of client’s request to a web application. When the user is navigating to a URL in the application, there is an expectation that the application will respond by showing a requested page or by changing the content of the current page. One example of this expected behavior is when the user navigates to URL of a login page in your application and expects to see a login form in return. In section 3.1.3 of chapter 3 you have seen the route definition for this kind of flow, it is provided in listing 4.1 below. While this code is not done properly (returning HTML from application’s logic instead of using the concept of Views), the result of its execution would still be a login page that is presented to the user:

Listing 4.1 Route that displays the login form HTML without the use of view templates

```
Route::get('login', function()  
{  
    // The response of this route is an HTML document, a view  
    return '<form action="login" method="post">  
        Username: <input type="text" name="username"><br>  
        Password: <input type="password" name="password">  
        <input type="submit" value="Login">  
    </form>';  
});
```

The HTML of the login page that the user is presented with after navigating to the “login” URL relative to application’s root (figure 4.2) would be called “a view”:

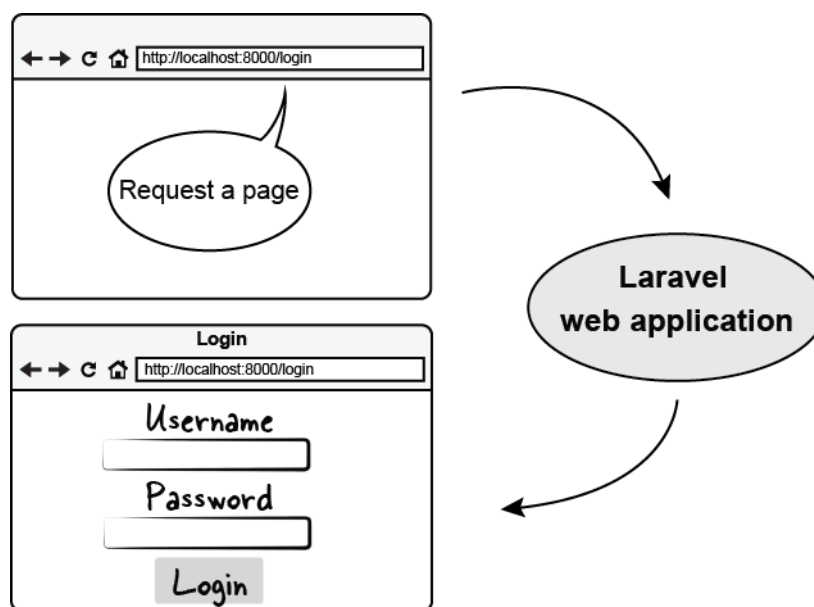


Figure 4.2 Login form view returned from the application as a response to a request

So, to show a page with a login form all that we did was return whatever HTML we wanted to see. Easy? Yes. But is it clean, maintainable and easy to work with? Probably not! Modern web pages have far more than two elements on a single page. Imagine if the page would have a navigation bar, a sidebar, a footer, and 15 form input fields instead of two? Add to that dynamic data that needs to be retrieved from a database and displayed on the page. The code that is needed to show all this output would quickly get messy and very difficult to modify.

Modern web application frameworks like Laravel solve this problem by the use of templates. Let's look at how Laravel uses the concept of templates to simplify creation and maintenance of views.

4.1.2 Templates in Laravel, Blade

In Laravel you can store HTML (or a template that would be compiled into HTML behind the scenes) outside of the application's logic, and just feed that template with the data that you retrieve from your application. Even better, you can create elements that are common to more than one page of the application – such as header, footer, sidebar and more – as separate templates that you could just reuse in any page to save amount of code that you need to duplicate.

Laravel uses its own template engine called “Blade”. This template engine is similar to other PHP template engines like Twig or Smarty but has its own syntax and shortcut methods. Using Blade makes it possible to combine many templates into a single HTML output (figure 4.3):

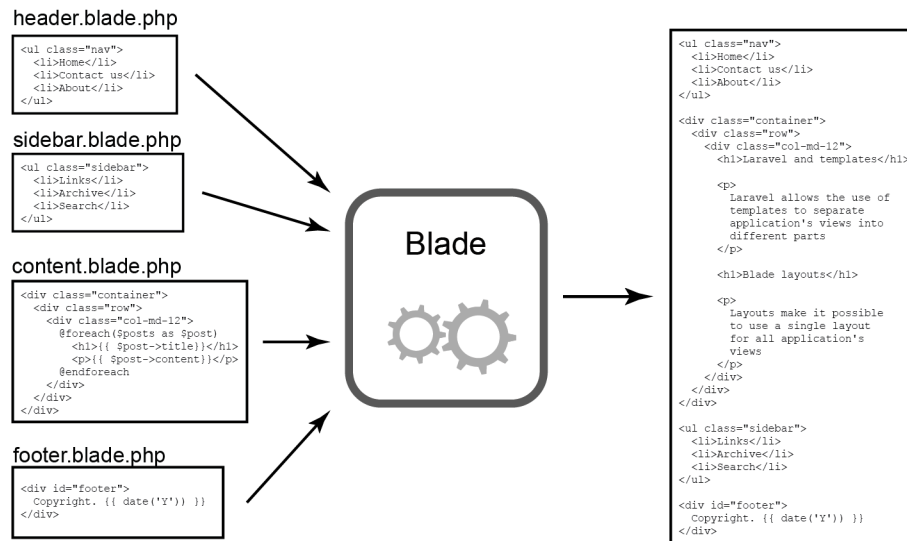


Figure 4.3 Blade template engine combines many templates into single HTML document

The days of mixing PHP code, database queries, HTML and application's logic are over. The concept of views in Laravel solves the problem of mixing everything together by providing methods that separate the logic of the application flow and the skeleton (template) for the results that the application should return to the user. Throughout this chapter you will learn about the View methods available in Laravel and about Blade template engine. First, let's take a look at how you can create a simple view template and show it to the user.

4.2 Creating and organizing views

Creating views in Laravel is simple. To see it in action, let's convert the code we created in section 4.1.1 to show the login form but this time using Laravel's views instead of mixing the HTML directly in the application's response.

By convention, in Laravel applications all views are stored in "app/views" folder as you might remember from the application structure overview in chapter 2. Let's create a brand new view template called "login.php" and place it there. As for the contents, we will just copy and paste the desired HTML output of the login route (listing 4.2):

Listing 4.2 Contents of the view file for the login page (app/views/login.php)

```

<form action="login" method="post">
  Username: <input type="text" name="username"> <br>
  Password: <input type="password" name="password">
  <input type="submit" value="Login">
</form>

```

Now that you have created the view template, you can use Laravel's methods to render the template from the application. The method that renders a view template stored in "app/views" folder is "View::make('file')" where "file" is the filename of the template that you'd like to show.



Note

View::make assumes that view files have extension ".php" or ".blade.php" so there is no need to specify file extension when passing a filename to it.

Let's modify the code of the login route to return the “login.php” view template (listing 4.3) to the user:

Listing 4.3 Route that displays the login form using a View method

```
Route::get('login', function()
{
    // Returning a view (login.blade.php in the app/views folder) will show
    // its HTML in the browser
    return View::make('login');
});
```

If you go to the “login” route now, you should see the same exact login page that we expected to see in section 4.1.1 (figure 4.4):

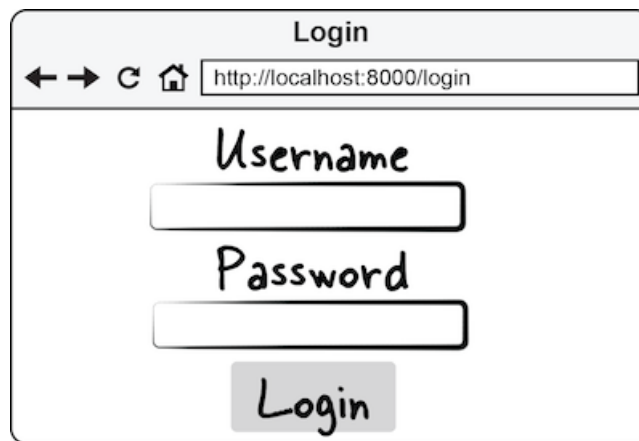


Figure 4.4 Login form displayed using View::make method

How is this different from the method we used to show the login form in listing 4.1? By using the “View::make” method we were able to separate the presentation of the login page from the logic of the application (in the case with the login form, we removed the HTML from the route definition). This allows for great maintainability of the page if you wanted to update it or change it completely without affecting the code of the application logic.

4.2.0.1 Separating view templates in folders

As the application grows, you will end up with many view templates. An average sized application could contain between 50-80 view templates or more when it is completed. Laravel does not prohibit you from storing the view templates inside of other folders in the “app/views” folder. For example if you wanted to store all view templates relating to user registration, login and logout functionality of your application, you are free to create another folder called “auth” and put the login.php and other view templates in there, giving the folder structure like in figure 4.5:

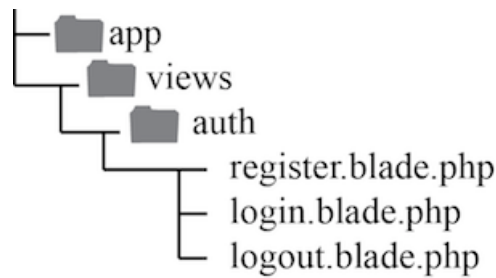


Figure 4.5 Example of separating view templates into a folder under “views” folder

In case of folder structure similar to figure 4.3, to specify that the view templates are located in “auth” folder, you would need to separate the folder name from the view template name by using a period (“.”), like in listing 4.4:

Listing 4.4 Using “.” to tell Laravel that view template is in a folder inside of app/views:

```
Route::get('login', function()
{
    // Display file login.blade.php from app/views/auth folder
    return View::make('auth.login');
});

Route::get('logout', function()
{
    // Display file logout.blade.php from app/views/auth folder
    return View::make('auth.logout');
});

Route::get('register', function()
{
    // Display file register.blade.php from app/views/auth folder
    return View::make('auth.register');
});
```

Keeping the view templates that have something in common together allows the developer to easily find the template that needs modification when the application code gets large.

While using view templates is very convenient and you can make the application respond with complete HTML documents, how would you go about outputting data from the application into the views? Laravel has you covered in that regard too. Let’s learn about passing data to the view templates.

4.3 Passing data to views

In most applications, not all pages are static. There is often a need to retrieve data from a database or other data source and embed it into the resulting HTML that will be presented to the user. Not so long ago, developers would embed SQL queries directly into the application code that would result in giant unmaintainable mess. For example a page that shows a table containing all order inquiries in a shop might look like listing 4.5:

Listing 4.5 Bad code mixing SQL queries, query parameters and HTML output

```

<?
$catId = (isset($_GET['catId']) && $_GET['catId'] > 0)? $_GET['catId'] : 0;
$sql = "SELECT * FROM tbl_orders, tbl_product, tbl_category
    WHERE tbl_orders.inquiry_pdid = tbl_product.pd_id AND tbl_product.cat_id = tbl_category\
y.cat_id
    AND tbl_category.cat_parent_id = $catId
    ORDER BY tbl_orders.inquiry_date DESC, tbl_category.cat_parent_id ASC";
$result = dbQuery($sql);
?>

<table width="100%" border="0" cellpadding="5" cellspacing="0" class="text">
    <tr id="listTableHeader">
        <td>Inquiry ID </td>
        <td>Batch ID</td>
        <td>Customer</td>
        <td>Brand</td>
        <td>Product</td>
        <td>Date</td>
    </tr>
    <?php if (dbNumRows($result) > 0) {
        while($row = dbFetchAssoc($result)) {
            extract($row);
        ?>
        <tr>
            <td><?php echo $inquiry_id; ?></td>
            <td><?php echo $inquiry_odid; ?></td>
            <td><?php echo $inquiry_batchall; ?></td>
            <td>
                <?php
                $sqlname = "SELECT cat_name FROM tbl_category WHERE cat_id = $cat_parent_id";
                $resultname = dbQuery($sqlname);
                $rowname = dbFetchAssoc($resultname);
                echo $rowname['cat_name'];
            ?>
            </td>
            <td><?php echo $cat_name; ?></td>
            <td><?php echo $pd_name; ?></td>
            <td><?php echo formatDate($order_date); ?></td>
        </tr><?php } } ?>
    </table>

```

Notice how the SQL queries, query parameters and HTML output are all inside of single PHP file making debugging and modification of this code practically impossible.

MVC frameworks like Laravel solve these problems by separating the application’s logic from the data output (visual representation). Laravel provides a few ways of passing the data retrieved inside of the application to the application’s views:

- By using the arguments of “View::make” method
- By using a method “with” appended to “View::make” method

Both of these methods lead to the same result – providing the view template with data that is passed from the application. Let’s look at these two methods in more detail.

4.3.1 Using “View::make” arguments to pass data

In section 4.1.2 you were introduced to “View::make” method that would render a view template stored in “app/views” folder. As you might remember, the first argument of “View::make” had to be the filename of the view template (without the “.blade.php” or “.php” extension). One of the methods of passing data to the view templates is to use the second argument of “View::make” method.

The second argument of “View::make” method accepts an array of data that you would like to use in the view template. The keys of the passed data array will become variables that you could then show in the view by using PHP’s echo function.

For example if you wanted to display current date and time in your login form while keeping the data separate from the view, you would first make that data available in the “View::make” method and then use PHP’s echo in the view template to display that data.



Note

Make sure you have set your time zone in app/config/app.php configuration file before executing the following code

Let’s see this in action in listing 4.6:

Listing 4.6 Using second argument of View::make to pass data to a view

```
// File app/routes.php
Route::get('login', function()
{
    // Create an array of data that will be used in the view template
    $data = array('currentDateTime' => date('Y-m-d H:i:s'));
    // Pass the data array as the second argument of View::make
    return View::make('login', $data);
});

// File app/views/login.php
Current date and time: <br>

<?php
    // Display the data in the view
    echo($currentDateTime);
?>
```

```
<form action="login" method="post">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password">
  <input type="submit" value="Login">
</form>
```

Execute this code by going to “login” route and you should see a page similar to figure 4.6:

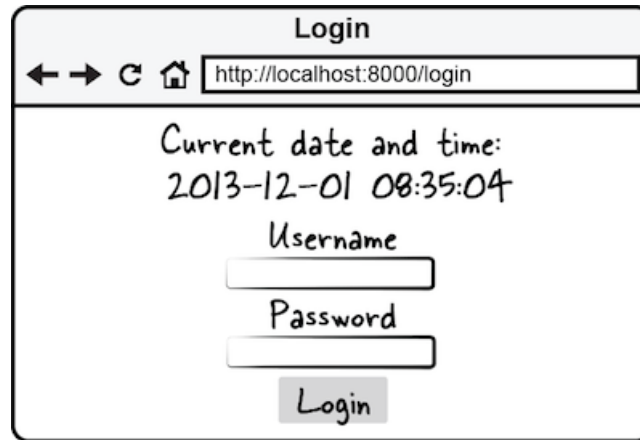


Figure 4.6 Result of passing date and time data to a view template

Using an array for data means you can pass as many variables or objects as you wish through it, as you will see in later parts of the book you could pass whole database objects from Laravel’s Eloquent ORM this way too.

Laravel has a shorthand method for passing data to the view templates that you can use to keep your code cleaner when you need to keep the name of the data variables explicit. Let’s look at using it!

4.3.2 Using method “with”

Instead of using the second argument of the “View::make” method you can append a method “with” to the end of the “View::make” method to pass any data to the view. Method “with” takes in two arguments, the name of the passed data and the data itself that will be assigned to that name in the view.



Tip

Method “with” is chainable, meaning you can put many methods one after another to pass as many variables as you want.

For example if you wanted to pass two variables to the view, let’s say the current date and current time and display them in the view, you would first assign the desired date and time values to those two variables. Then you would append method “with” to “View::make” assigning the names for the variables to the date data, and in the view template you would use the names of the variables you passed in to output them. Please see the code in listing 4.7 to see this in action:

Listing 4.7 Using method “with” to pass data to a view

```
// File app/routes.php
Route::get('login', function()
{
    // Store current date
    $date = date('Y-m-d');
    // Store current time
    $time = date('H:i:s');
    // Pass the date and time to the view
    return View::make('login')->with('date', $date)->with('time', $time);
});

// File app/views/login.php

// Display current date passed from the route
Current date: <?php echo($date); ?> <br>
// Display current time passed from the route
Current time <?php echo($time); ?>

<form action="login" method="post">
    Username: <input type="text" name="username"><br>
    Password: <input type="password" name="password">
    <input type="submit" value="Login">
</form>
```

The result of executing this code would be a screen similar to figure 4.7 telling the user current date and time on the login screen:

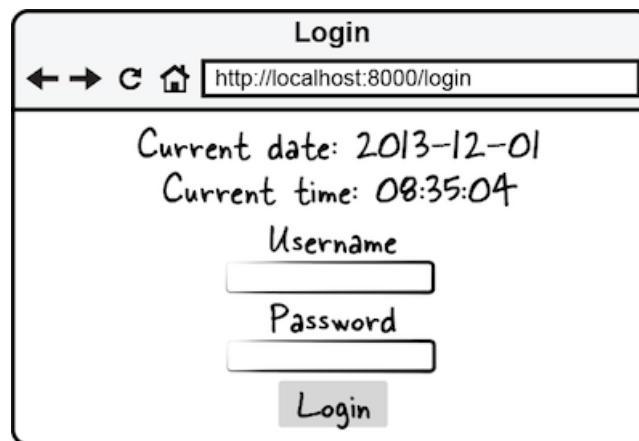


Figure 4.7 Result of passing date and time data to a view template by using method “with”

As you can see, using chainable “with” method makes it very convenient to bind data to the view when you need to keep the names of data variables easily visible in the code that passes the data to the view.

To Laravel there is no difference which method you will use to pass the data to your views, by passing data as a second argument to the “View::make” method or by using chainable method “with”. The result will

be the same – data that you can easily display in the view templates. It is up to you, the developer, which method you want to use in different scenarios.

While we have seen that using view templates greatly simplifies keeping the application code separate from the presentation, the provided view templates would still end up looking like PHP code when you have something more complex than outputting strings.

Imagine having PHP’s control structures like if/else, for/foreach loops and more in the view templates. Quickly enough, the templates would look messy and become hard to maintain, defying the purpose of views altogether. To alleviate those problems, Laravel comes with a simple to use, yet powerful template engine called “Blade” that you will meet in the next section!

4.4 Blade template engine

Blade is a template engine that comes with Laravel framework. Its purpose is to simplify outputting and looping over data, nesting of views, using conditional statements and more. In this section you will learn about Blade’s methods of working with view templates.

Internally, Blade converts files written with special “Blade” syntax into PHP files that contain the desired output. Some of the goals of Blade template engine are to make the view templates readable either by the back-end or front-end developer and to shorten the amount of code that needs to be written to display application’s data.



Note

View templates that use Blade have an extension of “.blade.php”

One of the ways in which Blade achieves the goals stated above is by attempting to minimize resemblance of the view templates to plain PHP code that was not designed for eloquence when it comes to presenting data. As you will see from the examples listed below in table 4.1, operations such as outputting data, looping through data, embedding conditional statements all could be easily done with Blade statements while keeping code a lot shorter and more readable for the developer.

Table 4.1 Comparison of using plain PHP and Blade template engine in the view templates

Plain PHP views (.php)	Blade views (.blade.php)
Outputting data in PHP <code><?php echo(date('Y')); ?></code>	Outputting data using Blade <code>{{ date('Y') }}</code>
Looping through data with PHP <code><?php foreach(\$users as \$user){ ?></code> <code><p></code> <code><?php echo(\$user->name); ?>
</code> <code><?php echo(\$user->address); ?></code> <code></p></code> <code><?php } ?></code>	Looping through data with Blade <code>@foreach(\$users as \$user)</code> <code><p></code> <code>{{ \$user->name }}</code> <code>
</code> <code>{{ \$user->address }}</code> <code></p></code> <code>@endforeach</code>
Using conditional statements in PHP <code><?php if(\$category == 'blog') { ?></code> <code>...</code> <code><?php } else { ?></code> <code>...</code> <code><?php } ?></code>	Using conditional statements in Blade <code>@if(\$category == 'blog')</code> <code>...</code> <code>@else</code> <code>...</code> <code>@endif</code>

Blade template engine provides the developer with many shortcuts that allow the developer to:

- Remove some of the inconsistencies of PHP
- Output and properly escape data
- Use conditional statements and loops
- Separate the view templates into layouts and sections
- Nest output throughout view templates
- Generate various HTML elements with minimum code

Let's gradually learn about all of these benefits of using Laravel's Blade template engine. By the end of this chapter you will be a Blade master, outputting data, looping through data and using layouts like a pro. First, let's start with some basic functions of Blade template engine.

4.4.1 Outputting and escaping data

Blade is very good at making PHP code look a whole lot better. By using few simple conventions and shortcuts, Blade is trying to be a more readable alternative to PHP when it comes to view templates while not completely replacing PHP from the equation.

Outputting a data string in Blade is achieved by putting a variable of type "string" inside of double curly braces "{{" and "}}". Imagine a simple example that would show "Hello, John" where "John" is a variable passed from a route. Listing 4.8 shows two ways of displaying the name variable, one using plain PHP and another one using Blade:

Listing 4.8 Displaying a string in a view using plain PHP and using Blade

```
// app/routes.php
Route::get('hello', function()
{
    $name = 'John';
    // Pass the name variable to the view
    return View::make('hello')->with('name', $name);
});

// app/views/hello.blade.php

// Display the name using PHP echo
Hello, <?php echo($name); ?>
<br>
// Display the name using Blade
Hello, {{ $name }}

// Same output achieved with both methods of displaying data:
Hello, John
```

This seems pretty easy, doesn't it? Need to display a variable in your views – put it in double curly braces. What will happen if you try to output a whole array using double curly braces? Laravel will promptly notify you that you need to have a variable that could be cast to type "string", Blade cannot echo an array just like it wouldn't work in plain PHP. You would need to loop over the array and display each element in the loop. We will look at using Blade's loops and conditional statements later in this chapter.

4.4.1.1 Escaping Data

Sometimes you need to make sure that the variables you display in the browser are HTML escaped for end-user security reasons. Imagine if your application deals with user-contributed data stored in the database and at some point you need to display that data to other users. If you display the data just like it is, without any escaping, there is a possibility that some users will try to enter malicious Javascript or HTML tags. To easily solve this problem, Laravel provides a way to escape data before it is displayed. You can escape and display the data by using triple braces “{{{” and “}}” instead of using double braces.



Note

Internally Laravel uses PHP’s `htmlspecialchars()` function to escape HTML characters when triple curly braces are used

Let’s see it in action in listing 4.9 where the HTML entities like quotes, less than (“<”) and greater than (“>”) characters are escaped into their corresponding HTML representations:

Listing 4.9 Escaping data using triple curly braces

```
// app/routes.php
Route::get('hack', function()
{ // Create malicious Javascript
    $data = "<script>alert('My hack');</script>";
    return View::make('hack')->with('data', $data);
});

// app/views/hack.blade.php

// Escape the output before it is displayed
The data: {{{ $data }}}

// Result (the HTML characters are properly escaped):
The data: &lt;script&gt;alert(&#039;My hack&#039;);&lt;/script&gt;
```

By escaping the data before it is displayed you will make sure that no malicious code can be displayed to the end-user of your application!

Now that you are able to output data with double curly braces and even escape it to protect the users of your application, let’s learn how you can control the flow of outputting the data and how you can use PHP loops in your view templates.

4.4.2 Using conditional statements and loops

Blade provides you with an easy way of using “if/else/elseif” conditional statements and PHP loops like “for”, “foreach” and “while”. Using conditional statements could be very useful when the application needs to hide or show something depending on data from the database or on the value of the user’s input. The loops could be used when the data to be displayed in the view template comes as an array of strings or as a database object.

You can use conditional statements in your view templates using plain PHP or using special Blade syntax. First let's take a look at using conditional statements with plain PHP and then we will transform the same code using convenient methods that Laravel's Blade provides for conditional statements.

4.4.2.1 Using conditional statements in the views with plain php

Imagine that your application needs to show a different message depending on what time of day it is. Since Blade views are still PHP friendly, you could use PHP conditional statements to control what the user will see. Let's look at the example in listing 4.10 where the application will show "Good morning, user!" if the time on the server where the application is executing is before noon and "Good afternoon, user!" if the time is after noon:

Listing 4.10 Showing a different message depending on server time (using plain PHP)

```
// app/routes.php
Route::get('greet', function()
{
    //Get the current time of day, 'am' or 'pm'
    $timeOfDay = date('a');
    return View::make('greet')->with('timeOfDay', $timeOfDay);
});

// app/views/greet.blade.php

//Use PHP's conditional statement to show a different message
<?php if ($timeOfDay == 'am') { ?>
    Good morning, user!
<?php } else { ?>
    Good afternoon, user!
<?php } ?>
```

When you access the URL "/greet" relative to your application's root, you will see a different greeting depending on time of day as shown in figure 4.8:

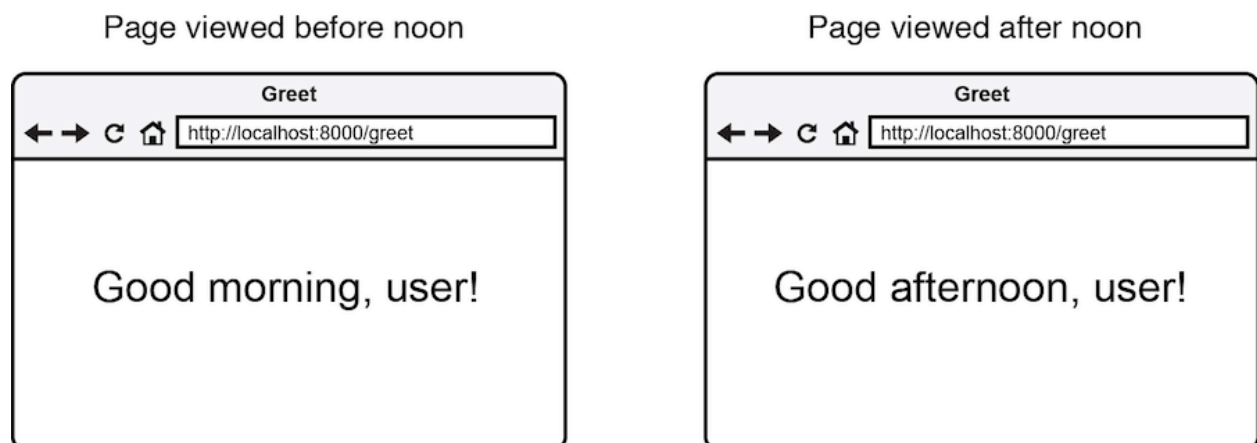


Figure 4.8 Showing different greeting message depending on time of day that the application is accessed

This is great, you can use PHP's conditional statements like "if", "else", "elseif" to control the flow of what the user will see, but isn't there a better way of doing this using Laravel's methods? Indeed there is a better way, by using special Blade helpers.

4.4.2.2 Using conditional statements in the views with blade

Blade has equivalents of PHP's conditional statements. Using Blade's syntax for conditional statements will cut the amount of code you need to write for the view templates and will make it a lot more readable.

Blade's conditional statements look almost like conditional statements in plain PHP with alternative syntax (listing 4.11). To use a conditional statement using Blade you would prepend the "if" with an "@" sign. Then, instead of using curly braces to separate the parts of the conditional statement, you would use Blade's special statements: @else, @elseif and @endif.



Note

There is no need to enclose Blade statements in PHP opening and closing tags, ""

Listing 4.11 Syntax of conditional statements in Blade

```
@if (expression)
    ...
@endif
@elseif (expression)
    ...
@else
    ...
@endif
```

To see the use of conditional statements in action, let's convert the code that we created above in listing 4.10 and use Blade syntax instead of plain PHP. The result on the screen will be the same but the code will be cleaner (listing 4.12):

Listing 4.12 Showing a different message depending on server time (using Blade)

```
// app/routes.php
Route::get('greet', function()
{
    // Get the current time of day, 'am' or 'pm'
    $timeOfDay = date('a');
    return View::make('greet')->with('timeOfDay', $timeOfDay);
});

// app/views/greet.blade.php
// Use Blade's conditional statement syntax to show a different message
@if ($timeOfDay == 'am')
    Good morning, user!
@else
    Good afternoon, user!
@endif
```

While you would get the same output as in figure 4.6, using Blade code made it easier for you as the developer to create the view template. Using conditional statements in Blade is simple but most importantly more elegant than writing plain PHP in the view templates. Besides conditional statements Blade makes it easier to use loops within the views. Let's learn about that next.

4.4.2.3 Using loops

Using loops in Blade is not much different from using conditional statements. The loops that are available in Blade are equivalent to PHP's "for", "foreach" and "while" loops and they have similar syntax to PHP's alternative syntax for loops (listing 4.13):

Listing 4.13 Syntax of loops in Blade

```
// "for" loop:
// Same as PHP's "for (expression1, expression2, expression3) { ... }"
@for (expression1, expression2, expression3)
    ...
@endfor

// simple "foreach" loop:
// Same as PHP's "foreach (array_or_object as $value) { ... }"
@foreach (array_or_object as $value)
    ...
@endforeach

// "foreach" loop that also assigns keys to a variable:
// Same as PHP's "foreach (array_or_object as $key => $value) { ... }"
@foreach (array_or_object as $key => $value)
    ...
@endforeach

// "while" loop
// Same as PHP's "while (expression) { ... }"
@while (expression)
    ...
@endwhile
```

You can use loops in the view templates to display attributes of an object or elements of an array. Why not take a look at an example? Let's imagine that you have an array of users in the application and you would like to display the list of users in the browser upon going to a route "users". We can use Blade's "foreach" statement to iterate over the users and we can use Blade's shortcut for "echo" by using double curly braces (listing 4.14):

Listing 4.14 Displaying a list of users using Blade's foreach loop

```
// app/routes.php
Route::get('users', function()
{
    // Create an array of users
    $users = array(
        'User 1',
        'User 2',
        'User 3',
        'User 4'
    );
    // Pass the array of users to the view "users"
    return View::make('users')->with('users', $users);
});

// app/views/users.blade.php
// Iterate over the array of users passed into the view using the foreach loop
@foreach($users as $user)
    // Output the element of the "users" array into the view
    <p>{{ $user }}</p>
    // Close the foreach loop
@endforeach
```

When you access the route “users”, the code above will display the list of users like in figure 4.9:

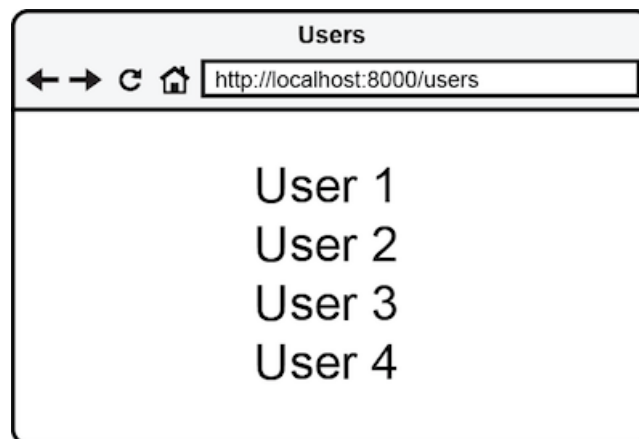


Figure 4.9 Result of using “foreach” loop to display items stored in an array

As you can see, it doesn’t take a lot to use loops when Blade has these built-in capabilities that are taken from PHP but look a bit different than PHP. Using loops in the view templates is something that almost every application incorporates to show data in table rows, lists, grids and more. Having methods of looping through data benefits the developer tremendously.

You are now able to use conditional statements and loops in the view templates which are very common throughout applications of any size. There are just a few more things left to learn about Blade template engine. Let’s move onto next topics like using layouts and sections!

4.5 Blade layouts

Blade layouts help the developer write even less code for view templates. Imagine that your application has 3 pages in total and every page has different content but the same header and footer throughout all pages (figure 4.10):

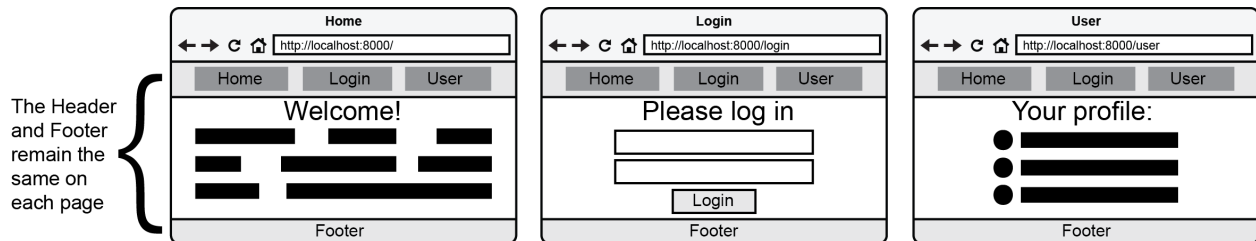


Figure 4.10 Elements that are common to all pages of the site could be combined into a layout

When you want to update either header or footer, you would need to go through all of the view templates and edit them one by one. This is doable (though not recommended) if your application is very small, but when your application grows to let's say 50-80 different view templates, one small update to a part that is common to all pages could take you a whole day to do! Laravel's Blade comes with a way to solve this problem by use of layouts.



Definition

Layouts are Blade views with HTML/CSS/JS elements that are common to many pages of the application and placeholders defined for the elements that change depending on the content

The concept of using layouts isn't new to web development frameworks. Laravel simply embraces a well-tested and helpful technique of making the code for application's views reusable and much more readable. Laravel has a few special keywords that you need to know when using View Layouts. These keywords, their placement in the views and their function is described in table 4.2 below:

Table 4.2 A list of Blade keywords and functions relevant to layouts

Blade keyword	Placement	Description
@yield()	in a layout	Insert a section of content that has a name specified via the argument
@extends()	in a view	Apply another Blade template specified via the argument as a layout for current Blade template
@section()	in a view	Define a section of content that will be inserted into the layout that the current Blade template uses. Name of the section is specified as the argument to @section().
...		
@stop		
@include()	in a view or in a layout	Insert any other Blade template specified as an argument.

Over next few pages you will learn how to create a layout that you will use throughout the rest of this chapter. This simple layout will contain a page title, header and footer that all view templates of the application that are using this layout will inherit. Let's start with creating an HTML page that will serve as a layout.

4.5.1 Creating and using a layout

The process of using layouts in a Laravel application consists of creating a layout template – a Blade view file that has special areas marked as placeholders for sections of content and then telling the views that need to be injected into those section areas to “extend” or apply that specific layout. If you’d like to have different templates for different parts of your application (for example administration panel and user-facing areas) Laravel allows you to define as many layouts as you want.

Creating a Blade layout is the same as creating any other Blade view. Like any Blade view, the layout is a file that ends with extension “.blade.php” and goes into the “app/views” folder. As a Blade view, a layout can use all the advantages of Blade including conditional statements and loops and more. Often a layout acts as a skeleton for application’s HTML representation and such things as page title, header and footer, references to CSS and JS files would be a part of a layout.



Note

Layouts should be stored as “.blade.php” files anywhere in “app/views” folder

Our example layout will be saved as “layout.blade.php” in the “app/views” folder. First it will contain just HTML and an output of current year in the footer, then it will be made more dynamic by the use of Blade “@yield” keyword. Let’s start creating this layout by starting out with some HTML from Listing 4.15:

Listing 4.15 Creating an HTML template for a layout (app/views/layout.blade.php)

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>My Laravel application</title>
    // If there is any CSS that all pages of the application need to use, put it here
    ...
  </head>
  <body>
    <header> Header </header>
    // This will be later replaced with a placeholder for application's content
    Application content will go here
    // Using Blade to output current year in the footer using PHP's "date" function
    <footer> Copyright, {{ date('Y') }} </footer>
    // If there is any JS that all pages of the application need to use, put it here
    ...
  </body>
</html>
```

This layout has enough HTML structure to be valid HTML5 and is only missing an actual placeholder for content. We will look at using placeholders a bit later but for now let’s take a look at how we can use the layout in the application’s views.

4.5.1.1 Using layouts from views

To use a layout in a view, first you need to specify the filename of the Blade layout using a special Blade function “@extends” that will go at the top of a view file. “@extends” accepts a single argument – a name of the Blade file that acts as a layout and has placeholder areas defined.

For example, if you had a Blade view “home.blade.php” in “app/views” and you wanted that view to use a layout that was defined in listing 4.15 above, you would specify the name of the layout (‘layout’, without “.blade.php” extension) in the “@extends” function as in listing 4.16:

Listing 4.16 Using Blade layout from a view (app/views/home.blade.php)

```
// Specify the filename of the layout (without extension)
@extends('layout')

// The rest of the code of the view template
...
```

Now, when the “home.blade.php” view is rendered using “View::make” method, the HTML from the “layout.blade.php” will be present in the result returned from the application.



Note

Any Blade view can use a Blade layout

What’s special about Blade layouts is the way that the placeholders are defined for the sections of content that will be injected into the layout. Let’s meet the first method of defining placeholders for sections of content: method “@yield”.

4.5.1.2 Using method “@yield” to specify placeholders

Method “@yield” is used in a Blade layout to specify that some content from a view using the layout will be injected into the layout when the page is output in the browser. In other words, method “@yield” specifies a placeholder for a section of content. You can define a placeholder for a specific area of the Blade template by using @yield(‘nameOfSection’), where “nameOfSection” is a name that you give to an area of content.

Let’s look at this in action. We will modify the layout provided in listing 4.15 and define an area for content, let’s call it “content”. Later, this area of the layout will correspond to an area inside of a Blade view that will provide an actual content for that placeholder. Listing 4.17 shows the lines of the layout that we will need to change in order to create a content area placeholder:

Listing 4.17 Specifying a placeholder for content (app/views/layout.blade.php)

```
...
<header> Header </header>

// Specifying an area of layout as a placeholder for content
@yield('content')

<footer> Copyright, {{ date('Y') }} </footer>
```

To pass the content from a Blade view to a Blade layout, the view needs to be using the layout and it needs to have the content areas specified as “sections”

4.5.2 Using sections

Blade layouts consist of a template that has “sections” specified to hold the content that will be provided by the views. Defining an area of a Blade view as a section of content consists of placing two special Blade constructs – markers that will mark the start and the end of a section. To mark the start of the section you need to place a function `@section('nameOfSection')` where ‘nameOfSection’ is the name corresponding to `@yield()` statement in the layout.



Note

The names of the sections need to correspond to the names of the placeholders in the Blade layout.

To mark an end of the section you need to place “`@stop`” letting Blade know that only the content between `@section ... @stop` will be the content that you want to inject in the template.

Let’s take a look at using sections in action. We will continue working on the view “home.blade.php” that will provide the content that we would like to inject into the Blade layout. Let’s specify the area of the view that will hold a section of content (listing 4.18):

Listing 4.18 Specifying area of a section inside a view (app/views/home.blade.php)

```
// Telling the view to use a layout ('app/views/layout.blade.php')
@extends('layout')

// Opening content section
@section('content')
    // The content that will be injected in the layout replacing "@yield('content')"
```

This is the home page

```
// Closing content section
@stop
```

When this view will be rendered using “`View::make`” method from a route, the content inside of the `@section('content')` from the view will replace the `@yield('content')` in the layout that the view is using. Here’s the complete listing that shows a route definition, the layout and the view template that is using the layout (listing 4.19):

Listing 4.19 Complete listing of using a layout in a view

```
// app/routes.php
Route::get('/', function()
{
    return View::make('home');
});

// app/views/layout.blade.php
<!doctype html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>My Laravel application</title>
```

```
</head>
<body>
  <header> Header </header>
  @yield('content')
  <footer> Copyright, {{ date('Y') }} </footer>
</body>
</html>

// app/views/home.blade.php
@extends('layout')

@section('content')
  This is the home page
@stop
```

Code in listing 4.19 will display the following page when you visit application's index page:

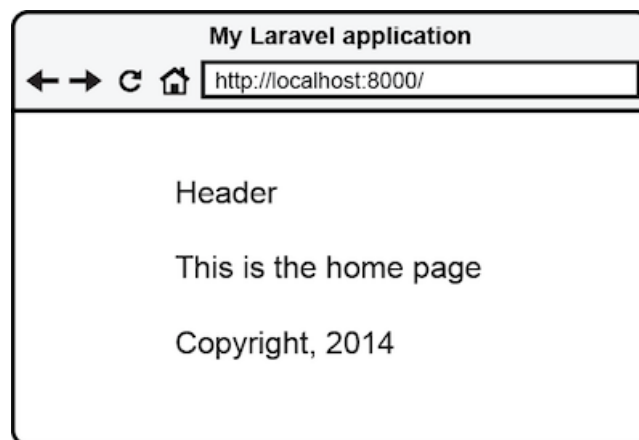


Figure 4.11 Result of using a layout in a view

Something interesting happened when the view was presented to the user. Some parts of the page like the header and footer come from the “layout.blade.php” layout and the content saying “This is the home page” come from the “home.blade.php” view. Laravel seamlessly combined these parts from two different files just like we wanted it to do, allowing us to use a layout template for displaying view’s content.

You are not limited to having a single section for content inside of the view (and corresponding placeholders in the layout). There could be as many content sections as you wish. For example if the layout had placeholders for “content”, “moreContent” and “evenMoreContent” while the view template had sections with corresponding names, the matching placeholders would be filled from the view template (listing 4.20):

Listing 4.20 Example of using more than one section for content

```
// app/views/layout.blade.php
...
<header> Header </header>
@yield('content')
@yield('moreContent')
@yield('evenMoreContent')
<footer> Copyright, {{ date('Y') }} </footer>
...

// app/views/home.blade.php
@extends('layout')

@section('content')
    <p>This is just Content</p>
@stop

@section('moreContent')
    <p>This is More Content</p>
@stop

@section('evenMoreContent')
    <p>This is Even More Content</p>
@stop

// Resulting HTML that will be displayed
<!doctype html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>My Laravel application</title>
    </head>
    <body>
        <header> Header </header>

        <p>This is just Content</p>
        <p>This is More Content</p>
        <p>This is Even More Content</p>

        <footer> Copyright, 2014 </footer>
    </body>
</html>
```

Ability to have unlimited number of sections for a layout gives you incredible flexibility in defining dynamic areas of a template underlying the views. Anything that you might need to include in the layout

template – additional scripts, stylesheets, content and more could be passed through the use of `@section` and `@yield`.

Laravel provides even more tools in your Blade arsenal. One very useful feature of Blade is nesting views inside of other views. Let's learn about it in the next section!

4.5.3 Nesting views by using `@include`

You can easily re-use complete Blade views inside of other views to keep your Blade views cleaner and prevent code duplication. Imagine that two or more pages of the application use a chunk of HTML that is absolutely the same for these pages but not the same throughout all the pages (therefore not a good candidate to be included in a layout). To make it easy to reuse the HTML throughout the pages you could inject that HTML from a Blade view by using a special Blade method “`@include`”.

Blade's `@include` works in a similar way to PHP's `include`. It includes and evaluates a Blade view file inside of another Blade view file. To use it in a view or layout, provide a name of a Blade view that you would like to include as a parameter to `@include`. For example if you had a piece of navigation that you wanted to use in some views, you would first create the navigation HTML and save it as a Blade view and then `@include` it in another view. Let's create a simple navigation HTML and store it in “`navigation.blade.php`” file inside of “`app/views`” folder. Then in any Blade files that we wanted to display this navigation HTML we would do `@include('navigation')` that would include and render the “`navigation.blade.php`” file. Listing 4.21 demonstrates using `@include` in action:

Listing 4.21 Example of nesting views by using `@include`

```
// app/routes.php
Route::get('blog', function()
{
    return View::make('blog');
});

// app/views/blog.blade.php
...
// Nesting another Blade view
@include('navigation')

My Laravel blog
...

// app/views/navigation.blade.php
<nav>
    <ul>
        <li><a href="/home">Home</a></li>
        <li><a href="/blog">Blog</a></li>
    </ul>
</nav>

// Resulting HTML
<nav>
```

```
<ul>
  <li><a href="/home">Home</a></li>
  <li><a href="/blog">Blog</a></li>
</ul>
</nav>
My Laravel blog
```

Using `@include` can greatly reduce the amount of HTML that you need to write for your view templates. In combination with view layouts it provides you with very powerful tools of managing the presentation layer of the application. There is a lot more tools available to you when using Blade template engine and Appendix B summarizes Blade methods and shortcuts available.

4.6 Summary

Deep knowledge of using Blade views and layouts in Laravel could save you hundreds upon hundreds hours of work and help keep the codebase of your application clean and maintainable. The shortcuts that Blade provides when outputting or escaping data are essential to applications of any size. Whether it is controlling what the user will see by using conditional statements or iterating through attributes of an object by using loops, Laravel's Blade template engine has you covered.

Using layouts and nesting methods removes the code that otherwise would be unnecessarily repeated and gives you a way to create beautiful web applications without messy code. Now that you have mastered the presentation layer of Laravel you can go onto next chapter where you will learn about using application's responses to get the desired format of application's response to a request.

5. Understanding Controllers

This chapter covers

- Default controllers
- Creating new controllers
- Controller routing
- RESTful and Resource Controllers

When the code of the application grows, it becomes harder for the developer to plan new features and to maintain existing features. Up to this point in the book, all functional code of Laravel applications was put inside of “app/routes.php” file.

To help with separating application’s features into smaller, manageable pieces Laravel allows the use of PHP classes called “controllers”. The concept of using controllers is present in many modern frameworks. In applications built with Laravel controllers play an important part of Model-View-Controller application architecture. Working together with Laravel’s routing mechanisms, controllers make maintenance, testability and expansion of the application a lot easier.

In this chapter you will learn how controllers can help you in separating the code of the application and why they could be used when the application expands. You will take a look at two controllers that come with Laravel out of the box: BaseController and HomeController. Then you will learn about creation of your own Basic, RESTful and Resource controllers while using convenient shortcuts that they provide for managing application’s execution. After that, you will become a pro at combining Laravel’s routing with controllers. Finally, you will learn how to use routing conventions to pass parameters to the methods of the three types of controllers.

5.1 Introducing controllers, the “C” in MVC

Using controllers becomes necessary when your application has more functionality than just showing a couple web pages. As the code of the application grows, putting all application’s features in the routing file is not a wise option. Through the use of controllers – special PHP classes - developer can separate the functionality of the application into different classes and connect the functions inside of these classes to Laravel’s routing.

Routing VS. Controllers?

When developers come to MVC frameworks like Laravel they get a notion that their application should only use routes or only controllers. The concept of using routes and controllers could confuse some newcomers.

While it is true that for very small applications it is possible to get by without even touching controllers, a better practice is to put the application’s functionality into controllers as the application’s features add up.

The title of this sidebar is misleading on purpose. When you build a web application with Laravel, it is the harmony of routes and controllers working together that could make the application extendable, testable and maintainable. Using controllers is necessary for any growing application. With the flexibility that Laravel

provides for routing to controllers' methods you can start using controllers right away.

Let's compare two scenarios of application structure, one that uses just routing file (routes.php) to store the application logic and the other one – using controllers to do the same. You will gradually understand the benefits of using controllers as you read through the examples.



Definition

Application logic is a set of executable actions that the application follows to validate input, retrieve or store data.

Imagine that you have an application that shows a login page to the user. As you have learned in chapter 3, when a client's browser makes a request to a URL in your Laravel application, the routing mechanism will match the requested URL to the routes defined in app/routes.php file. If the requested URL matches one of the defined routes, let's say a "login" route, the function passed as the second argument of that route will be executed displaying a webpage in the client's browser, as illustrated in figure 5.1 below:

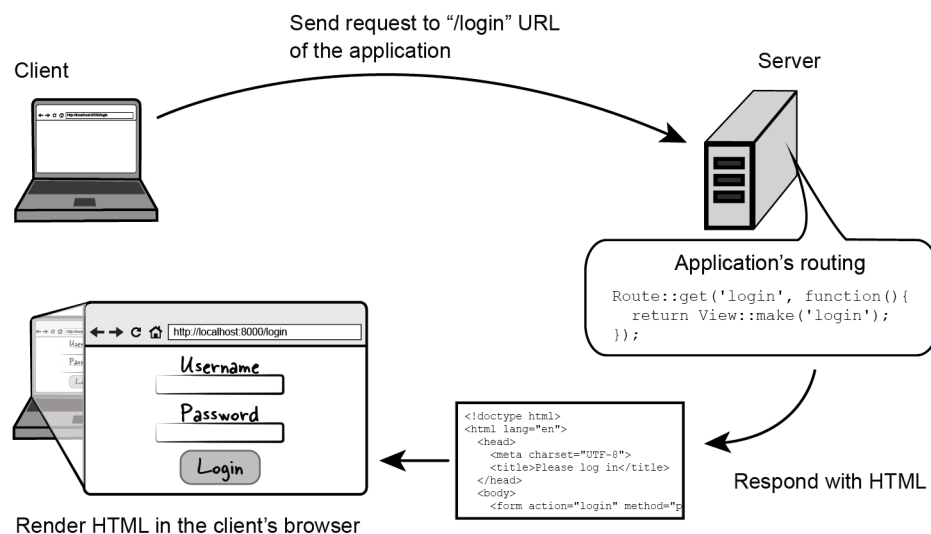


Figure 5.1 Flow of a client's request to a Laravel application that only uses routing

As an alternative to routes-only application Laravel provides you with a way to tell the routing to execute a member function - also called an "action" - of an existing PHP class instead of executing a function passed as an argument to a routing method. The actions could contain any application logic such as working with database or file system, redirecting the user to other pages within the application or simply showing an HTML page. The PHP classes that contain application logic and whose methods or "actions" are mapped to the application's routing definitions are called "controllers".



Definition

Controllers are PHP classes that contain application's logic and are called from the application's routing. Controllers contain methods called "actions" that direct the execution flow of a web application by working with application's data (models), producing output (views) or redirecting.

By creating a separate PHP class (controller) and specifying which routes in the routing file use its member functions you are able to take full advantage of MVC pattern in your Laravel applications. How does the application flow change in this case comparing to having application logic in the routing file? Let's take a look below.

When a request comes in to the application, the routing mechanism will match the requested URL and execute specified controller action. That controller action could contain logic for working with data (model) and displaying the result of execution by presenting an HTML document (view). In case with displaying a login page the request lifecycle using a controller to direct the flow of the application will look like the figure 5.2:

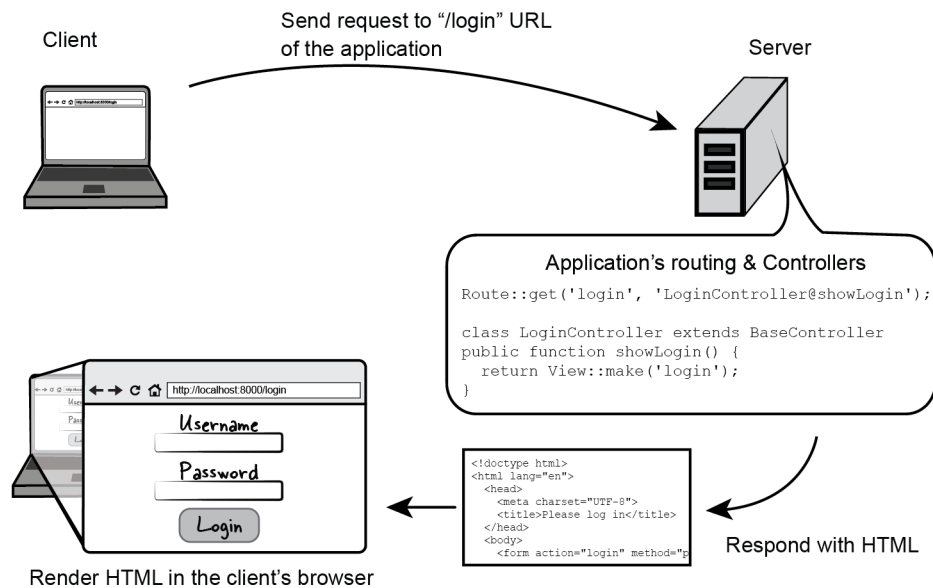


Figure 5.2 Flow of a client's request to a Laravel application that uses routing in conjunction with controllers

Another important advantage of using controllers with routes instead of putting all code into the "route.php" file is that using separate PHP classes to build application's functionality brings the full benefits of Object Oriented Programming (OOP) such as dependency injection, encapsulation, inheritance and more. This aspect of application architecture is incredibly important for developers who practice good design patterns in their applications.

Now that we have looked at the advantages of using controllers and have seen how they could integrate in the Laravel applications, let's take a look at how Laravel applies this concept in action by providing two default controllers out of the box.

5.2 Default Controllers: BaseController and HomeController

By convention, all controllers in a Laravel application are located in "app/controllers" folder. A fresh Laravel installation comes with two default controllers that are not in use until they are connected to the routing mechanisms:

- BaseController
- HomeController

When used together, these two controllers provide you with a small example of controller-based application architecture. Out of the box HomeController includes simple functionality to show a page to the user, while BaseController acts as a foundation for all application's controllers. We will build our controllers on top of the default BaseController in this book but it's worth knowing that you can create your own controller that serves as a parent to all controllers. Let's look at the two default controllers and understand their purpose in a Laravel application.



Note

The controllers need to be connected to the routing mechanisms in order to be executed when the application runs. We will take a look at controller routing later in this chapter.

5.2.1 Base controller (app/controllers/BaseController.php)

BaseController acts as a base of the rest of your application controllers. Its job is to provide a common place to store all logic that other controllers extending the BaseController will be using. Out of the box, BaseController extends Laravel's Controller class and comes with a single function that could be used to setup a view layout for all methods inside a controller. You can see the contents of the BaseController.php from a fresh Laravel installation in listing 5.1:

Listing 5.1 Contents of BaseController.php from a fresh Laravel installation

```
<?php

// BaseController is a child of Laravel's "Controller" class
class BaseController extends Controller {
    // setupLayout() function could be used to set up a view layout that will be used in all \
controllers unless it is overwritten in the individual controllers
    protected function setupLayout()
    {
        if ( ! is_null($this->layout))
        {
            $this->layout = View::make($this->layout);
        }
    }
}
```



Note

When building applications that use controllers, BaseController acts as a parent controller and all other controllers should be extending BaseController.

A good example of a BaseController's child controller is HomeController that also comes with a fresh Laravel installation. Let's take a look at this controller and its purpose next.

5.2.2 Home controller (app/controllers/HomeController.php)

HomeController is a small controller that extends BaseController and demonstrates the use of controller-based architecture in a new application. By default, this controller has one and only method called “showWelcome” that, when connected to the routing (which we will explore in a later section), is responsible for showing the “hello” page that tells that Laravel is functioning properly (figure 5.3):



Figure 5.3 The result of executing showWelcome function of HomeController

The content of the HomeController.php that shows the page in figure 5.3 is provided in listing 5.2 below:

Listing 5.2 Contents of HomeController.php in a fresh Laravel installation

```
<?php

// HomeController is a child of BaseController
class HomeController extends BaseController {
    // showWelcome function displays a “hello” page when it’s called from the routing
    public function showWelcome()
    {
        return View::make('hello');
    }
}
```

While HomeController is a good starting point containing only one function for a demonstration of showing a simple page, it can easily be modified to contain more functions responsible for application’s homepage actions. But what if you wanted to go beyond the default controllers that come with Laravel out of the box? Let’s look at creating our own controllers in the next section.

5.3 Creating controllers

Creating a new controller in Laravel applications is as simple as creating a new PHP class that extends BaseController and placing it in the “app/controllers” folder. Let’s learn more about creation of new controllers while restructuring some of the code you have encountered previously in this book.

In the beginning of this chapter you have seen a diagram of an application flow that would display a login page when “login” URL is requested. You can see that diagram again in figure 5.4 below:

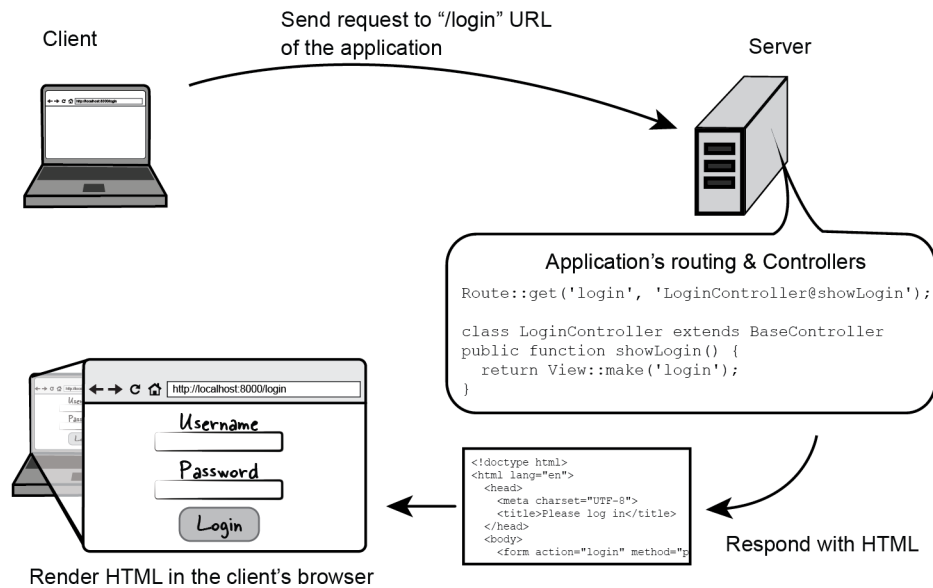


Figure 5.4 A high level view of a browser request to application's 'login' route that uses a controller to show the login page

This application flow in the figure above uses a combination of routing and controller to process the request and display a view (HTML page). Because controllers are meant to contain related logic, the same controller could also be used to not only show the login page but also to process the form submission.

Over the next few pages you will implement a login page flow by creating a new controller responsible for displaying a login page and processing it upon submission of the login form.

5.3.1 Creating a new controller: LoginController

As it was mentioned above any new controllers should be extending the BaseController class by convention. To create a new controller you would need to create a new PHP class and place it in the "app/controllers" folder. Then all operations that the controller will execute should be placed in this new class' methods.



Note

New controllers should extend BaseController class and should be placed in "app/controllers" folder

We will create a controller that will be responsible for the logic related to the user login, so it would be helpful to have the name of the controller descriptive of its responsibility, for example "LoginController". The new file called "LoginController.php" will be put in "app/controllers" folder, containing the class definition as is shown in listing 5.3:

Listing 5.3 Definition of LoginController (app/controllers/LoginController.php)

```
<?php

// The LoginController is a child of BaseController class
class LoginController extends BaseController {

}
```

5.3.2 Creating logic for the LoginController

Now that the new controller is created, you can place the logic that the controller will execute when it will be called from the routing. The functionality of the controller that the routing can call directly needs to be placed in controller's class methods that are declared public. For example if you wanted to display a login page to the user, you would create a function that returns a view using Laravel's "View::make" methods (listing 5.4):

Listing 5.4 Showing a login form from the LoginController

```
<?php

class LoginController extends BaseController {
    // Method that will display the login page to the user
    public function showLogin()
    {
        return View::make('login');
    }
}
```

For now the LoginController would only show the login form to the user. We will add the functionality to process this form a bit later. Now let's define the routing for the single "showLogin" action of the LoginController.

5.3.3 Defining routing for LoginController

Having this simple controller functionality defined, we can connect it to application's routing by specifying that the "showLogin" function of the "LoginController" class will be called when the user goes to the "login" URL of your application in the browser. Laravel comes with an easy way to specify which method of a controller will be called when a particular route gets called. Instead of passing a closure function to the routing methods like you have seen before, you can provide a string containing the name of controller and the name of the method that should be executed when the route gets called. This way of routing is called "explicit routing".



Definition

Explicit routing is a way of routing to controllers that requires each controller method to be defined in the routing file and correspond to a URL in the application

We can see the concept of explicit routing in action by making the route definition for the login route in “app/routes.php” as follows (listing 5.5):

Listing 5.5 Connecting the “login” route to “showLogin” method of “LoginController”

```
// Specifying that the showLogin function of LoginController will be executed  
// when the user executes a GET request to the “login” route  
Route::get('login', 'LoginController@showLogin');
```

With the LoginController defined in “app/controllers” and its showLogin method specified as an argument to Route::get('login', ...), what exactly happens when the user requests the login page? Step by step, the application will go through the following process (illustrated in figure 5.5) ending up in showing the login page to the user:

- Laravel’s router looks for a match between the browser’s request (GET request to the “login” URL relative to the application) and a route definition.
- Because browser’s request type and URL correspond to the “login” route definition, Laravel will try to execute its second argument looking for a class “LoginController” and a “showLogin” method of that class
- The “showLogin” method responds by serving “login.blade.php” view template stored in “app/views”

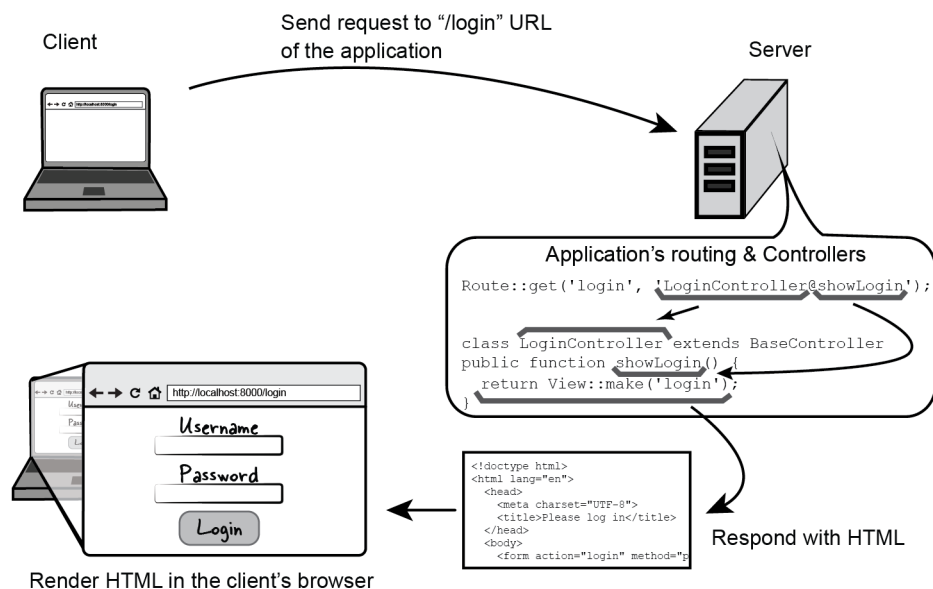


Figure 5.5 Diagram of the execution flow for the “login” route of the application. “showLogin” method of the “LoginController” responds to a GET request to “login” URL by serving a login page to the browser

This simple application now responds to “login” route by executing a method from “LoginController”, nice! As you can see from the diagram above, using controllers and controller routing keeps the functionality of the application grouped into separate classes instead of polluting the routing file. What if you wanted to add more functionality related to user login to this application?

To expand the functionality related to login you could add that functionality to LoginController and specify which route should execute it. For example to process submission of the form on the login page

creating a method called “processLogin” in the “LoginController” and placing the logic that would check user’s credentials there would be a great idea. Then, this new controller method could be connected to the routing by specifying that “processLogin” should be executed when there is a POST request to the ‘login’ route. The addition of the “processLogin” method to the “LoginController” and adding it to the route definition enables the application to show the login form and respond to user’s submission so that the flow of this small application would look like figure 5.6:

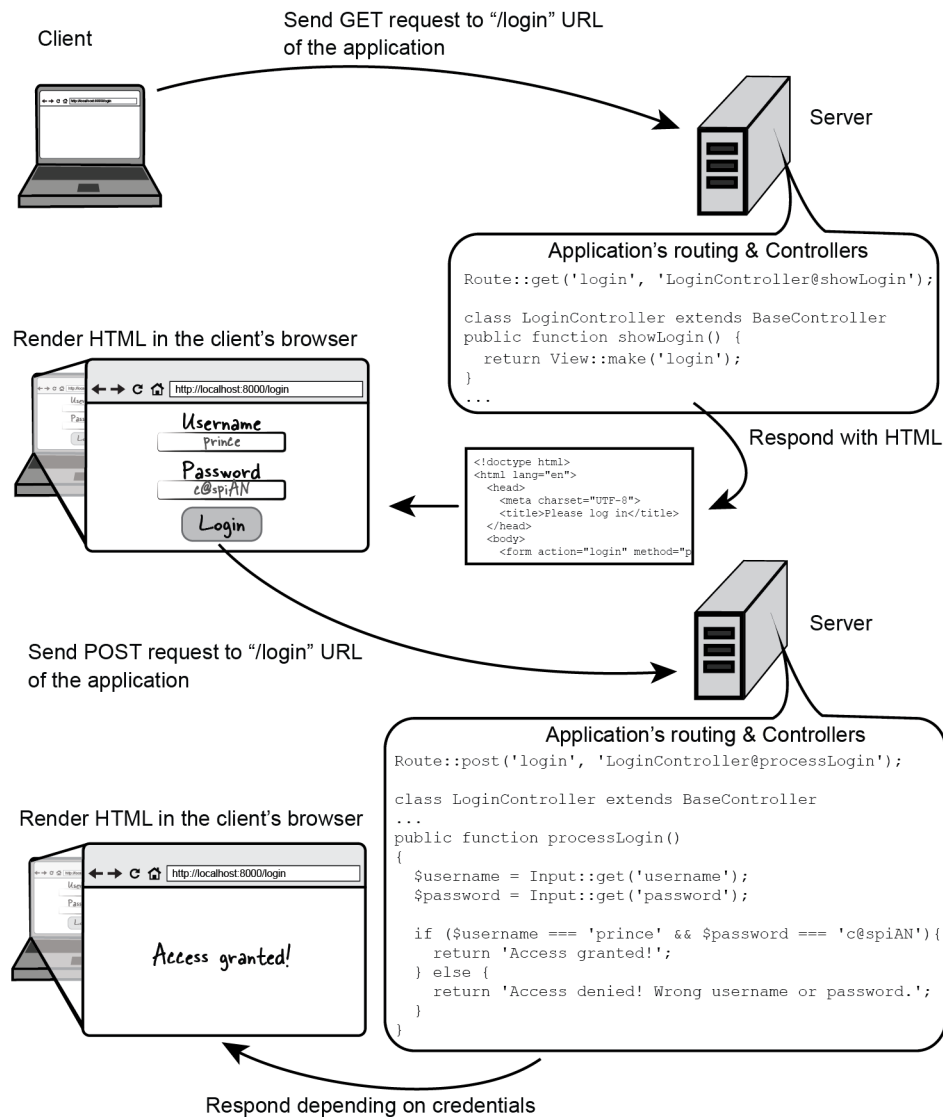


Figure 5.6 Execution flow for GET and POST requests to the route “login” that uses a controller to contain application’s functionality

The complete code listing that combines showing and processing the login form is in listing 5.6 below:

Listing 5.6 Using LoginController to show and process a login form

```
// The definition of routes that execute methods of LoginController (app/routes.php)
Route::get('login', 'LoginController@showLogin');
Route::post('login', 'LoginController@processLogin');

// The LoginController (app/controllers/LoginController.php)
class LoginController extends BaseController {

    // Display the login form
    public function showLogin()
    {
        return View::make('login');
    }

    // Process submission of the login form by verifying user's credentials
    public function processLogin()
    {
        $username = Input::get('username');
        $password = Input::get('password');
        if ($username === 'prince' && $password === 'c@spiAN') {
            return 'Access granted!';
        } else {
            return 'Access denied! Wrong username or password.';
        }
    }
}

// Contents of the login form Blade template (app/views/login.blade.php)
<form action="login" method="post">
    Username:
    <input type="text" name="username"><br>
    Password:
    <input type="password" name="password">
    <input type="submit" value="Login">
</form>
```

A controller like LoginController demonstrates extendable application structure that benefits the developer when the application's codebase is growing. Two methods or twenty, using controllers is a great way to group code related in its functionality together.

When the application grows to a point where there are many controllers and each controller has a few different methods, explicit controller routing would not be always effective because the routing file would become very large. To get around this potential problem and to accommodate developer preferences Laravel provides a few other methods of combining controllers and routing together. These methods allow for using some of the best practices in modern web development while reducing the amount of code that needs to be

created. Let's learn about the three types of controllers available in Laravel, their differences and how they could benefit your application.

5.4 Using Basic, RESTful and Resource controllers

There are three types of controller structure available in Laravel, each having a particular purpose depending on what functionality the controllers will contain. The need for these different types of controllers comes from the web development patterns that emerged over the years. The types of controllers that Laravel allows the developer to use are:

- Basic Controllers
- RESTful Controllers
- Resource Controllers

While the purpose of controllers remains the same - to control the execution flow of the application - there are certain cases when using one type of controller could be more appropriate than using another.

Basic controllers are great for simple functionality that might not be strongly related. RESTful controllers are very good when there is a need for making an API within the application. Resource controllers are a good fit for those cases when you want to have a consistent blueprint of managing a particular kind of data (resource). The answer to the question "which type of controller to use?" depends on the type of functionality that the controller will be executing and on the developer's architectural decisions for the application.

A short note on REST and RESTful controllers

In the context of Laravel controllers "REST" (abbreviation for "REpresentational State Transfer") and "RESTful controller" is an architectural pattern that requires the name of controller methods to describe what HTTP method ("get", "post", etc.) and what URL the method will be responsible for. Using RESTful controllers can shorten the amount of route definitions in the routing file significantly but on the other hand it could make the routing too implicit and harder to maintain.

The biggest structural difference between these three types of controllers is in the naming of controller's methods and in addition, ways of routing to them. While Basic Controllers should use explicit routing, RESTful and Resource Controllers could use Laravel's automatic routing (also called "implicit" routing) requiring less route definitions. Let's take a look at the purpose of each of these types of controllers and the differences in their method names in table 5.1:

Basic Controllers	RESTful controllers	Resource controllers
Purpose		
Simple controllers with flexible method names	Controllers suitable for RESTful API design	Controllers for RESTful management of resources
Routing to controller's methods		
Explicit (every route needs to be defined)	Implicit (automatic)	Implicit (automatic)
Controller method naming conventions		
Could be anything, e.g.: <ul style="list-style-type: none"> ▪ "showWelcome" ▪ "processContactForm" ▪ "logout" ▪ "displayUsers" 	Has to start with one of HTTP methods supported in Laravel: <ul style="list-style-type: none"> ▪ get (e.g. "getIndex") ▪ post (e.g. "postLogin") ▪ put ▪ patch ▪ delete 	Has to be one of the following: <ul style="list-style-type: none"> ▪ index ▪ create ▪ store ▪ show ▪ edit ▪ update ▪ destroy

Table 5.1 Difference between Basic, RESTful and Resource controllers

These three types of controllers are enough for vast majority of modern web applications. While at first glance you could say that Basic controller is all you need, method naming conventions for RESTful and Resource controllers allow you to use special routing shortcuts that you will learn a bit later in this chapter.



Note

You are not limited to using only one type of controller in your application. It is perfectly fine to mix controllers of these three different types in a single application

In the next three sections you will get familiar with creating controllers of these different types and understand their differences in practice. Let's start with the Basic controllers.

5.4.1 Creating and using Basic Controllers

You can create a Basic controller by making a new child class of "BaseController" and placing it in "app/controllers" folder. Basic controllers can have as many methods as necessary and these methods can have any name.



Tip

It is highly recommended to append the word “Controller” to the name of the controller classes to avoid naming conflicts, e.g. “UserController” and not “User”

Without knowing it, you created a Basic controller in section 5.3. As you might remember, the LoginController had two methods named “showLogin” and “processLogin” (listing 5.7).

Listing 5.7 Definition of LoginController (app/controller/LoginController.php)

```
class LoginController extends BaseController {  
  
    public function showLogin()  
    {  
        ...  
    }  
  
    public function processLogin()  
    {  
        ...  
    }  
}
```

The method naming of LoginController’s methods corresponds to the Basic controller naming convention from table 5.1. As you will learn later in this chapter, the routing for Basic controllers needs to be explicit. Every method of a Basic controller needs to be specified for the router to know which URL is controlled by it.

5.4.2 Creating and using RESTful Controllers

The only big difference between RESTful and Basic controllers is the method naming. Despite that difference, creating a RESTful controller follows the same process as creating a Basic controller:

1. Create a child class of BaseController in “app/controllers” folder
2. Define controller methods (actions)

RESTful controllers solve the problem of creating a route definition for every single method by having a simple convention for names of controller methods. This convention is as follows:



Method names in a RESTful controller should start with one of the HTTP verbs supported in Laravel: “get”, “post”, “put”, “patch” or “delete”, etc. The second part of the method name should start with a capital letter, e.g. getUsers, postCheckout

As an example that will help you understand the concept of RESTful controllers in action, imagine that you are developing an area of the site that allows a user to manage blog posts. The user should be able to do the following:

- View a list of all blog posts
- View an individual blog post
- Create a new blog post
- Edit a blog post
- Delete a specific blog post

A controller that would be responsible for this kind of post management could be called “PostsController”. At high level, this controller would contain the functionality for displaying and managing blog posts. The logic of this controller would be placed in the following methods (listing 5.8):

Listing 5.8 Example of a RESTful controller definition

```
class PostsController extends BaseController
{
    // View all posts
    public function getIndex(){ ... }
    // Show an individual post
    public function getView($id){ ... }
    // Show the form for the creation of a new post
    public function getNew(){ ... }
    // Process submission of a new post
    public function postNew(){ ... }
    // Show a form editing a post
    public function getEdit($id){ ... }
    // Process submission of edited post
    public function postEdit($id){ ... }
    // Delete a specific post
    public function deletePost($id){ ... }
}
```

Following the RESTful method naming convention allows you to use a special routing shortcut that automatically binds the controller’s actions to the routes that match the HTTP method and the URL of the request. Putting the route definition like in listing 5.9 would automatically enable the URL “posts” relative to the application URL to execute the “getIndex” method, URL “posts/new” to execute “getNew” method and so on.

Listing 5.9 Routing to a RESTful controller (app/routes.php)

```
// Enable automatic (implicit) routing to all methods inside of
// PostsController controller
Route::controller('posts', 'PostsController');
```



Note

RESTful controller routing will be explored in more detail in a later section of this chapter

That is only one line of code to route all controller’s methods! If the “PostsController” was a Basic controller, you would have to create seven route definitions, one for each method of the controller. As you

can see from this example, using a RESTful controller can save a lot of lines of code in the routing file. Laravel is flexible about controllers, in fact so flexible that there is one more type of controllers that could be useful in certain situations. Let's talk about "Resource Controllers".

5.4.3 Creating and using Resource Controllers

Resource Controllers are named that way because they deal with "resources" - types of data that the application works with. For example if your application manages videos, these videos could be considered as a type of a resource. Or if an application manages addresses, they could also be viewed as a resource. Controllers that manage resources are called "Resource Controllers".



Definition

Resource Controllers provide a consistent way of controlling creation, reading, updating and deleting for a resource.

Resource Controllers follow a convention that is somewhat different from RESTful controllers yet it allows for creation of RESTful application structure. With Resource Controllers each kind of operation on a resource such as creation, editing, etc. has its own designated controller method that is implicitly routed by using a simple route definition. Resource controllers can handle up to seven different actions on resource items:

- index (to show a list of the items of the resource)
- create (to show a form for creation of a new item of the resource)
- store (to save a new item of the resource)
- show (to show a specific item of the resource)
- edit (to show a form for editing a specific item of the resource)
- update (to process updating of a specific item of the resource)
- delete (to delete a specific item of the resource)

These seven actions are all methods that could be present in a Resource Controller. Creating a definition for a controller containing all these methods could take a bit of typing so Laravel comes with a tool to help you generate a blueprint for such a controller.

5.4.3.1 Creating resource controller with artisan command

Artisan, Laravel's command line helper has a built-in command that allows you to generate a template for a Resource Controller. Run the following command in the terminal to create a new Resource Controller:

```
php artisan controller:make SomeController
```

Where "SomeController" is the name of the class that you wish to be created, e.g. "VideosController". Running this command will create a new class in "app/controllers" folder with seven default methods already defined (listing 5.10):

Listing 5.10 Resource Controller created by Artisan command

```
// The class name comes from the name specified in the Artisan's controller:make command
class SomeController extends \BaseController {
    // List the items of the resource
    public function index(){}
    // Show a form for creation of a new item of the resource
    public function create(){}
        // Save a new item of the resource
    public function store(){}
        // Show a specific item of the resource
    public function show($id){}
        // Show a form for editing an item of the resource
    public function edit($id){}
        // Process updating of an item of the resource
    public function update($id){}
        // Delete a specific item of the resource
    public function destroy($id){}
}
```

While you can create a Resource Controller by hand, using the “controller:make” Artisan command accelerates the process by creating a skeleton for your controller. The seven actions of a Resource Controller need to contain some logic in order to be functional and what logic you put in to work with the application’s data is up to you. One of the main benefits of using Resource Controllers is that they help you create API-friendly controllers that work with application’s data.

**Note**

Using Resource Controllers makes it easy to integrate frontend Javascript frameworks like Backbone.js, Ember, AngularJS with Laravel applications

When the controller is generated and the logic for managing data is in place, you can easily tell the application’s routing to use the Resource Controller for a specific URL prefix by using the “Route::resource” route definition:

Listing 5.11 Example route definition for a Resource Controller

```
Route::resource('videos', 'VideosController');
```

This is all that is needed to route a Resource Controller to its many methods! Just like the RESTful controllers and their automatic routing using this routing shortcut will save you a lot of typing.

**Note**

Resource Controller routing will be explored in more detail in the next section of this chapter

Great! You have learned about the three different types of controllers (Basic, RESTful and Resource controllers). You now know how to create them and in what regard they are different. When developing

applications with Laravel controllers will be something you will use a lot so being a master in this domain will certainly help. While you have been introduced to controller routing on the surface, Laravel's powerful routing techniques provide you with multiple ways to link controllers and their methods to URLs in your application. Let's look at that in great detail in the next section.

5.5 Using controllers with routes

As mentioned before, controllers' methods will not be executed until they are called from application's routing methods. In order to tell the application which controller to use when a certain URL is requested, the mapping between the route and the controller need to be specified in the "app/routes.php" routing file. You have learned in section 5.2 that there are three types of controllers that you may use in your application: Basic, RESTful and Resource. Laravel provides three ways to connect controllers' member functions to the application's routes depending on the controller type:

- By pointing each method of a Basic controller to each individual route (explicit routing)
- By using "RESTful" controller architecture and its automatic route mapping (implicit routing)
- By using "Resource" controller architecture and its automatic route mapping (also implicit routing)

Over the next few pages we will learn about controller routing in greater detail. Besides routing to controller methods we will also review the concepts of passing parameters and using "filters" and how those important features of Laravel could be applied to controller routing. Let's start with routing to Basic Controllers.

5.5.1 Routing to Basic Controllers

Basic Controllers require the routing to be explicitly defined for each controller method. Only the methods that are defined that way would be executed when the browser sends a request to the application. You have seen it in the previous sections of this chapter but let's reiterate on how the Basic controller's methods could be connected to application's routing.

For example imagine you were making a controller responsible for showing and processing a contact page on your site, let's call it "ContactController". This controller would have at least two methods: one that would show the contact form (let's call it "showForm") and another one that would do the processing of contact form submission (let's call it "processForm"). The controller definition would look like this (listing 5.12):

Listing 5.12 Definition of ContactController (Basic Controller)

```
class ContactController extends BaseController {

    public function showForm()
    {
        // Display the contact form using a Blade template stored
        // in 'app/views/contact.blade.php'
        return View::make('contact');
    }

    public function processForm()
    {
```

```

    // Gather user's input
    $email      = Input::get('email');
    $name       = Input::get('name');
    $message    = Input::get('message');
    // Do input processing (send to email, store in DB, etc)
    ...
    // Show a page with a thank you message stored in 'app/views/thanks.blade.php'
    return View::make('thanks');
}
}

```

To route these two methods to URLs in the application, two route definitions need to be created, one for each method. These routes will make the methods executable when the user requests the contact page by going to URL “contact” relative to the application’s URL and also when he or she submits the form at the same URL (listing 5.13):

Listing 5.13 Routing to methods of a Basic Controller (app/routes.php)

```

// Routing to method showForm of ContactController
Route::get('contact', 'ContactController@showForm');
// Routing to method processForm of ContactController
Route::post('contact', 'ContactController@processForm');

```

To build a route definition for a single method of a Basic Controller you need to specify the type of HTTP request, destination of the request and a controller method of an existing class, separated by “@” sign. That method will then be executed when the incoming request matches the URL and HTTP method defined for that route. Diagram in figure 5.7 breaks down the components of a route definition to a method of Basic Controller:

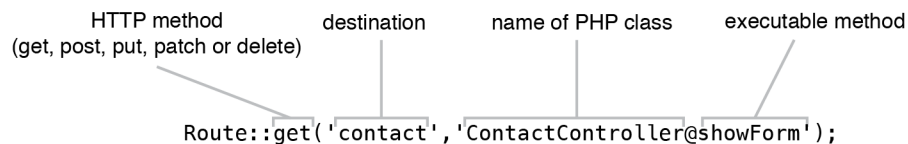


Figure 5.7 Route definition that links a single action of a controller to a specified destination



Note

You can use explicit routing to route to any of the three types of controllers, not just Basic Controllers

One advantage of using explicit routing is that when a developer looks at the routing file he or she can see exactly which methods will get executed depending on the type and URL of the request. As you see, routing to Basic controllers is not much different from passing a closure function as an argument to routing mechanisms like we have done in previous chapters of this book. In this case though, instead of writing out the whole function you can just point the routing definition to a method of an existing PHP class, controller.

5.5.2 Routing to RESTful Controllers

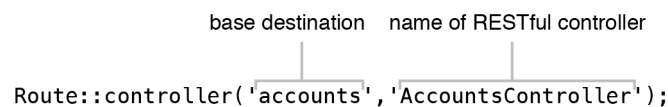
Routing to RESTful controllers is extremely simple as far as route definitions go. If you have a RESTful controller, for example called “AccountController”, you only need to put a single line to automatically assign all of its methods to a specified base destination.



Definition

In the context of routing, base destination (or base URL) is a URL relative to the application’s root URL. It acts as a point of reference for routing of RESTful and Resource Controllers

Figure 5.8 shows a route definition that will route all requests to URLs that start with “accounts/” to the methods of the RESTful “AccountsController” controller:



```
Route::controller('accounts', 'AccountsController');
```

Figure 5.8 Automatic routing to all methods of RESTful controller

While defining the route to a RESTful controller is easy, understanding how the automatic resolution of its methods works could be a little challenging and maybe even confusing at first sight. When a controller is routed by the use of “Route::controller”, a method of that controller will be executed only if these two conditions are satisfied:

- The type of the incoming request (get, post, put, patch or delete) matches the first part of controller’s method’s name
- The destination of the incoming request matches both, base destination and a lowercase representation of the second part of method’s name separated by a “/” sign

There is one exception to this rule – controller’s index route. Requests to the base destination will execute methods whose names end with “Index”.

Let’s explore an example of RESTful controller routing to see when the controller’s methods get executed. Imagine that “AccountsController” has the following three methods defined (listing 5.14):

Listing 5.14 Example RESTful controller (app/controllers/AccountsController.php)

```
class AccountsController extends BaseController
{
    public function getIndex(){ ... }
    public function getShow($id){ ... }
    public function postEdit(){ ... }
}
```

The “getIndex” method will be called when the URL “accounts” gets requested by the browser because it is controller’s index route. The “getShow” method will be executed when the user goes to the URL “accounts/show” because that method’s name consists of the “get” type of request and the destination “show”

which is derived from lowercase of “Show” and is appended to the base destination “accounts/”. Figure 5.9 shows how a GET request to the URL “accounts/show” would be routed to the “getShow” method of RESTful controller “AccountsController”:

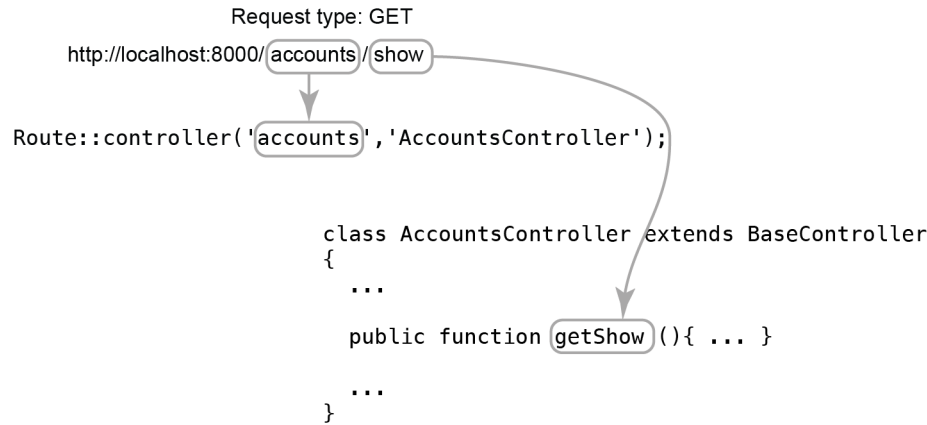


Figure 5.9 RESTful routing of “accounts/show” URL to “getShow” method of “AccountsController”

If the user makes a POST request to “accounts/edit” URL, the “postEdit” method will be executed because its name matches the “post” type of request to the URL “edit” that is appended to the “accounts/” base destination. Diagram in Figure 5.10 below shows the three cases of browser requests when those three methods will get called by the automatic routing and respond to the requests:

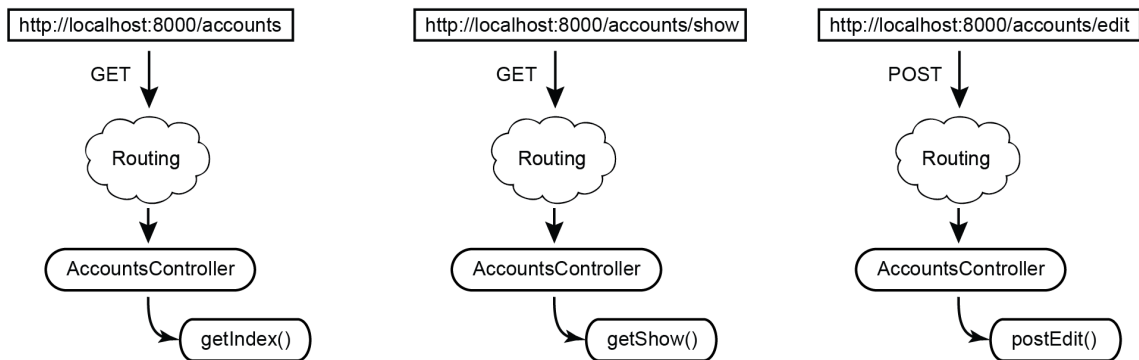


Figure 5.10 Browser requests that execute the methods of “AccountsController” that is automatically routed via “Route::controller” definition

As you can see, in comparison to the Basic Controllers, routing to the executable methods of a RESTful controller is tightly dependent on the names of the controller’s methods. Because of that it is very important to name the methods properly when you use RESTful controllers.



Note

If the name of RESTful controller’s method has two capitalized words following the type of the request then URL to that method will be separated through a dash, e.g. a URL to “getUserProfile” method would end with “user-profile”

Now that we have learned the concept of routing to RESTful Controllers in Laravel let's dive deeper into the world of controller routing. Let's explore how routing to Resource Controllers works.

5.5.3 Routing to Resource Controllers

Routing to Resource Controllers is very similar to how the RESTful Controllers are routed to because they share a similar concept – implicit (automatic) routing to all methods of the controller. The only big difference of Resource Controllers is that they can have a maximum of seven methods that deal with management of a resource:

- index
- create
- store
- show(\$id)
- edit(\$id)
- update(\$id)
- delete(\$id)

To route a particular base URL to a controller that has these methods defined only one route definition is needed. For example if you have a Resource Controller called “ImagesController”, only this line would need to be present in “app/routes.php” file to automatically route the base URL “images” to all of this controller's methods (figure 5.11):

```
Route::resource('images', 'ImagesController');
```

The diagram shows the code `Route::resource('images', 'ImagesController');`. Above the string `'images'`, the text "base destination" is written with a vertical line pointing down to the string. Above the string `'ImagesController'`, the text "name of Resource controller" is written with a vertical line pointing down to the string.

Figure 5.11 Automatic routing to all methods of a Resource Controller

The way routing to a Resource Controller works is that each of the seven methods have an implied combination of type of HTTP request and a path that is added to the base destination. For example the controller method with the name “store” will be executed when a request of type POST is sent to the base destination URL of the resource. Method “show” will be executed when a request of type GET is sent to a URL that combines the base destination URL and an ID of the resource that will become method's parameter. The “Route::resource” route definition will automatically route the methods of a Resource Controller in a way that is defined by a convention which you can see in table 5.2 below:

Request Type	Path	Controller Method	Route Name	Purpose
GET	/resource	index	resource.index	Show a list of the items in resource
POST	/resource	store	resource.store	Store a new item in resource
GET	/resource/create	create	resource.create	Show a form to create new item in resource
GET	/resource/{id}	show(\$id)	resource.show	Show an item of resource
GET	/resource/{id}/edit	edit(\$id)	resource.edit	Show a form to edit an item of resource
PUT or PATCH	/resource/{id}	update(\$id)	resource.update	Edit an item of resource
DELETE	/resource/{id}	destroy(\$id)	resource.destroy	Delete an item of resource

Table 5.2 Routing conventions for the Resource Controller routing

As you can see from the table 5.2, four methods of a Resource Controller require a parameter “id” to be passed in to the methods: “show”, “edit”, “update” and “destroy”. In case with routing to Resource Controllers the parameter “id” gets passed in through the URL of the request, for example the URL “images/2/edit” would execute the method “edit” of the “ImagesController” with the \$id parameter equal to 2. Passing parameters to controllers is something that we will explore in the next few pages.

5.6 Passing route parameters to controllers

When using either type of controllers, you might need to make the routing to controller’s methods a bit more dynamic in order to work with specific data. Telling a controller’s method an ID of an item or a specially formatted name of a blog post are cases when routing needs to be more flexible than just responding to base destinations like “images” or “posts”. As you might remember from the chapter on Routing (chapter 3), Laravel provides a way to specify what section of a route should be dynamic by the means of route parameters. The code in listing 5.15 demonstrates the use of a route parameter in a route definition that executes a closure function:

Listing 5.15 Passing route parameter to a closure function (no controller in use)

```
// Anything that is in place of {kind} in the URL becomes available as $kind
// parameter to the anonymous (closure) function
Route::get('coffee/{kind}', function($kind)
{
    // Visiting URL 'coffee/espresso' would output "Requested kind of coffee drink: espresso"
    return 'Requested kind of coffee drink: '.$kind;
});
```

Route parameters allow your application to be very dynamic and thankfully passing a variable to a controller's method is something that is very easy with Laravel's powerful routing mechanisms. In fact, you won't need to change any route definitions in order to pass parameters to RESTful and Resource controllers' methods. Over the next few pages we will explore how the route parameters may be passed to each controller type available in Laravel.

5.6.1 Passing parameters to methods of a Basic Controller

As you know, in Basic Controllers each method name has to have its own corresponding route definition. What is great is that to pass a parameter to Basic Controller's methods you only need to specify where in the URL you are expecting a dynamic parameter and Laravel will take care of passing it to controller's method.

Imagine an example of an application that responds to requests for coffee drinks by simply telling the user what kind of coffee drink they requested, just like the small route definition in the listing 5.15 above did. Let's create a Basic Controller that would process a request like that and let's name this controller "CoffeeController". It could have a few different methods, but one that we are especially interested in is method "showDrink" that would take in a parameter passed through the URL and would display to the user what kind of drink they requested. The code in listing 5.16 shows a definition of a "CoffeeController" and its "showDrink" method:

Listing 5.16 Definition of CoffeeController (app/controllers/CoffeeController.php)

```
class CoffeeController extends BaseController {
    ...

    public function showDrink($kind)
    {
        return 'Requested kind of coffee drink: '.$kind;
    }

    ...
}
```

Now, in order to let the application's routing know about a parameter that is passed to a URL responsible for executing "showDrink" method all you need to specify is where in the URL the parameter could occur. Listing 5.17 shows a route definition that makes the "showDrink" method of the "CoffeeController" react to the "coffee/{kind}" URL dynamically:

Listing 5.17 Passing a route parameter to a method of Basic Controller

```
Route::get('coffee/{kind}', 'CoffeeController@showDrink');
```

Having a route definition like this enables a specific method of a Basic Controller to use a variable that is passed to the method as an argument. What if you wanted to pass more than one parameter? That would not be a problem either. Just define the placeholders for the method parameters and Laravel will make them available for use in the method execution (listing 5.18):

Listing 5.18 Passing multiple route parameters to a method of Basic Controller

```
// app/routes.php
Route::get('coffee/{kind}/{size}', 'CoffeeController@showDrink');

// app/controllers/CoffeeController.php

class CoffeeController extends BaseController {

    public function showDrink($kind, $size)
    {
        return 'Kind of coffee drink: '.$kind.', size: '.$size;
    }
}
```

If the user went to a URL “coffee/latte/grande” relative to the application root URL, he or she would see a response like in figure 5.12:

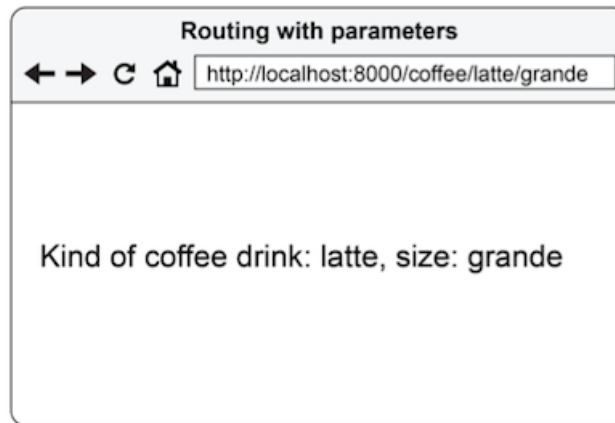


Figure 5.12 Result of method execution that has two parameters defined

This looks very similar to the way it was done using routing without controllers, doesn't it? Laravel promotes consistency in application architecture and that shows in the similarities between routing with parameters for a Basic Controller and passing parameters to closure functions in the route definitions. By using this technique you can pass as many parameters as you wish to as many methods as your Basic Controllers have.

5.6.2 Passing parameters to methods of a RESTful Controller

The methods of a RESTful controller are automatically routed by “Route::controller” definition therefore there is no way to specify which method of the RESTful controller should accept parameters and which method should not (listing 5.19).

Listing 5.19 Routing to a RESTful controller

```
Route::controller('accounts', 'AccountsController');
```

Laravel takes a unique approach to solve this problem by letting the developer pass parameters to controller’s methods without specifying them in the routing definition.

By using implicit routing of “Route::controller” definition all methods of the RESTful controller become eligible for passing parameters to them. Let’s explore this in action on an example of a RESTful controller called “AccountsController”. While this controller could have many methods, we will only focus on one of them, “getShow” method. To be able to accept a parameter for this method you just have to register it in the method’s definition (listing 5.20):

Listing 5.20 Registering a parameter for a method of a RESTful controller

```
class AccountsController extends BaseController
{
    ...
    public function getShow($id)
    {
        return 'Showing account '.$id;
    }
    ...
}
```

Now if the user tries to access the URL “accounts/show/100” he or she will see a response that says “Showing account 100” (screenshot in figure 5.13):

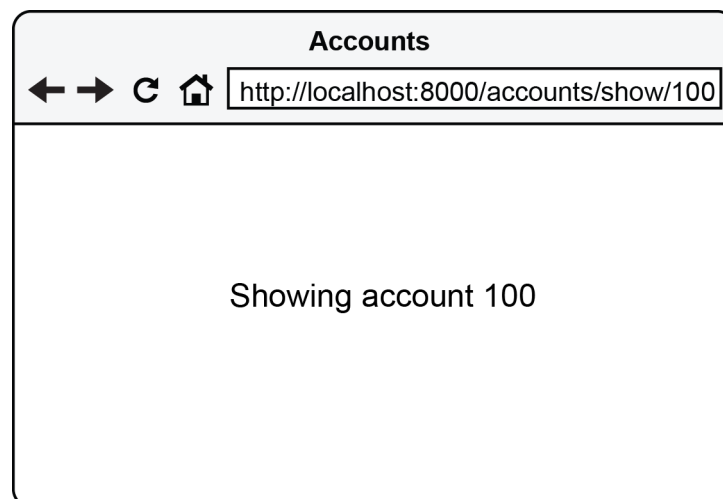


Figure 5.13 Result of passing a parameter to a method of RESTful controller

When it comes to RESTful controller routing there is one special detail about passing multiple parameters. Laravel assumes that multiple parameters in the method definition should be equal to parameters in the URL separated by a slash sign. Imagine that the “getShow” method has three parameters instead of one. Let’s change the method definition to accommodate these three parameters (listing 5.21):

Listing 5.21 Accepting multiple parameters in a method of a RESTful controller

```
public function getShow($id, $name, $phone)
{
    return 'Showing account with ID:'. $id.', name:'. $name.', phone:'. $phone;
}
```

Without doing any changes in the routing file, the “getShow” method will now accept three parameters and display them to you if you access the URL “accounts/show/12/Laravel/555-555-5555” (screenshot in figure 5.14):

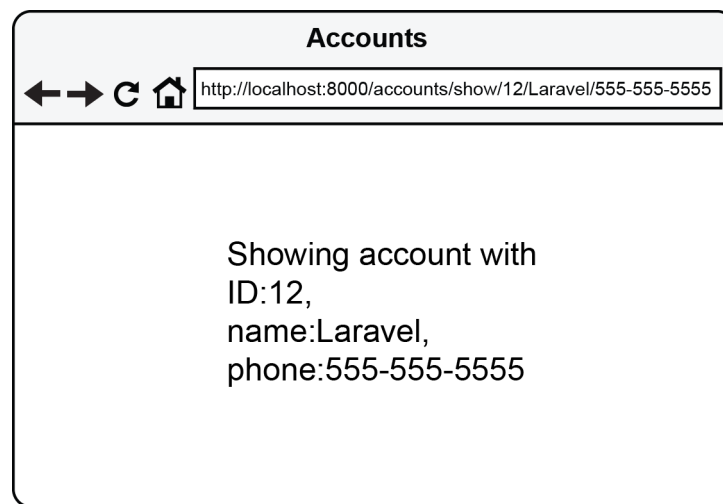


Figure 5.14 Result of executing a method with three parameters



Note

You can provide a maximum of up to 5 parameters to a method in a RESTful controller

As you can see, passing parameters to RESTful controllers is easy and logical without any modifications in the routing definition – just specify which methods will take the parameters and Laravel will do the rest for you. Let’s take a look at the last type of controller whose methods could take parameters – Resource Controllers.

5.6.3 Passing parameters to methods of a Resource Controller

Resource Controllers don’t have any flexibility when it comes to passing parameters to their methods. In fact, the only four methods of a Resource Controller that are able to take a parameter are:

- show

- edit
- update
- destroy

And these four methods are only able to take a single parameter, usually an ID of the specific resource. For example if you had a Resource Controller responsible for different books stored in the system, let's say "BooksController", you would use the parameter of the "show" method to retrieve a specific book from the resource (listing 5.22):

Listing 5.22 Passing the ID of an item to a method of Resource Controller

```
// app/routes.php
Route::resource('books', 'BooksController');

// app/controllers/BookController.php
class BooksController extends BaseController {
    ...
    public function show($id){
        return 'Showing a book with ID '.$id;
    }
    ...
}
```

Using the route and class definition from listing 5.22, if you access the URL "books/show/25" you would see the ID of the requested book, as you can see in figure 5.15:

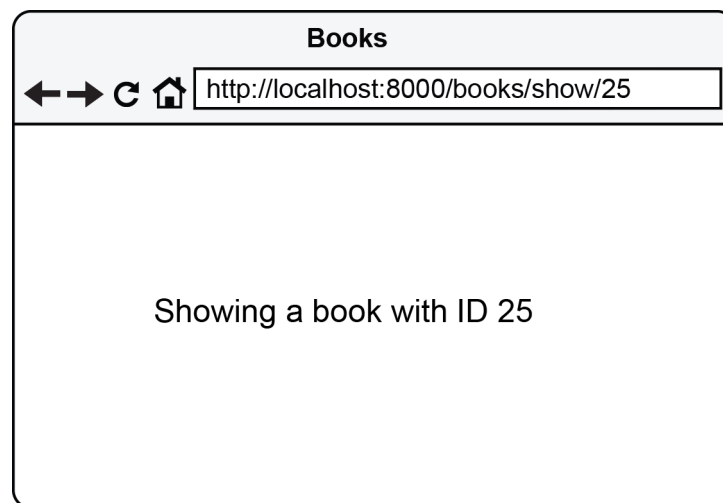


Figure 5.15 Result of passing a parameter "25" to method "show" of "BooksController" Resource Controller

Methods "edit", "update" and "destroy" behave similarly to the "show" method and do not allow you to pass more than one parameter.

5.7 Using filters with controllers

Laravel allows attaching a custom function that would be executed before or after execution of a controller's methods. These functions are called “filters” and you have met them before in chapter 3 in the context of routing. As you might remember, filters make it possible to put certain limits on the application flow depending on the user's interaction with the application. For example, you could use filters for:

- Restricting certain parts of application to be accessible to logged in users only
- Protecting the application from cross-site request forgery
- Logging and reporting

While the possibilities for custom filters are practically endless, filters that come with Laravel by default are mainly for security purposes. Default filters are stored in “app/filters.php” file (listing 5.23):

Listing 5.23 Default filters in app/filters.php file

```
<?php
// Global filters that are executed at the beginning and at the end of every
// request to the application
App::before(function($request) {});
App::after(function($request, $response) {});
// Filter that prevents users that are not logged in from accessing the content
// of the application and redirects to "login" URL relative to site's root
Route::filter('auth', function()
{
    if (Auth::guest()) return Redirect::guest('login');
});
// Filter that enables the use of basic authentication (great for building APIs)
Route::filter('auth.basic', function()
{
    return Auth::basic();
});
// Filter that prevents not logged in users from seeing pages that are strictly
// meant for logged in users
Route::filter('guest', function()
{
    if (Auth::check()) return Redirect::to('/');
});
// Filter that compares CSRF tokens on form submission, a useful security feature
Route::filter('csrf', function()
{
    if (Session::token() != Input::get('_token'))
    {
        throw new Illuminate\Session\TokenMismatchException;
    }
});
```

Laravel gives a lot of flexibility when it comes to applying these filters to application's execution. Any of the filters defined in “app/filters.php” file could be attached to any methods of any controller. The process of attaching these filters consists of executing a special function on controller's initialization. Let's take a closer look at how filters could be applied to controller's methods.

5.7.1 Attaching filters to a controller

You can attach a filter from the filters defined in “app/filters.php” file to the methods of a controller by specifying them in controller's constructor. As you remember filters could be executed before or after the execution of the functionality that they are attached to.

When it comes to using controller filters this concept remains the same. The two types of filters are “before” and “after” and the difference between them is the order in which they would be executed. For example if you wanted to attach the “auth” filter that would run before certain actions of “AccountsController” and make sure that the user is logged in, you would add it to controller's constructor in a way shown in listing 5.24. Please note that the second parameter of “beforeFilter” is omitted for now:

Listing 5.24 Attaching a “before” filter to a controller

```
class AccountsController extends BaseController
{
    public function __construct()
    {
        $this->beforeFilter('auth', ...);
    }

    public function getShow($id){...}
}
```

In case with controller filters, the two types of filters that Laravel provides are:

- beforeFilter
- afterFilter

These two types of filters could be applied to controller's methods using syntax like in figure 5.16:

type of filter
(before or after) filter which method or
 type of request
 to apply filter on

\$this->beforeFilter('auth', ...)

Figure 5.16 Structure of a controller filter definition

Now because the “auth” filter was provided as the first argument of “beforeFilter”, the closure function to “Route::filter(‘auth’)” in “app/filters.php” would be fired before the controller method(s) that the filter is applied to.

5.7.2 Applying filters to specific methods of a controller

Laravel allows you to fine tune when controller filters will be executed. In some cases you would want to apply a filter to only a certain controller method or a set of methods. In other cases you would want to apply a filter on a specific type of HTTP request (“get”, “post”, etc.). Laravel accommodates those situations by providing you with these three keywords:

- on
- except
- only

5.7.2.1 Using the “on” keyword

Using the “on” keyword allows you to limit execution of the filter to only specific type of the HTTP request. To use the “on” keyword provide it as a key of an array specifying what type of HTTP request you would want to apply it to. For example if you wanted to apply the “auth” filter only to requests of type “POST” you would provide an array with key/value of ‘on’ ⇒ ‘post’ as a second argument to the “beforeFilter” function (listing 5.25):

```
1 class AccountsController extends BaseController
2 {
3     public function __construct()
4     {
5         $this->beforeFilter('auth', array('on' => 'post'));
6     }
7     ...
8 }
```



Note

You can replace ‘post’ with any HTTP request types supported in Laravel.

5.7.2.2 Using the “except” and “only” keywords

If you wanted to limit the execution of the filter to only specific methods of the controller you could use the keywords “except” and “only”. For example to limit execution of “auth” filter to run for all controller’s methods, except the “showLogin” method, you can provide the “except” keyword as a key of an array specifying which method the filter wouldn’t apply to. Listing 5.26 shows an example where the showLogin method of LoginController will not have the “auth” filter executed:

Listing 5.26 Attaching “auth” filter to all methods of a controller except “showLogin”

```
class AccountsController extends BaseController
{
    public function __construct()
    {
        $this->beforeFilter('auth', array('except' => 'showLogin'));
    }
    ...
}
```

Great! But what if you wanted to attach the filter to only a few specified methods instead of attaching it to all methods? The “only” keyword is to the rescue in that case! It allows you to provide an array listing which methods of the controller should have a filter applied to them. Listing 5.27 shows a simple example where the “auth” filter is applied only to “updateProfile” and “showUser” methods:

Listing 5.27 Attaching “auth” filter to only few methods of a controller

```
class AccountsController extends BaseController
{
    public function __construct()
    {
        $this->beforeFilter('auth', array('only' => array( 'updateProfile', 'showUser' )));
    }
    ...
}
```

With this filter firing before the user can run the “updateProfile” or “showUser” methods ensures that only logged in users can execute those methods.

The flexibility of attaching filters to controller actions comes in very handy when you are building a large application with fine tuned access control. It allows you to get very creative with how you want to handle the requests to your application.

5.8 Summary

Controllers play an incredibly important part of web applications. Using them to control the execution flow in your applications is irreplaceable if you want your applications to be maintainable, extendable and testable. In this chapter you have explored in detail how to create controllers of three various types that Laravel supports: Basic, RESTful and Resource.

You have seen the advantages and disadvantages of each type of controllers, examples of their practical application and shortcuts that accelerate creation of Resource Controllers. You have also learned how to tell your application to route to these types of controllers and how to pass parameters to controllers’ methods. The information you have learned here in this chapter will be essential for the next chapters of the book.

In the next chapter we will take a look at database operations in Laravel applications.

6. Database operations

This chapter covers

- Configuring Laravel to use a database
- Introduction to Query Builder and Eloquent ORM
- Using Query Builder operators for managing data
- Performing Join queries

Working with data stored in a database is an essential feature of any modern web application. Saving user's orders and preferences, storing information about the products, creating data about uploaded files, all of these are just a few examples of application's actions that could use a database for storing data.

Managing data in the database used to be a painful process for PHP developers. Absence of built-in security, lack of common standards and consistency, messy database operations intermixed with the rest of the application's code have plagued PHP applications for many years. All of these and more problems concerning database management and database operations have mostly been solved by modern PHP frameworks like Laravel.

With Laravel's powerful database operators creating and managing records in a relational database like MySQL or PostgreSQL becomes much simpler and much more secure. Laravel interprets the data stored in the database into objects that you can easily manipulate like you would manipulate any other PHP object. This allows the developer to focus on other parts of the application instead of figuring out complex Structured Query Language (SQL) queries necessary to work with the data.

In this chapter you will learn how to set up Laravel to work with one of the many database engines that it supports. You will be then introduced to Laravel's Query Builder that makes creating and managing data in the database a breeze. After that you will get an introduction to Laravel's powerful Object Relational Mapping (ORM) called Eloquent and see how it could help manage related data. Then you will learn how you can use Laravel's query chaining and Query Builder operators to filter, sort, group and combine retrieved data for further operations.

6.1 Introducing Database Operations in Laravel

Laravel makes connecting to a database and managing data in it extremely simple and secure. Instead of requiring the developer to write complicated SQL queries it provides a convenient way of telling the application how you want to manipulate the data and Laravel would automatically translate those commands into SQL queries behind the scenes. As you will see throughout this chapter, from simple operations like insertion and updating of records to complex filtering and sorting to working with interrelated data, Laravel has the right tool for the job.

Laravel doesn't lock you into using a single database engine. Out of the box, it supports the following relational database systems:

- MySQL
- Postgres
- SQLite

- SQL Server

You can use any of these engines in your application after a simple configuration of one of the application's configuration files. When the database is configured, you can immediately use it in your application by using clean and elegant syntax of either Eloquent ORM or Query Builder.



Please note

Laravel provides two ways of operating with data in the database, by using Query Builder or Laravel's Object-Relational Mapping (ORM) called Eloquent. This chapter will focus on the Query Builder, while the whole next chapter will be devoted to Eloquent

When the application's code contains any DB operations, behind the scenes Laravel converts this code into proper SQL statements that are then executed on any of the DB engines supported by Laravel. The result of the SQL execution is then returned all the way back to the application code and the application could use the result to either show data to the user, or process the data according to application's requirements (figure 6.1):

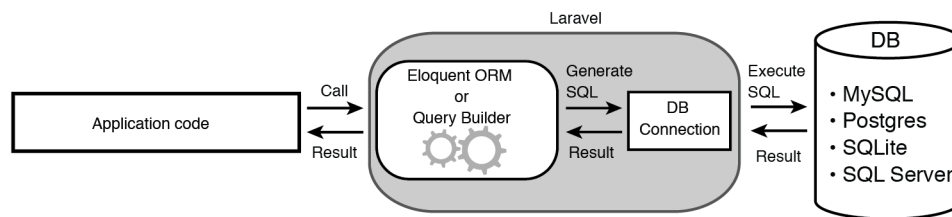


Figure 6.1 Database operations in Laravel

Laravel's flexibility and abundance of ways to work with relational data separate it from other frameworks. What other frameworks achieve with extensions and packages, Laravel has already built-in to help you manage data stored in a database efficiently. Some of the advanced database operations that Laravel provides out of the box are:

- Managing database tables and fields through special files called “migrations”
- Populating database with sample data (also called “seeding”)
- Working with separate read/write databases
- Pagination
- Many different relationship types and advanced methods of querying relations
- Database transactions
- Query logging

When combined together, these features make Laravel one of the best PHP frameworks to work with complex databases while keeping things simple for the developer.

Also, an important feature of database operations in Laravel is that all queries are run through PDO (PHP Data Objects) extension. PDO is a powerful interface for database operations and it comes with all versions of PHP following version 5.1. Laravel makes use of PDO parameter binding that provides additional security.



Please note

Laravel uses PDO extension to execute any database operations which adds an extra layer of protection against SQL injection attacks and allows the use of many database engines beside MySQL

You will get to experience Laravel's powerful database operations in action as you read through this chapter. First, let's learn what options Laravel provides for configuring the database settings.

6.1.1 Configuring database settings

Laravel makes it extremely easy to configure the settings for interacting with a database. All configuration settings of a Laravel application are located in "app/config" folder. The file that stores the configuration settings for the database is named "database.php".

To use the database in your application you only need to open up "database.php" file and specify connection credentials for an existing database. Laravel will then use those credentials to do any further database-related operations.



Please note

Note Laravel cannot create new databases. It can only manage existing databases

Out of the box, the database configuration file has almost everything ready for you to be able to use the database. Sometimes the developer's needs go beyond the conventions. The database configuration file provides you with the following options for database operations:

- Preferred method of object retrieval (PDO::FETCH_CLASS, PDO::FETCH_ASSOC, PDO::FETCH_OBJ or any other method available in PDO)
- Which database connection the application should use by default
- Connection settings for MySQL, Postgres, SQLite or SQL Server database
- Connection settings for Redis
- Name of the table that will be used for "migrations" if migrations are used

By default, the database connection is set to use MySQL engine, PDO is set to fetch objects through "PDO::FETCH_CLASS" and most of the connection settings are already in place. Listing 6.1 shows the contents of database configuration file that Laravel comes with:

Listing 6.1 Default database configuration (app/config/database.php)

<?php

```
return array(  
    // PDO fetch style (PDO::FETCH_CLASS, PDO::FETCH_ASSOC, PDO::FETCH_OBJ etc.)  
    'fetch' => PDO::FETCH_CLASS,  
  
    //The DB connection that will be used by the application for all DB operations  
    'default' => 'mysql',
```



```
'connections' => array(
  // SQLite connection settings
  'sqlite' => array(
    'driver'    => 'sqlite',
    'database'  => __DIR__.'../database/production.sqlite',
    'prefix'    => '',
  ),

  // MySQL connection settings
  'mysql' => array(
    'driver'    => 'mysql',
    'host'      => 'localhost',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
  ),

  // Postgres connection settings
  'pgsql' => array(
    'driver'    => 'pgsql',
    'host'      => 'localhost',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
    'charset'   => 'utf8',
    'prefix'    => '',
    'schema'    => 'public',
  ),

  // SQL Server connection settings
  'sqlsrv' => array(
    'driver'    => 'sqlsrv',
    'host'      => 'localhost',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
    'prefix'    => '',
  ),
),
'migrations' => 'migrations', // Name of the table used for migrations

// Redis connection settings
```

```
'redis' => array(
    'cluster' => false,
    'default' => array(
        'host'     => '127.0.0.1',
        'port'     => 6379,
        'database' => 0,
    ),
),
);
```

If you use a MySQL database on your machine, as all examples in this book are using, you will need to provide the name of an existing database. The default connection settings will most likely match your database for the exception of the username and password fields. Updating those configuration options is enough to start using a MySQL database on your local machine from Laravel!

Swapping database engines

One of the major advantages of Laravel using PDO for its database operations is that if you wanted to use a different database engine like Postgres instead of MySQL, you wouldn't change a single line of application's code and only database settings would need to be updated. This makes your applications much more future-proof and allows for easy swap out of database engines without rewriting any of application's code.

When you have the database settings configured, you are ready to start using the database in your application. Let's explore the two ways that you can use to work with databases in Laravel: Query Builder and Eloquent ORM.

6.1.2 Introducing Query Builder & Eloquent ORM

Laravel comes with two powerful sets of features to execute database operations, by using its own Query Builder or by using concept of "models" in Eloquent ORM. While you can use both in the same application (and as you will see later, even combine both to get the most flexibility out of DB operations) it helps to know the difference between the two. Let's first look at the Query Builder and then you'll meet Eloquent ORM.

6.1.2.1 Query Builder

Laravel's Query Builder provides a clean and simple interface for executing database queries. It is a powerful tool that can be used for various database operations such as:

- Retrieving records
- Inserting new records
- Deleting records
- Updating records
- Performing "Join" queries
- Executing raw SQL queries

- Filtering, grouping and sorting of records

Database commands that use the Query Builder use Laravel’s “DB” class and look like the following (listing 6.2):

Listing 6.2 Examples of using Query Builder to create, filter, calculate & update data

```
// Create three new records in the "orders" table
DB::table('orders')->insert(array(
    array('price' => 400, 'product' => 'Laptop'),
    array('price' => 200, 'product' => 'Smartphone'),
    array('price' => 50, 'product' => 'Accessory'),
));

// Retrieve all records from "orders" table that have price greater than 100
$orders = DB::table('orders')
    ->where('price', '>', 100)
    ->get();

// Get the average of the "price" column from the "orders" table
$averagePrice = DB::table('orders')->avg('price');

// Find all records in "orders" table with price of 50 and update those records
// to have column "product" set as "Laptop" and column "price" set as 400
DB::table('orders')
    ->where('price', 50)
    ->update(array('product' => 'Laptop', 'price' => 400));
```

Query Builder is very easy to use yet powerful way to interact with the data in your database. As noted before, in this chapter we will mainly focus on using the Query Builder and you will get to see most of its methods of working with database in action. Now let’s take a look at Eloquent ORM before diving deeper into the Query Builder and other concepts.

6.1.2.2 Eloquent ORM

Since its very first version Laravel featured an intelligent ORM called “Eloquent ORM”. Eloquent is Laravel’s implementation of Active Record pattern that uses the concept of “models” to represent data and it makes working with interrelated data easy. Eloquent is very good at working with complicated data relationships while remaining efficient, easy to manage and fast.

ORM, Active Record and Eloquent

Object-Relational Mapping (ORM) is a technique of representing data stored in database tables as objects that make database operations as easy as working with any other objects in PHP.

Active Record is an architectural pattern that enables representing database tables as special classes (also called “models”) and brings the concepts found in Object Oriented Programming into database operations.

Popular web development frameworks like CodeIgniter, Ruby on Rails and Symfony use Active Record

pattern to simplify database operations. Eloquent ORM uses Laravel's own powerful implementation of Active Record. When a database table is represented as a model, creation of a new row in that table is as easy as creating a new instance of the model. Working with tables and with the data stored in them becomes similar to working with any other classes and objects in PHP and even complex data relationships are easy to manage with Eloquent.

To get started using Eloquent you only need to configure the database connection settings in “app/config/database.php” file and create “models” that correspond to tables of your database. The Eloquent syntax looks not much different from plain OOP PHP code so it is very easy to read. Eloquent provides many advanced features to manage and operate on data, the most prominent of them are:

- Working with data by the use of “models”
- Creating relationships between data
- Querying related data
- Conversion of data to JSON and arrays
- Query optimization
- Automatic timestamps

From simple data insertion to association of data between tables to complex filtering within related data, Eloquent has you covered. The clean and logical syntax is something that separates Eloquent from other existing ORMs. Listing 6.3 shows an example of Eloquent syntax in action to create a new user, new order and then associate the new order to the newly created user:

Listing 6.3 Example of using Eloquent ORM to create and associate related data

```
// Create a model definition for the table “users” indicating its
// relationship to the “orders” table

// app/models/User.php
class User extends Eloquent {
    public function orders()
    {
        return $this->hasMany('Order');
    }
}

// Create a model definition for the table “orders”
// app/models/Order.php
class Order extends Eloquent {}

// application code

// Create a new row in “users” table
$user = new User;
$user->name = "John Doe";
```

```

$user->save();

// Create a new row in "orders" table and link it to the user
$order = new Order;
$order->price = 100;
$order->product = "TV";

$user->orders()->save($order);

```

The example above is just barely scratching the surface of Eloquent’s features. There is just so much about Eloquent that it deserves its own chapter and you will get to experience this mighty ORM at a much deeper level in the next chapter. Right now let’s take a closer look at using the Query Builder.

6.2 Using Query Builder

From inserting and managing records to sorting and filtering, Query Builder provides you with convenient operators to work with the data in an existing database. Most of these operators could be combined together to get the most out of a single query.

Application code using Query Builder uses Laravel’s “DB” class. When the application executes any command of class “DB” (a command could also be called an “operator”), Laravel’s Query Builder will build an SQL query that will be executed against a table specified as an argument to “table” method (figure 6.2). That query will be executed on a database using the database connection settings specified in the “app/config/database.php” file. The result of the query execution will then be passed all the way back to the caller changing its state: by returning retrieved records, a Boolean value or returning an empty result set.

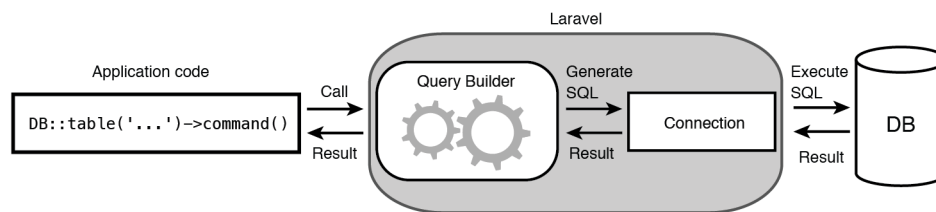


Figure 6.2 Executing database operations using Query Builder

The operators that Query Builder provides for some of the most common operations are listed in table 6.1 along with descriptions of these operators:

Table 6.1 Most commonly used operators of the Query Builder

| Operator | Description |
|---------------------------------|--|
| <code>insert(array(...))</code> | insert a new record using the data provided as an array of key – value pairs |
| <code>find(\$id)</code> | retrieve a record that has a primary key “id” equal to the provided argument |
| <code>update(array(...))</code> | update a record using the data provide as an array of key – value pairs |
| <code>delete()</code> | delete a record |
| <code>get()</code> | retrieve the record as an object containing the column names and the data stored in the record |
| <code>take(\$number)</code> | retrieve only a limited number of records |

Throughout the next few pages you will get to see the Query Builder in action and meet a few of its powerful operators listed in the table 6.1. Let's start with the basics, inserting new records into an existing table in a database.

6.2.1 Inserting records

The “insert” operator inserts a new row (record) into an existing table. You can specify what data to insert into the table by providing an array of data as a parameter to “insert” operator. Let's take a look at a simple example. Imagine that you have a table called “orders” that has 3 fields (columns) in it: auto incrementing ID, price and a name of the product (table 6.2).

Table 6.2 Table structure for the “orders” table

| Key | Column | Type |
|---------|---------|-----------------------------|
| primary | id | int (11), auto-incrementing |
| | price | int (11) |
| | product | varchar(255) |

6.2.1.1 Inserting a single record

Let's use the “insert” operator of Query Builder to insert a new record into this table. By passing an array formatted as column-value pair you can tell the Query Builder to insert a new record using the data passed as a parameter to the “insert” operator (listing 6.4):

Listing 6.4 Inserting a new record using Query Builder

```
// Tell the query builder to do an insertion on the “orders” table
DB::table('orders')->insert(
    array(
        'price' => 200, // Set a value for the column “price”
        'product' => 'Console' // Set a value for the column “product”
    )
);
```

When this code is executed the following process will take place behind the scenes. Laravel's Query Builder will translate the “insert” command into SQL query specific to the database that is currently in use by the application (specified in “database.php” configuration file). The data passed as an argument to the “insert” command will be placed into the SQL query in form of parameters. The SQL query will then be executed on the specified table (“orders”) and the result of query execution will be returned to the caller. Figure 6.3 illustrates this whole process:

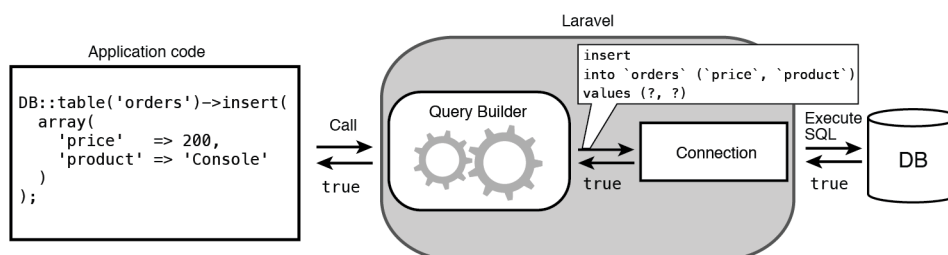


Figure 6.3 Behind the scenes process of running “insert” operator

You might have noticed the question marks instead of actual values passed to the SQL query above. As noted in the introduction section of this chapter, Laravel uses PDO to execute SQL statements. Behind the scenes PDO will attach the data (in this case “200” and “Console”) to the unnamed placeholders seen as question marks. Using a prepared statement by including placeholders for data adds protection from SQL injection and increases security of insertions and updates of data. As a developer you don’t need to enable this in any way, Laravel does this for you out of the box. Now that you know how to insert a single record, let’s also take a look at inserting many records.

6.2.1.2 Inserting multiple records

The same “insert” operator of the Query Builder could be used to insert more than one row of data. Passing an array containing arrays of data allows you to insert as many rows as you would like to, two, three or a thousand. Listing 6.5 shows the use of “insert” operator to create three new rows of data:

Listing 6.5 Inserting multiple records using Query Builder

```
// Create three new records in the "orders" table
DB::table('orders')->insert(array(
    array('price' => 400, 'product' => 'Laptop'),
    array('price' => 200, 'product' => 'Smartphone'),
    array('price' => 50, 'product' => 'Accessory'),
));
```

When executed, the “insert” statement in listing above will create three new records, and the SQL query that Laravel will build behind the scenes to do that is:

```
1 insert into `orders`(`price`,`product`) values (?, ?),(?, ?),(?, ?)
```

As you can see Laravel is smart enough to do the insertion of three rows of data in one query instead of running three separate queries. Now that you can easily insert data using Query Builder, let’s learn how to retrieve data.

6.2.2 Retrieving records

Query Builder provides a few convenient operators of getting the data from the database. To accommodate many different situations it is very flexible, allowing you to do the following operations:

- Retrieve a single record from a table
- Retrieve all records in the table
- Retrieve only specific columns of all records in the table
- Retrieve a limited number of records in the table

Over the course of the next few pages we will look at these operators of retrieving data. Let’s start with the basics, getting a single record from the database.

6.2.2.1 Retrieving a single record

You can retrieve a single record from the table by using operator “find”. Just provide the value of the primary key of the record that you’d like to retrieve as a parameter to operator “find” and Laravel will return that record as an object or return NULL if the record is not found. The example in listing 6.6 illustrates using “find” to retrieve a record with the primary key set to “3”:

Listing 6.6 Retrieving a single record using operator ‘find’

```
// Retrieve a record with primary key (id) equal to "3"
$order = DB::table('orders')->find(3);

// If there is a record with ID "3", The $order variable will contain:
object(stdClass)#157 (3) {
    ["id"]=>
    string(1) "3"
    ["price"]=>
    string(3) "200"
    ["product"]=>
    string(10) "Smartphone"
}
```

**Please note**

Operator “find” expects the column containing record’s primary key to be called “id”. You will need to use other operators of retrieving records if your table doesn’t use “id” as the primary key

Running the code in listing 6.6 would execute the following SQL, binding the value passed into “find” operator as a parameter and limiting the number of returned records to a maximum of 1:

```
1 select * from `orders` where `id` = ? limit 1
```

As you can see, retrieving a single record is easy with the use of operator “find”. What if you wanted to retrieve all records from a table?

6.2.2.2 Retrieving all records in the table

To retrieve all records from a table you can use operator “get” without any parameters. Running “get” on a specified table without any prior operators would return all records from that table as an array of objects. Imagine that you wanted to retrieve all records from the “orders” table. Listing 6.7 below shows how you would get all data from that table as an array of objects:

Listing 6.7 Retrieving all records using ‘get’

```
// Using “get” without any prior operators will grab all records in the
// specified “orders” table
$orders = DB::table('orders')->get();
```

Running operator “get” without any parameters would run the following SQL query behind the scenes and return its result as an array of objects:

```
1 select * from `orders`
```

The result returned from execution of code in listing 6.7 would have an array of objects containing data of all rows in the table. Each object would have array’s keys storing the names of the columns of the table and array’s values storing the row’s values. Listing 6.8 shows the resulting array of objects:

Listing 6.8 Result of retrieving all records (contents of orders table)

```
array(4) {
  [0]=>
  object(stdClass)#157 (3) {
    ["id"]=>
    string(1) "1"
    ["price"]=>
    string(3) "200"
    ["product"]=>
    string(7) "Console"
  }
  /* ... 3 more rows returned as objects ... */
}
```

What if you wanted to retrieve just a few specific columns of all records of the table instead of all columns? Let's take a look how you could do that with Query Builder.

6.2.2.3 Retrieving specific columns of all records in the table

To get only specific columns of all records in the table you can still use the “get” operator, but this time pass the desired column names as an array of parameters to the “get” operator. For example if you wanted to retrieve columns “id” and “price” of all records, while omitting “product” column, you would pass “id” and “price” as a parameter to the “get” operator like in listing 6.9:

Listing 6.9 Retrieving specific columns of all records in a table

```
// Tell the Query Builder to only retrieve “id” and “price” columns
$orders = DB::table('orders')->get(array('id', 'price'));
```

Running this statement would produce the following SQL:

```
1 select `id`, `price` from `orders`
```

The returned data in this case would contain an array of objects as follows in listing 6.10:

Listing 6.10 Result of retrieving specific columns of all records (contents of \$orders)

```
array(4) {
  [0]=>
  object(stdClass)#157 (2) {
    ["id"]=>
    string(1) "1"
    ["price"]=>
    string(3) "200"
  }
  ... 3 more rows returned as objects ...
}
```

Laravel's Query Builder tries to make retrieval of data extremely simple and efficient. The flexibility of specifying which columns of the table you'd like to get simply by passing the names of the columns to "get" operator could be very useful. It could come handy when your application works with big amounts of data. Retrieving only specific columns from the table instead of the whole table cuts down the amount of RAM your application uses.

What if you wanted to limit the maximum number of records that a Query Builder statement would return? Let's take a look at how you can tell Query Builder to only retrieve a specific number of records.

6.2.2.4 Retrieving a limited number of records

To specify a maximum number of records that you'd like to get from a table you can use operator "take" with a number of records passed as a parameter to it and appending operator "get" to the query. For example imagine if you had 1000 records in the "orders" table and you wanted to only retrieve a maximum of 50 records from it. Using the operator "take" with 50 passed as a parameter, you can. Listing 6.11 shows this in action:

Listing 6.11 Retrieving a limited number of records from a table

```
$orders = DB::table('orders')->take(50)->get();
```

Executing this query would result in \$orders being an array of objects with maximum of 50 objects that store the data retrieved from the table. Now that you know how to create and retrieve data in various ways, you are ready to learn how to update existing data in the database.

6.2.3 Updating records

Updating records using Query Builder is very similar to creating new records. To update the data of an existing record or a set of records you can append operator "update" to the query and pass an array of new data as a parameter to it. You can use query chaining, a concept that we will explore later in this chapter, to target specific records that would be updated.

6.2.3.1 Updating specific records

To update only specific records in the table you would need to somehow tell Laravel which records you want updated. Fortunately you can use Query Builder's filtering operators and append the "update" operator to the end of the query targeting only specific records.

One of the filtering operators is operator "where" that allows you to choose records by specifying a column, comparison operator and a value to compare the data to. For example if you wanted to update all records in table "orders" that have price set to more than 50, you would construct the following query (listing 6.12):

Listing 6.12 Updating a column of all records in a table that match a criterion

```
// Tell the Query Builder to target all rows with column 'price' set to more than 50 in
// table "orders"
DB::table('orders')
->where('price', '>', '50')
->update(array('price' => 100)); // Update column 'price' with a new value
```

Executing this query would generate the following SQL query:

```
1 update `orders` set `price` = ? where `price` > ?
```

You are not limited to updating just one column of the records. Passing more key-value pairs in the array supplied to “update” operator as a parameter would update all specified columns. What if you wanted to update all records in the table at once?

6.2.3.2 Updating all records in the table

Even though it is a rare case to update all records in a table, you can update columns of all records in the table by appending “update” operator to the beginning of the DB query. Listing 6.13 shows an example of updating the column “product” of all records in the “orders” table:

Listing 6.13 Updating a column of all records in a table

```
// Tell the Query Builder to update column 'product' of all records to be 'Headphones'
DB::table('orders')->update(array('product'=>'Headphones'));
```

Executing this query would generate the following SQL query:

```
1 update `orders` set `product` = ?
```

As you can see, Query Builder allows updating the records in a table in multiple ways that are easy to write. In the next section we will take a look at deleting existing records.

6.2.4 Deleting records

Deleting records from a table using Query Builder follows the same patterns as updating records. You can delete specific records that match some criteria or delete all records from the table by using operator “delete”.

6.2.5 Deleting specific records

To delete specific records in the table you can also use filtering operators like “where” to target records matching some sort of filtering criteria. For example if you wanted to delete the records that have “product” column set to “Smartphone”, you would append “delete” operator to the query specifying which records to target (listing 6.14):

Listing 6.14 Deleting records that match a criterion

```
// Tell the Query Builder to target all records that have column "product"
// set to "Smartphone"
DB::table('orders')
->where('product', '=', 'Smartphone')
->delete(); // Delete records matching the criterion
```

Executing this query would generate the following SQL query:

```
1 delete from `orders` where `product` = ?
```

The query in listing 6.14 would affect only records that were targeted by the “where” operator. If you wanted to delete all records from a table, you would do it in a way similar to updating all records, by only having operator “delete” following “DB::table()” statement.

Now that you know how to insert, retrieve, update and delete records, let’s learn how to achieve precise control when executing any of these actions by filtering, sorting and grouping data.

6.3 Filtering, sorting and grouping data

When managing data in the database applications often need to have precise control over which records will be affected. This might be either getting exactly the set of data required by application's specifications or deleting just a few records that match some criteria. Some of these operations could get very complicated if you were to use plain SQL. Laravel's Query Builder allows you to filter, sort and group data while maintaining clean consistent syntax that is easy to understand. You can see a list of most commonly used operators to filter, sort, and group data in table 6.3 below:

Table 6.3 Query Builder operators for filtering, sorting and grouping data

| Operator | Description |
|---|--|
| <code>where('column', 'comparator', 'value')</code> | retrieve records that match a criterion |
| <code>orderBy('column', 'order')</code> | sort records by specified column and order (ascending or descending) |
| <code>groupBy('column')</code> | group records by a column |

Over the next few pages you will get deeper understanding of how to operate on very specific records from the database using the operators from table 6.3. While getting to know the various ways of operating with data you will get to meet and use the concept of “query chaining”.

6.3.1 Query Chaining

Query chaining allows you to run many database operations in a single query. Imagine that you had a table “users” and wanted to retrieve an object containing all users with first name “John”, located in California and sorted by their birthdate. How would you do that in an efficient way?

One way would be to first retrieve an object containing all users with first name “John”, then do further filtering using PHP's “foreach” loops to get to the result that you want. While this sounds acceptable in theory, it is not efficient and could take up a lot of server's RAM. In addition, if the requirements for the application change, there would be no way to quickly update code like that to accommodate new functionality. Of course like with many other common web development problems, Laravel provides a better option.

Instead of creating separate queries to filter and sort data, you could put many of them together, effectively “chaining” them. A query chain combines a sequence of various actions that can be performed with data one after another to get a specific result that can be operated on. Filtering data by various parameters, sorting data and more could be represented as a series of actions performed upon data in a table (figure 6.4):

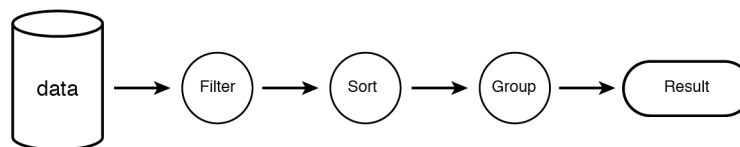


Figure 6.4 Concept of chaining actions together to get specific data from the database

Laravel allows you to put as many queries together as you want. From running two operations to ten and more, query chaining can significantly cut down on the amount of code you need to write to execute complicated database operations. For example to execute aforementioned actions on the “users” table you could put the filtering and sorting together into a single query chain like on the left side of figure 6.5:

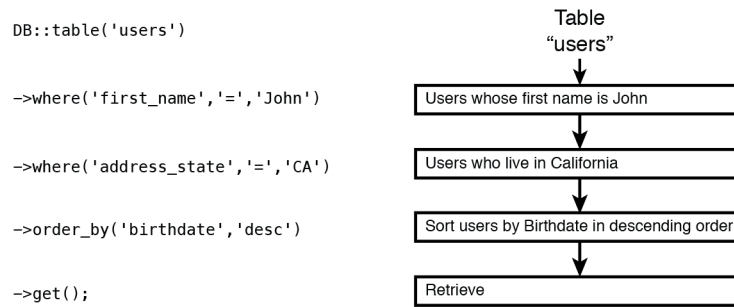


Figure 6.5 Example of query chaining in action



Note

You can use query chaining to execute multiple operations like sorting, filtering, grouping to precisely target a set of data that can be further retrieved, updated or deleted. You cannot mix insert/get/update/delete operations together in a single query.

While query chaining is not a new concept in the world of web development frameworks, Laravel provides one of the most elegant ways to do it. Let's look at the first operator that demonstrates the use of query chaining, operator "where".

6.3.2 Using "where" operator to filter data

Query Builder's operator "where" makes filtering records easy. It provides a clean interface for executing SQL "WHERE" clause and has syntax that is very similar to it. Some of the operations that you could do with this operator include:

- Selecting records that match a certain criterion
- Selecting records that match either one criterion
- Selecting records that have a column that lies in a certain range of values
- Selecting records that have a column that is out of range of values

After the records are selected using "where" you can do any of the operations discussed previously: retrieval, updating or deleting. Alternatively you could apply more "where" operators by the use of query chaining to achieve more precise matching. The flexibility that Laravel provides for filtering data is truly superb. Let's start exploring the powerful operator "where" by looking at using it in a simple query.

6.3.2.1 Simple "where" query

A query using "where" consists of supplying three parameters that will be used to filter the data:

- The name of the column that will be used for comparison
- The operator that will be used to compare data
- The value that the columns' contents will be compared to

The diagram in figure 6.6 shows the syntax of using operator “where”:

```

      Column name      Operator      Value
      |               |             |
where( 'first_name', '=', 'John' )

```

Figure 6.6 Syntax of using operator “where”

Operator “where” can be used not only for finding entries that exactly match the desired value. In fact you can use any of the operators supported by the database to specify what kind of matching you need. For numerical comparison you might want to use “>” or “<” operators to get data that has values of a certain numerical column greater or less than provided value. For string comparison you might use “like” or “not like” comparison operator.



Note

If you pass only two parameters to operator “where”, Laravel will assume that you want to do strict checking (using “=” to compare) which in certain cases can cut down on the amount of code even more

A list of allowed comparison operators that work in all databases supported by Laravel is presented in table 6.4 below:

Table 6.4 Allowed comparison operators for “where” operator

| Operator | Description |
|--------------|---|
| “=” | Equal |
| “<” | Less than |
| “>” | Greater than |
| “<=” | Less than or equal |
| “>=” | Greater than or equal |
| “<>” or “!=” | Not equal |
| “like” | Simple pattern matching (you can use the “%” sign with this operator to define wildcards both before and after the pattern) |
| “not like” | Negation of simple pattern matching |

Besides using a single “where” operator you can chain more than one “where” to filter the results even further. Under the covers Laravel will automatically put an “AND” in the SQL statement between the chained “where” operators. For example if you wanted to get all users that have last name starting with letters “Sm” and whose age is less than 50, you would create a query chain as shown in listing 6.15:

Listing 6.15 Chaining multiple “where” operators

```
// Use table “users”
$users = DB::table('users')
    // Match users whose last_name column starts with “Sm”
    ->where('last_name', 'like', 'Sm%')
    // Match users whose age is less than 50
    ->where('age', '<', 50)
    // Retrieve the records as an array of objects
    ->get();
```

Running this query would produce the following SQL behind the scenes:

```
select * from users where last_name like ? and age < ?
```

Laravel allows chaining as many “where” operators as you need to allow very precise control over which records you would like to operate with. Let’s now look at using another operator, “orWhere” that complements “where” operator nicely.

6.3.2.2 Using “orWhere” operator

To select data that matches at least one criterion out of several could be achieved by using “orWhere” operator. This operator has exactly the same syntax as “where” operator and it has to be appended to an existing “where” operator in order for it to work.

Let’s imagine that you wanted to delete all records from the “orders” table that have been either marked with “1” in “processed” column or whose “price” column is less than or equal to 250. Using “where” and “orWhere” operators together and appending “delete” operator at the end will do just that. Listing 6.16 shows using “orWhere” operator to filter records and subsequently deleting the records that match at least one criterion:

Listing 6.16 Combining “where” and “orWhere” operators

```
// Use table “orders”
$orders = DB::table('orders')
    // Match orders that have been marked as processed
    ->where('processed', 1)
    // Match orders that have price lower than or equal to 250
    ->orWhere('price', '<=' ,250)
    // Delete records that match either criterion
    ->delete();
```

This query would generate the following SQL:

```
delete from orders where processed = 1 or price <= 250
```



Note

You can chain more than one “orWhere” operator together for most flexibility. We looked at using “where” and “orWhere” operators and you now know how to use them together. Now let’s look at “whereBetween” operator.

6.3.2.3 Using “whereBetween” operator

“whereBetween” operator comes handy when you want to match records that have a numerical column set to a value that falls within a certain range. The syntax of “whereBetween” is a bit simpler than that of “where” or “orWhere” operators (figure 6.7). This operator expects only two parameters, a column that will be used for matching and an array containing two numerical values indicating a range.

```
whereBetween('credit', array(10,200))
```

Figure 6.7 Syntax of “whereBetween” operator

Adding this operator will match the records that have the value of the specified column fall in the provided range. For example if you wanted to retrieve all records in the “users” table that have the value of the “credit” column set to anything with 100-300 range, you would use the following query:

Listing 6.17 Using “whereBetween” operator

```
// Use table “users”
$users = DB::table('users')
    // Match users that have the value of “credit” column between 100 and 300
    ->whereBetween('credit', array(100,300))
    // Retrieve records as an array of objects
    ->get();
```

The query in the listing above would execute the following SQL:

```
select * from users where credit between ? and ?
```

You have now learned many different ways to get the desired data. As you have seen, filtering data with the operator “where” and its derivatives doesn’t involve complex SQL statements when using Query Builder. The Query Builder has a lot more features that allow more control for the way the data is retrieved. Let’s take a look at sorting and grouping data.

6.3.3 Using “orderBy” to sort data

“orderBy” operator of the Query Builder provides an easy way to sort the data that is retrieved from the database. Just like “where” operator is similar to “WHERE” clause in SQL, “orderBy” operator is very similar to the “ORDER BY” clause found in SQL. Query Builder’s syntax for this operator is practically the same as it is in SQL. To sort a set of data by some column, two parameters need to be passed to the “orderBy” operator: the column by which to sort the data and the direction of sorting (ascending or descending). Figure 6.8 shows syntax of “orderBy” operator:

```
orderBy('price', 'asc')
```

Figure 6.8 Syntax of “orderBy” operator used to sort data

Applying “orderBy” operator to a set of data retrieved by one of the operators of Query Builder will sort the data according to the column name and the direction specified through the use of parameters. For example imagine that you had a table called “products” and wanted to sort the products by their price, in ascending order. You could use the query from listing 6.18 to do that:

Listing 6.18 Using “orderBy” operator to sort products by price

```
// Use table “products”
$products = DB::table('products')
    // Sort the products by their price in ascending order
    ->orderBy('price', 'asc')
    // Retrieve records as an array of objects
    ->get();
```

You could also use query chaining to accommodate more complex filtering and sorting scenarios. For example if you wanted to get all records from the “products” table whose “name” column contains letters “so” and whose “price” column is greater than 100 and sort the result by the “price” column in ascending order, you would construct the following query as in listing 6.19:

Listing 6.19 Using “orderBy” operator with other operators

```
// Use table “products”
$products = DB::table('products')
    // Get products whose name contains letters “so”
    ->where('name', 'like', '%so%')
    // Get products whose price is greater than 100
    ->where('price', '>', 100)
    // Sort products by their price in ascending order
    ->orderBy('price', 'asc')
    // Retrieve products from the table as an array of objects
    ->get();
```

“orderBy” operator is chainable just like “where” operator. You can combine multiple “orderBy” operators to get a sorted result that you need to achieve. If you also want to group records together, you can use the “groupBy” operator.

6.3.4 Using “groupBy” to group data

You can group records together using “groupBy” operator that is similar to SQL “GROUP BY” clause. It accepts only one parameter – the column by which to group the records by. For example if you had a table “products” and wanted to retrieve all records from it grouped by the product name, you would create the following query (listing 6.20):

Listing 6.20 Using “groupBy” operator to group data

```
// Use table “products”
$products = DB::table('products')
    // Group products by the “name” column
    ->groupBy('name')
    // Retrieve products from the table as an array of objects
    ->get();
```

Now that you know how to use filtering, sorting and grouping of records in various ways and are familiar with the basics of query chaining, we can go on to a bit more complicated topic and learn how to use Laravel’s advanced query operators to do “join” statements.

6.4 Using Join Statements

Laravel’s Query Builder supports all types of Join statements supported by the database you are using. Join statements are used to combine records from multiple tables that have values common to those tables. Consider two tables, “Users” and “Orders”, with contents as shown in figure 6.9:

| users | | orders | | |
|-------|------|--------|---------|--------|
| id | name | id | user_id | item |
| 5 | John | 100 | 5 | TV |
| 6 | Mark | 120 | 7 | Laptop |
| 7 | Sam | 122 | 5 | Phone |

Figure 6.9 Contents of two sample tables

While you can use Join statements to combine more than two tables, we will use only the two small tables from figure 6.9 to demonstrate some of the types of Join statements that you can use in Query Builder.



Note

Caution should be taken if the joined tables have columns with the same names. You can use “select()” operator to alias duplicate columns

6.4.1 Using Inner Join

Inner Join is a simple and most common type of Join. It is used to return all of the records in one table that have a matching record in the other table. For example if you wanted to know which users have placed an order you could use an Inner Join statement that would return a list of all users with orders in the “orders” table and data about each of their order.



Note

You can chain more than one Join statement together to join more than two tables

Inner Join in Query Builder follows the syntax in figure 6.10 below:

Name of the Key of the Operator Key of the
 second table first table second table
 join('orders', 'users.id', '=', 'orders.user_id')

Figure 6.10 Syntax of Inner Join

Let's apply this syntax to the "users" and "orders" tables, to see which records are matching between them. In our example tables the primary key in each table is the "id" column. Listing 6.21 shows the Join query applied using the "users" table as first table and "orders" table as the second table:

Listing 6.21 Using Inner Join

```
// Use table "users" as first table
$usersOrders = DB::table('users')
    // Perform a Join with the "orders" table, checking for the presence of matching
    // "user_id" column in "orders" table and "id" column of the "user" table.
    ->join('orders', 'users.id', '=', 'orders.user_id')
    // Retrieve users from the table as an array of objects containing users and
    // products that each user has purchased
    ->get();
```

When we run this query, the following SQL would be executed behind the scenes:

```
1 select * from `users` inner join `orders` on `users`.`id` = `orders`.`user_id`
```

This query would look for values of "user_id" column in the "orders" table that have a matching value in the "id" column in the "users" table. A Venn diagram on the left side of figure 6.11 shows the result of Inner Join as the shaded area between the two sets of data. If represented as a table, the resulting array of objects assigned to \$usersOrders after running the Join statement in listing 6.21 would look like the table on the right side of figure 6.11:

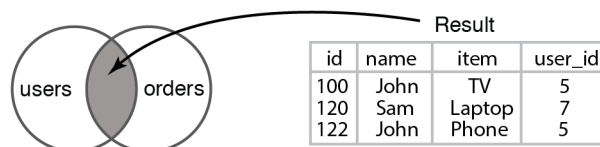


Figure 6.11 Result of running an Inner Join query between the "users" and "orders" tables

As you can see, running an inner Join could be useful and simple with the Query Builder. Now let's explore another type of join, a Left Join.

6.4.2 Using Left Join

Left Join is a bit more inclusive than the Inner Join and has syntax similar to it. It produces a set of records that match between two tables and in addition it returns all records from the first table that don't have a match. When creating this type of join the only difference in syntax is using "leftJoin" operator instead of "join" as shown in figure 6.12 below:

Name of the
second table
Key of the
first table
Operator
Key of the
second table
`leftJoin('orders', 'users.id', '=', 'orders.user_id')`

Figure 6.12 Syntax of Left Join

Let's use this type of Join on our two sets of data, the “users” and “orders” table to see what the result would look like. Listing 6.22 shows usage of the “leftJoin” operator to join the two tables:

Listing 6.22 Using Left Join

```
// Use table “users” as first table
$usersOrders = DB::table('users')
    // Perform a Left Join with the “orders” table, checking for the presence of
    // matching “user_id” column in “orders” table and “id” column of the “user” table.
    ->leftJoin('orders', 'users.id', '=', 'orders.user_id')
    // Retrieve an array of objects containing records of “users” table that have
    // a corresponding record in the “orders” table and also all records in “users”
    // table that don't have a match in the “orders” table
    ->get();
```

When we run this query, the following SQL would be executed behind the scenes:

```
1 select * from `users` left join `orders` on `users`.`id` = `orders`.`user_id`
```

This query would contain all rows from “users” regardless of having a matching entry in the “orders” table. The values of columns of the result would consist of the same values as if you had an Inner Join but those rows from the “users” table that don't have a match in the “order” would return NULL for “id”, “item” and “user_id” columns (figure 6.13). If represented as a table, the resulting array of objects assigned to \$usersOrders after running the Left Join statement in listing 6.22 would look like the table on the right side of figure 6.13:

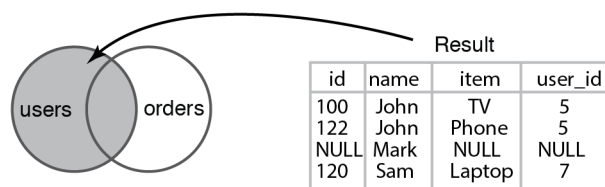


Figure 6.13 Result of running a Left Join query between the “users” and “orders” tables

Left Joins are useful in those cases when you need to get all records from the first table even if there are none matching records in the second table. There might be a need to use another type of Joins. Fortunately Laravel doesn't limit you by just Inner Joins and Left Joins. In fact, you can use any type of join supported by your database and we will learn how to do that in the next section.

6.4.3 Using other types of Joins

Laravel's Query Builder is so flexible that it takes into account special cases of Join queries and allows you to execute all types of Join queries supported by your database. While most SQL database engines (like

MySQL and SQLite) support Inner Right and Left Joins/Outer Right and Left Joins, other SQL engines like Postgres and SQL Server support Full Joins in addition to other types of Join queries. You can execute any type of Join that your database supports by supplying a fifth argument into “join” operator specifying which type of Join query you want to execute. Figure 6.14 shows the syntax of Join operator with a custom type of the Join query:

The diagram shows the following syntax for the join operator:

```
join('orders', 'users.id', '=', 'orders.user_id', 'right')
```

Labels with arrows pointing to the arguments:

- Name of the second table: 'orders'
- Key of the first table: 'users.id'
- Operator: '='
- Key of the second table: 'orders.user_id'
- Type of Join: 'right'

Figure 6.14 Using fifth argument of “join” operator for custom type of Join query

From cross joins to outer joins and full joins, the flexibility of the “join” operator would fit any possible Join query scenario. Listing 6.23 shows some of the custom types of Joins that work with all database engines supported by Laravel:

Listing 6.23 Using other types of Join queries

```
// Right Join
join('orders', 'users.id', '=', 'orders.user_id', 'right')

// Right Outer Join
join('orders', 'users.id', '=', 'orders.user_id', 'right outer')

// Excluding Right Outer Join
join('orders', 'users.id', '=', 'orders.user_id', 'right outer')
->where('orders.user_id', NULL)

// Left Join
join('orders', 'users.id', '=', 'orders.user_id', 'left')

// Left Outer Join
join('orders', 'users.id', '=', 'orders.user_id', 'left outer')

// Excluding Left Outer Join
join('orders', 'users.id', '=', 'orders.user_id', 'left outer')
->where('orders.user_id', NULL)

// Cross join
join('orders', 'users.id', '=', 'orders.user_id', 'cross')
```

Multiple types of chainable Join queries could make almost any complex database operation possible. Great support of Join queries along with other database operators you have seen up to this point makes working with database in Laravel a pleasant process instead of being a mundane operation.

6.5 Summary

In this chapter you have looked at Laravel's powerful methods of working with databases. From simple connection configuration to creation of complicated filtering and sorting scenarios Laravel provides convenient database operators at every step.

You have met Query Builder and have used its easy-to-follow syntax to insert, retrieve, update and delete records. You have used its "where" operators to filter records, "orderBy" to sort them and "join" to build various types of Join queries. Besides, on the way to getting a close look at the features of Query Builder you have learned an important concept of database operations in Laravel – query chaining that makes multi-stage database operations efficient and easy to work with.

The concepts you have learned so far in this chapter are fundamental to the next chapter where you will get to know one of the most elegant ORMs of modern web development frameworks, Eloquent. In the next chapter we will take a closer look at Eloquent ORM and become familiar with its smart ways to manage data and relationships between the data in Laravel-powered web applications.

7. Eloquent ORM

This chapter covers

- Introduction to Eloquent ORM and models
- Managing data with Eloquent ORM
- Creating relationships between data
- Using query scopes

Modern web applications are hungry for data. Remembering blog comments, user settings, order history, messages and more are common features of many web applications. It is not unusual to see a single application deal with 20, 30 or more different database entities, such as products, users, comments, orders, posts, etc., storing thousands and even millions of rows of data. With growing demand for information managing complex relationships between data and doing it efficiently becomes an essential and important part of almost any web application developed today.

Laravel's Object Relational Mapping (ORM) called Eloquent is one of the most powerful tools among modern PHP frameworks that simplify complex database operations. From storing and accessing data in a database to relating data entities and querying complex relations, Eloquent helps web developers to manage data effectively. Eloquent's advanced features like eager loading and interoperability with Query Builder allows for building and optimizing of even the most complex queries.

In this chapter you will learn how to use Laravel's powerful Eloquent ORM and will understand the benefits it provides when working with data in a database. You will first learn about conventions and flexibility of Eloquent. Then you will get to experience the methods it provides to manage and relate data entities effectively. You will also explore Laravel's methods of filtering and accessing specific data from the retrieved data set. Finally, you will learn how to work with the retrieved data, how to convert it to other formats like an array or JSON and how to display the data to the user.

7.1 Introducing Eloquent ORM

Eloquent is an intelligent implementation of Object-Relational Mapping (ORM) that comes with Laravel. It allows working with the data in the database in a convenient way, abstracting the database into objects that are easy to manipulate, familiar to any developer practicing Object-oriented programming. When these objects - also called "models", are created, updated, or deleted, corresponding changes are made in the database by Eloquent.



Definition

Object-Relational Mapping (ORM) abstracts the database structure and its contents into objects that are easy to work with in an object-oriented programming language. Eloquent is a name for Laravel's ORM.

Diagram in figure 7.1 below shows a simple example of Eloquent ORM in action, inserting a new row in table "users" by creating a new PHP object and assigning values to its attributes:

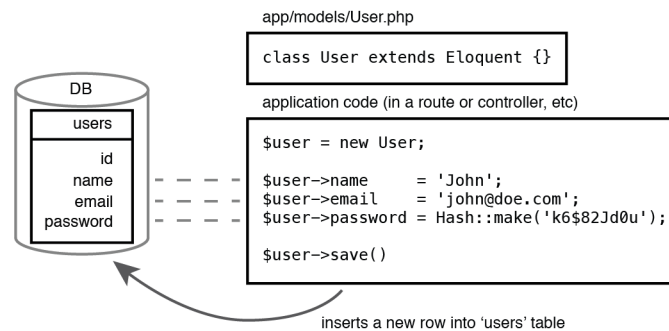


Figure 7.1 Using Eloquent to create a new record in the database

Eloquent simplifies working with complex database structures by enabling the developer to create relationships between data. Linking orders to users who placed them, assigning tags to blog posts, relating accounts to a company and more are possible with Eloquent's built-in relationships. Eloquent allows relating records between tables in the database in the following ways:

- Linking a record from one table to another record from a different table (One-to-One relationship)
- Linking a record from one table to many records from a different table (One-to-Many relationship)
- Linking a number of records from one table to a number of records in another table (Many-to-Many relationship)
- Creating a relationship where a record of one table could be dynamically linked to many other tables (Polymorphic relationship)
- And more

These built-in relationships between data entities in the database help the developer make sense of the data and easily work with it. For example if you were creating a blog, you could benefit from using the relational capabilities of Eloquent. Figure 7.2 below shows one of the possible scenarios of relationships between data entities in a blog application where each user can publish posts and each post can contain comments from other users:

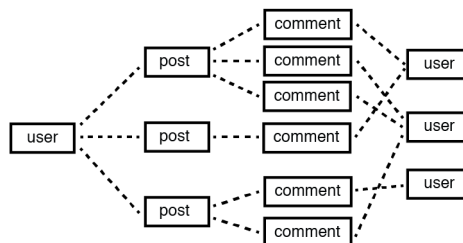


Figure 7.2 Example of relationships between the data in a blog

This and much more complex scenarios could be easily achieved with Eloquent's relationships. As you read through this chapter and get familiar with Eloquent ORM you will also get to know the most prominent relationship types it provides and see them in action.

Eloquent uses the concept of "models" to represent data in the database. Let's learn about that before diving any deeper.

7.1.1 Introducing models

Like in other ORM implementations, a model in Laravel's Eloquent is a special class that represents a single entity, usually a single table in the database. Using models allows working with data in the database in the same way as you would work with any other objects in PHP.



Note

In Laravel applications models are stored as PHP classes in the folder “app/models” and are child classes of Eloquent.

When these classes – models - use Eloquent's functionality, Laravel automatically generates SQL queries that are executed behind the scenes on the database and return a result (figure 7.3):

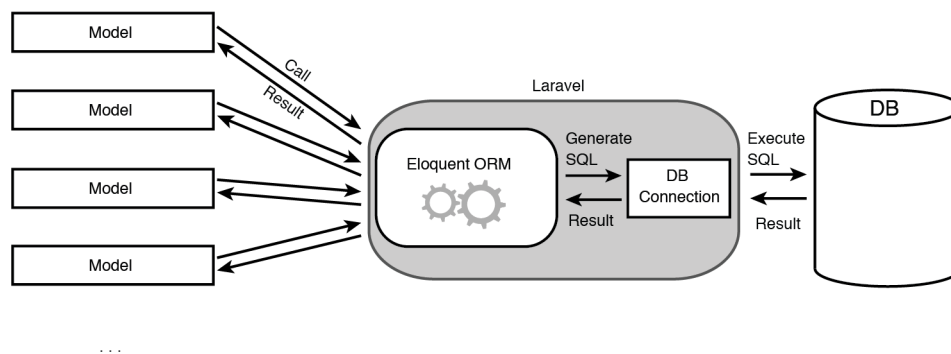


Figure 7.3 Interaction between Eloquent models and the database

As you will learn soon, when Eloquent model is defined, you can use it anywhere in your application to manage the data that it represents in the database. Let's first learn about some conventions that Eloquent follows, and then you'll be introduced to the basics of using models to manage data.

What are models in your application?

Think of models as objects or entities that your application works with. For example if the application is an online shop – the objects that it works with would be the products, orders, categories of products, users, etc. If the application were a social network – the objects would be status updates, photos, videos, locations, relationships between users, etc. Representing the entities in this fashion could greatly simplify the structure of your database and enable you to use Eloquent in your application.

7.1.2 Understanding conventions

Following convention over configuration approach, Laravel makes a few assumptions when it comes to database structure and Eloquent models. While these default assumptions could be easily overridden (you will

see later how to do that), following the conventions could save you a lot of time. The three main assumptions that Laravel makes when you are working with database tables through Eloquent's models are:

- Primary key on the table should be an unsigned auto-incrementing integer called "id".
- The table should have "created_at" and "updated_at" fields of type "datetime" to store the timestamps of record creation and of the updates to the record.
- The table name should be a lowercase plural of the model's class name that corresponds to the table. I.e. if you have a model defined in "app/models/User.php", the table should be called "users", if the model is "app/models/Product.php" - the table would be "products".

Laravel is smart when it comes to pluralizing table names and it can even handle irregular and uncountable names. Table 7.1 below shows an example of table names that Eloquent expects to have in the database if you have models that correspond to those tables:

Table 7.1 Examples of class names and table names assumed by Eloquent

| Class name (model) | Table name in DB |
|--------------------|------------------|
| User | users |
| Post | posts |
| Comment | comments |
| Order | orders |
| Profile | profiles |
| Map | maps |
| Movie | movies |
| Person | people |
| Series | series |
| Index | indices |



Note

If the name of the model is CamelCased (i.e. UserEvent) then the table name should consist of the words separated with an underscore and only the last word should be pluralized, i.e. user_events

While you can use almost any English words for the model class names, there are a few class names that are already in use by Laravel. Naming your model with a name of class in use could lead to class collisions unless the models are put into a separate PHP namespace. Table 7.2 below shows a list of class names that are in use by Laravel and are reserved:

Table 7.2 Class names that are reserved by Laravel

| | | | |
|------------|-------|----------|---------|
| App | Crypt | Lang | Route |
| Auth | Event | Log | Schema |
| Cache | File | Mail | Session |
| Config | Form | Password | View |
| Controller | Hash | Queue | |
| Cookie | Input | Request | |

Eloquent is very flexible and allows you to bypass built-in assumptions. As you will see later, defining your own table names instead of relying on Eloquent’s magic, specifying custom primary keys and disabling timestamps are all possible.

Now that you know these simple yet timesaving conventions, let’s learn how to create models and manage records in the database using Eloquent.

7.2 Creating and using models with Eloquent

Defining models allows you to manipulate data in the database using Eloquent. Each class in the “app/models” folder corresponds to a single entity - a table in the database. Coming back to an example used previously, imagine that you are creating an application to write blog posts. These blog posts would be stored in a table called “posts”. Creating an Eloquent model that corresponds to the “posts” table would allow for easy manipulation of data in that table (figure 7.3):

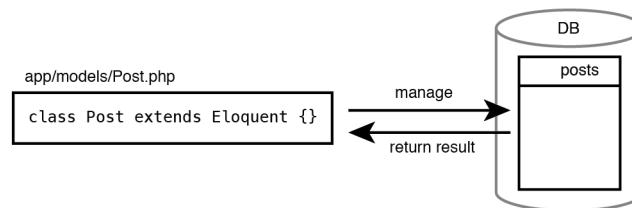


Figure 7.3 Defining a “Post” model abstracts the “posts” table as a PHP object

7.2.0.1 Creating models

To create a new model for a data entity, create a new child class of Eloquent and place it in “app/models” folder. As you have learned in the section about conventions, the name of this new class should be singular of the table name describing the data that will be stored in that table. Since we would like to store blog posts (“posts” table) and comments (“comments” table), we would have two models named “Post” and “Comment” respectively. Let’s create these models right now (listing 7.1):

Listing 7.1 Creating models for blog posts and for comments

```
// app/models/Post.php
<?php

class Post extends Eloquent {}

// app/models/Comment.php
<?php

class Comment extends Eloquent {}
```

Now that these classes - models are defined, you can use them to create, update, delete data in the database and take advantage of all benefits that come with Eloquent ORM.

7.2.0.2 Using models

When models are created and placed in “app/models” folder you can immediately use them in your application as you would use any other PHP class. Whether you place your code that works with models in routes, in controllers, or in other parts of the application – is up to you.



Note

Before using models make sure that the tables corresponding to them exist in the database, otherwise you’ll see an error stating that the table doesn’t exist when you are trying to work with the database

Eloquent provides many different methods that could be called by the models to perform actions in the database. A list of few basic operations and the purpose of each is listed in the table 7.3 below:

Table 7.3 Basic methods of Eloquent models

| Method | Description |
|-----------------------------|---|
| <code>all()</code> | Retrieve all records corresponding to the model |
| <code>find(\$id)</code> | Retrieve a record with primary key equal to \$id |
| <code>first()</code> | Retrieve the first record |
| <code>save()</code> | Save the current instance of a model as a record in DB |
| <code>create(\$data)</code> | Create a new record using an array of data passed as \$data |
| <code>delete()</code> | Delete current record or a set of records in DB |
| <code>destroy(\$id)</code> | Delete a record with the primary key equal to \$id |

Like you have experienced it with Query Builder, most of Eloquent’s methods are chainable and even better, they could be used with Query Builder interchangeably to get the most out of database operations. Throughout the next few pages you will explore the basic methods and then you will get into more advanced topics of Eloquent, such as using model relationships.

Let’s define table structure for “posts” and “comments” tables to see how Eloquent models work in action. Each post will have a unique id, a title, the post itself (body) and timestamps of post creation and modification. Each comment will have a unique id, an id of the post that this comment is posted under, the comment itself (body) and the timestamps. The diagram in figure 7.4 illustrates the table structure of these two tables:

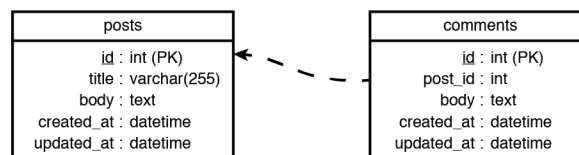


Figure 7.4 Table structure for posts and comments tables

The attributes of an Eloquent model represent the columns of the table that the model corresponds to. Operating with the values of model’s attributes is the same as working with the values in table’s columns. For example if you wanted to retrieve the title and the contents of a post with ID of “1” from the “posts” table, you would access the “title” and “body” attributes on a Post model as shown in figure 7.5 below:

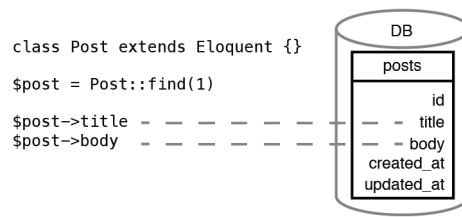


Figure 7.5 Model's attributes correspond to the table's columns

This elegant way of representing data could greatly simplify writing and maintaining application code that deals with data in a database. With the tables for comments and posts in place and models for them defined in the “app/models” folder we’ll be able to look into the features of Eloquent and learn more about this powerful ORM. Let’s start by looking at simple operations like inserting, retrieving, updating and deleting records.

7.2.1 Inserting records

Imagine that you wanted to create a new blog post as a new row in the “posts” table. You would like to set the “title” column to “My first post” and the “body” column to “This post is created with Eloquent”. To achieve this with Eloquent you would create a new instance of the Post class, assign the values to its attributes and call Eloquent’s “save()” function to insert a record in the DB. Let’s do that by creating a simple route and placing functionality there as shown in listing 7.2 below:

Listing 7.2 Creating a new record in DB with Eloquent model

```

// Define a route that the user needs to navigate to in order to create a new blog post en\
try
Route::get('posts/new', function(){
    // Create a new instance of the Post model
    $post = new Post;
    // Assign values to model's attributes
    $post->title = "My first post";
    // Assign values to model's attributes
    $post->body = "This post is created with Eloquent";
    // Insert the record in the DB
    $post->save();
    // Display the new record containing the blog post
    return $post;
});

```

Navigating to a URL “/posts/new” relative to the application’s root will create a new record in the “posts” table by using Eloquent ORM. When a model calls any methods of Eloquent, behind the scenes Laravel converts the object and operations on the model into SQL query that is then executed on the database and returns an object that was just created (figure 7.6):

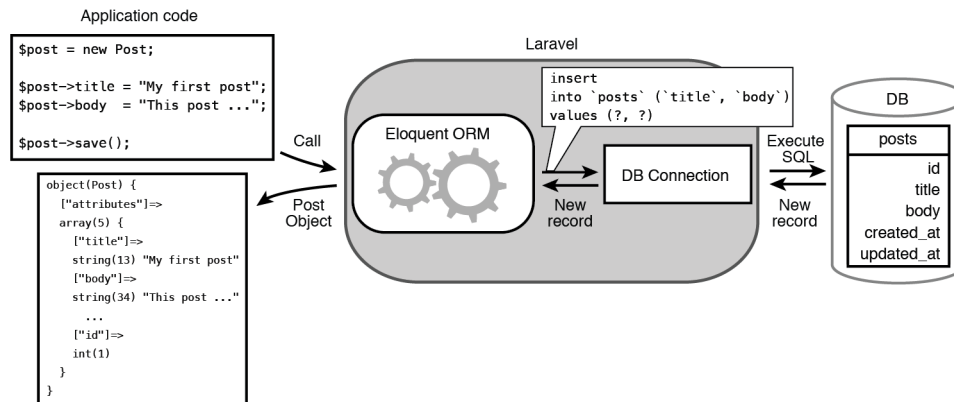


Figure 7.6 Creating a new record with Eloquent

How is this different from using Query Builder to create a new record? By using Eloquent you did not have to specify which table the record will be inserted into. The creation of the new record was done completely by operating with an object instead of running any database operations.

Now using the same concept let's insert a few comments that will be assigned to this new blog post by having "post_id" that matches the "id" of the post we just created, "1". We'll place the code that creates two comments inside of a route for "/comments/new" URL relative to application's root (listing 7.3):

Listing 7.3 Inserting comments

```

Route::get('comments/new', function()
{
    $firstComment = new Comment;
    $firstComment->post_id = 1;
    $firstComment->body = "My first comment";
    $firstComment->save();

    $secondComment = new Comment;
    $secondComment->post_id = 1;
    $secondComment->body = "My second comment";
    $secondComment->save();

    return 'Comments have been inserted';
});
  
```

Now if you navigate to "comments/new" URL in your browser you should see a message notifying that comments have been inserted and if you check your local database, you should see the new entries in "posts" and "comments" tables. Great! We can now explore other features of Eloquent. When the records are inserted in the database you can access them in the same way as you would retrieve them with the Query Builder.

7.2.2 Retrieving records

You can use any operators of the Query Builder on Eloquent models to retrieve or manipulate data in the database. If you remember from the previous chapter, Query Builder provides the following operators that retrieve records from a database:

- `find($id)`
- `get()`

Eloquent has two more operators that were not available in Query Builder:

- `all()`
- `first()`

Let's take a look at using these four operators to retrieve records from the database.

7.2.2.1 Retrieving specific record by using query builder's operator `find()`

As with Query Builder, you can retrieve a record from a table that corresponds to an Eloquent model by passing a value of record's primary key to operator `find()` executed on the model. By convention, the name of the primary key in the table is "id".



Note

Operator "find()" uses "id" as the default primary key of the table, if you would like to use a different name of the primary key, refer to the section titled "Overriding Conventions"

Executing this operator will return an object containing columns of the table represented as keys of an array and rows as values of the array. For example to retrieve a comment with "id" of 2 as an object, we can use `find()` on the "Comment" model (listing 7.4):

Listing 7.4 Retrieving a single record of Eloquent's model using operator "find"

```
// Retrieve a record with primary key (id) equal to "2"
$comment = Comment::find(2);

// If there is a comment with ID "2", The $comment variable will contain:
object(Comment)#137 (20) {
  ["attributes":protected]=>
  array(5) {
    ["id"]=>
    string(1) "2"
    ["post_id"]=>
    string(1) "1"
    ["body"]=>
    string(17) "My second comment"
    ...
  }
  ...
}
```

This operator is intended to serve as a shortcut method and executing it is the same as executing the following query, where `$id` is the value of the primary key of the record:

```
1 Comment::where('id', $id)->get();
```

7.2.2.2 Retrieving first record by using eloquent's operator first()

Like operator `find()`, Eloquent's operator `first()` will return a single record. Only in this case it will be the first record that matches some filtering criterion. This is useful when there could be a lot of records potentially retrieved from the table but you want only the first one that matches a criterion. For example if you wanted to retrieve the first record from the “comments” table that contains the word “comment” you could use this operator like the code in listing 7.5 shows:

Listing 7.5 Retrieving the first record using Eloquent's operator “first()”

```
// Define a route that the user needs to navigate to in order to see the first comment mat\
ching a criterion
Route::get('comments/first', function()
{
    // Retrieve the first comment that has word “comment” in it's body
    $comment = Comment::where('body', 'like', '%comment%')->first();
    // Show the retrieved object to the user
    return var_dump($comment);
});
```



Note

If operator `first()` is the only operator executed on the model, for example `Comment::first()`, it will retrieve the first record in the table that the model represents

Now that you know how to retrieve a single record using Query Builder and Eloquent operators let's take a look at retrieval of all records from a table that corresponds to a model.

7.2.2.3 Retrieving all records from a table using eloquent's operator all()

To retrieve all records from a table you can use Eloquent's operator `all()`. Execute this operator on a model to get all records from the table corresponding to the model. Code in listing 7.5 shows a definition of a route that will show all comments stored in the database as an array of objects:

Listing 7.5 Retrieving all records of a table

```
// Define a route that will show the comments
Route::get('comments', function()
{
    // Retrieve all comments from the “comments” table
    $comments = Comment::all();
    // Display the comments as an object containing all comments
    return var_dump($comments);
});
```



Note

Calling this operator will always retrieve all records of the model, even if there are other operators in the query meant to filter records

7.2.2.4 Using operator get()

The operator `get()` is special and has multiple uses throughout Eloquent and Query Builder. Depending on its position in the query chain and parameters passed into the operator, it can be used for:

- retrieving all records corresponding to the model
- executing a query that has operators for filtering and grouping records in the model
- retrieving specific columns of records corresponding to the model

If no other operators precede it in the query chain, using operator `get()` will retrieve all records of a model. For example, using this operator on a model to retrieve all records of model 'Comment' would look like code in listing 7.6:

Listing 7.6 Retrieving all records of a table using operator `get()`

```
// Define a route that will show the comments
Route::get('comments', function()
{
    // Retrieve all comments from the "comments" table using operator get() without
    // any other operators preceding it
    $comments = Comment::get();
    // Display the comments as an object containing all comments
    return var_dump($comments);
});
```

You can use this operator to retrieve a result of filtering and grouping of records by placing it at the end of the query, as you did with Query Builder in the previous chapter. By placing it at the end of the query you can take advantage of query chaining and even mix Eloquent's features with Query Builder operators. For example if you wanted to retrieve all blog posts that contain the word "Laravel" in their title you would append operator `get()` at the end of the filtering query like in listing 7.7 below:

Listing 7.7 Retrieving records using operator `get()`

```
// Define a route that will show the results of executing the query
Route::get('published', function()
{
    // User operator get() to retrieve posts from the "posts" table that have
    // "Laravel" in their title
    $posts = Post::where('title', 'like', '%Laravel%')->get();
    // Display the posts as an object containing posts matching the criterion
    return var_dump($posts);
});
```

You are not limited to having just one other operator, chaining two, three or a dozen operators and ending them with `get()` is allowed.

If you would like to retrieve specific columns of the table corresponding to the model, you can pass the names of those columns as an argument of operator `get()`. For example, retrieving only the "title" and "body" columns of all blog posts could be possible by providing those two columns as an array passed to `get()`, as shown in listing 7.8 below:

Listing 7.8 Retrieving specific columns of a table using operator get()

```
// Define a route that will show the results of executing the query
```

```
Route::get('posts', function()
```

```
{
```

```
    // Retrieve only "title" and "body" columns of all posts
```

```
    $posts = Post::get(array('title', 'body'));

```

```
    // Display the retrieved object in the browser
```

```
    return var_dump($posts);

```

```
});
```

```
// The $posts variable will now contain a collection of posts
```

```
object(Illuminate\Database\Eloquent\Collection)#149 (1) {
```

```
    ["items":protected]=>
```

```
    array(1) {
```

```
        [0]=>
```

```
        object(Post)#137 (20) {
```

```
            ...
```

```
            ["attributes":protected]=>
```

```
            array(2) {
```

```
                ["title"]=>
```

```
                string(13) "My first post"
```

```
                ["body"]=>
```

```
                string(34) "This post is created with Eloquent"
```

```
            }
```

```
            ...
```

```
        }
```

```
    }
```

```
}
```

Using only the necessary columns of the data can improve application's performance by conserving the amount of memory it is using. Specifying only a few columns of the table via attributes of operator get() is also useful when you are building an API and want to be flexible about what data the API exposes.

Now that you know how to create and retrieve data, let's take a look at updating data with Eloquent.

7.2.3 Updating records

Updating records with Eloquent is as easy as updating model's attributes and calling operator save() on the model. This is different from using Query Builder because with Eloquent you are working with an object that represents the table and not with raw table-columns. Updating records with Eloquent is done in the following sequence:

1. Retrieve the record that belongs to the model by using find(), get() or first() operators on the model
2. Assign new values to model's attributes
3. Call operator save() on the model to save the changes in the database

For example if you want to update a single record in the “posts” table that you retrieved with either `find()` or `get()` operators, assign new values to the instance of the `Post` object and call `save()` on this object. Code in listing 7.8 shows a definition of a route that will update the title of a record in “posts” table with ID of “1” when the user navigates to it in the browser:

Listing 7.8 Updating a single record

```
// Define a route that will show the results of executing the query
Route::get('posts/update', function()
{
    // Retrieve a record from “posts” table with id of 1
    $post = Post::find(1);
    // Display the current title of the record
    var_dump($post->title); // "My first title"
    // Update the title attribute of the model
    $post->title = "Updated title";
    // Save the record with updated attribute to the DB
    $post->save();
    // Display the updated title of the record
    return var_dump($post->title); // "Updated title"
});
```

Updating a set of records

Eloquent does not allow updating a single column of a set of records. To update many records of the same model at once you would have to use Query Builder’s `update()` operator or loop through

7.2.4 Deleting records

Coming soon, please stay tuned

7.3 Using relationships

- One-to-One
- One-to-Many
- Many-to-Many
- Polymorphic
- And more

7.3.1 One to one relationships

Coming soon, please stay tuned

7.3.2 One to Many relationships

Coming soon, please stay tuned

7.3.3 Many to Many relationships

Coming soon, please stay tuned

7.3.4 Polymorphic relationships

Coming soon, please stay tuned

7.3.5 Eager loading relations

Coming soon, please stay tuned

7.4 Filtering data

Coming soon, please stay tuned

7.4.1 Using query builder

Coming soon, please stay tuned

7.4.2 Using query scopes

Coming soon, please stay tuned

7.5 Working with retrieved data

Coming soon, please stay tuned

7.5.1 Using collections to iterate through data

Coming soon, please stay tuned

7.5.2 Converting results

Coming soon, please stay tuned

7.5.3 Displaying data

Coming soon, please stay tuned

7.6 Overriding conventions

Coming soon, please stay tuned

7.7 Summary

Working together with Query Builder, Eloquent seamlessly works with the data stored in the database. Laravel's unique ORM features many features that improve the way you can work with any amount of sophisticated data. Almost any imaginable DB operations are provided out of the box and enable the developer to focus on application's logic instead of figuring out complicated SQL statements. For those cases when Laravel's built in features are not enough, Laravel is flexible and allows using plain SQL to get the data to behave the way you want it to.