# Getting Started with Laravel 4

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

**Raphaël Saunier**

[PACKT] open source*
PUBLISHING    community experience distilled

# Getting Started with Laravel 4

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

**Raphaël Saunier**

# Getting Started with Laravel 4

# Credits

**Author**
Raphaël Saunier

**Reviewers**
Fabio Alessandro Locati

Pavel Tkachenko

**Acquisition Editors**
Akram Hussain

Llewellyn Rozario

**Commissioning Editor**
Poonam Jain

**Technical Editors**
Ritika Singh

Nachiket Vartak

**Copy Editors**
Sarang Chari

Gladson Monteiro

Adithi Shetty

**Project Coordinator**
Michelle Quadros

**Proofreader**
Lucy Rowland

**Indexer**
Priya Subramani

**Graphics**
Ronak Dhruv

**Production Coordinator**
Shantanu Zagade

**Cover Work**
Shantanu Zagade

# About the Author

**Raphaël Saunier** works as a full-stack Web Developer for Information Architects in Zürich, Switzerland. He holds a degree in Information Management for Business from University College London.

He is always looking for excuses to learn and play with new technologies, tools, and techniques. He is also able to make pragmatic decisions that take into account the strengths and weaknesses of the many well-established tools at the disposal of web developers.

A strong advocate of Laravel, Ember.js, Vim, and PostgreSQL when he is among developers, he is also passionate about teaching programming and general computer literacy to people of all ages.

# About the Reviewers

**Fabio Alessandro Locati** is an Italian IT external consultant. His main areas of expertise are Linux, networking, security, data centers, and web applications. With nearly 10 years of work in the field, he has experienced a lot of different IT roles, technologies, and languages. Fabio has worked in many different companies, from single-man companies up to huge companies such as Tech Data. This has allowed him to see the various technologies from different points of view, making him able to develop critical thinking and to understand if a technology is the right one in a very short time. As he is always on the lookout for better technologies, he always tries the new technologies to see the advantages over the old ones. For web development, he often uses PHP with Laravel due to its power and simplicity since he discovered it in the early part of 2012. Fabio has used Laravel for public websites as well as for intranet applications.

> I'd like to thank my father who introduced me to computer science before I was able to even write, and to my whole family, who have always been supportive.

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

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

# www.PacktPub.com

## Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

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

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



`http://PacktLib.PacktPub.com`

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

## Why Subscribe?

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

## Free Access for Packt account holders

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

# Table of Contents

# Preface

This book aims to bring you up to speed with the Laravel PHP framework. It introduces the main concepts that you need to know in order to get started and build your first web applications with Laravel 4.1 and later.

Mastering a new framework, such as Laravel, can be a challenging but very rewarding experience. Not only do they introduce new ways of approaching problems, frameworks can also challenge your prior knowledge of a particular area and show you more efficient ways of achieving certain tasks. Above all, their aim is to make you more productive and help you produce better code.

In the learning process, the quality of the documentation and the availability of learning material are the decisive factors. Laravel is fortunate enough to have a vibrant community that actively improves the official documentation and produces a large number of resources. However, if you are a complete beginner, this wealth of information might be somewhat overwhelming and, as a result, you might not know where to start. This book will walk you through the different steps involved in creating a complete web application and also present the different components bundled with Laravel. After reading this book, you will be well-equipped to read any part of the documentation or a tutorial on a particular component without feeling lost.

## What this book covers

*Chapter 1*, *Meeting Laravel*, will introduce the main concepts used by Laravel, its key features, and the default structure of a Laravel project.

*Chapter 2*, *Composer All Over*, will enable you to install and use the Composer dependency manager to download and install Laravel and third-party packages.

*Chapter 3*, *Your First Application*, will walk you through the different steps involved in creating an application that interacts with a database.

*Chapter 4*, *Authentication and Security*, will show you how to add the user authentication feature to your application. It will also cover the different security considerations to bear in mind when developing applications with Laravel.

*Chapter 5*, *Testing – It's Easier Than You Think*, will demonstrate how to write and run tests with PHPUnit, and will look at the different test helper methods that are bundled with Laravel.

*Chapter 6*, *A Command-line Companion Called Artisan*, will introduce you to the use of Artisan commands (Artisan is Laravel's command-line utility) to speed up your workflow and write custom command-line scripts.

*Chapter 7*, *Architecting Ambitious Applications*, will give us the opportunity to take a second look at the components that were used in the previous chapters, and to uncover their more advanced capabilities.

*Appendix*, *An Arsenal of Tools*, presents the different tools and helpers that you get for free when installing Laravel so that you do not find yourself rewriting the functionality that already exists in the framework.

# What you need for this book

In order to run the code samples in this book, you will need an installation of PHP 5.3.7 or later compiled with mcrypt support on Mac OS X, Linux, or Windows. PHP is available as a standalone installation, but you can also use a local server such as XAMPP or EasyPHP, on Windows, or MAMP on Mac OS X.

> Although Mac OS X does ship with a version of PHP, it is not compiled with mcrypt. You will either have to install a more recent version with a tool such as Homebrew or use the bundled binary with MAMP.

All the code examples use a file-based SQLite database, but you are more than welcome to use PostgreSQL or MySQL if you have either of them installed on your system.

You will of course need a code editor, such as Vim, Sublime Text, or TextMate, to create and edit the source file. If you are uncomfortable using the SQLite, MySQL, or PostgreSQL command-line utilities, you may use a graphical database administration interface, such as Sequel Pro or phpMyAdmin, although this is not strictly necessary.

The installation of Laravel and many other packages is done using the Composer dependency manager, and is covered in detail in *Chapter 2*, *Composer All Over*.

# Who this book is for

No prior knowledge of Laravel or any other modern web application framework is assumed. If you already know your way around Laravel, you may want to consider acquiring a different book, as a significant portion of this book deals with the basics.

This book is therefore ideal for web developers with prior experience of the PHP programming language—or any C-like languages such as JavaScript, Perl, or Java—along with some understanding of the basic OOP concepts.

Any experience with MVC frameworks, such as ASP.NET MVC or Ruby on Rails, will certainly be beneficial but is not required. Lastly, some familiarity with the command-line interfaces will also help, but is not essential.

# Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "They are also known as `closures` and were introduced in PHP 5.3."

A block of code is set as follows:

```
Route::get('hello/{name}', function($name){
  return "Hello " . $name;
});
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
Route::get('hello/{name}', function($name){
  return "Hello " . $name;
})->where('name', '[a-zA-Z]*');
```

Any command-line input or output is written as follows:

```
$ php artisan routes
```

When the command-line input is specific to Windows, it is written as follows:

```
> php artisan routes
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking on the **Next** button moves you to the next screen.".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

# Reader feedback

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

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

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

# Customer support

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

# Downloading the example code

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

# Errata

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

# Piracy

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

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

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

# Questions

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

# 1
# Meeting Laravel

In the jungle of PHP frameworks, the latest newcomer, Laravel, has been getting more and more attention recently. On many discussion forums, it has even dethroned CodeIgniter and Symfony as the number one recommended framework. What is it about this framework that makes both young and seasoned developers rave about it?

In this chapter, we will cover the following topics:

- How web application frameworks help to increase productivity
- The fundamental concepts and the key features of Laravel
- The general structure and conventions of a Laravel application
- General advice if this is the first time you are working with a Model-View-Controller (MVC) framework
- Migration tips for users of the previous version of Laravel

We will look at its key features and how they have made Laravel an indispensable tool for many web developers. We will present the limitations of PHP, especially when it is used without a modern framework, and how Laravel helps you to overcome those shortcomings and write more robust, and more structured, applications. Then, we will take a closer look at the anatomy of a Laravel application and present the different features of PHP as well as the third-party packages it leverages. After reading this chapter, you will have all the conceptual knowledge that is required to get started and build your first application.

# The need for frameworks

Of all the server-side programming languages, PHP undoubtedly has the weakest entry barriers. It is almost always installed by default on even the cheapest web hosts, and it is also extremely easy to set up on any desktop operating system. For newcomers who have some experience with HTML, the concepts of variables, inline conditions, and `include` statements are easy to grasp. PHP also provides many commonly used functions that one might need when developing a website. All of this contributes to what some refer to as the *immediacy* of PHP. However, this instant gratification comes at a cost. It gives a false sense of productivity to beginners, who almost inevitably end up with unnecessarily complex and tangled code as they add more features to their site. This is mainly because PHP, out of the box, does not do much to encourage the separation of concerns.

# The limitations of homemade tools

If you already have a few PHP projects under your belt, but have not used a web application framework before, then you will probably have your personal collection of commonly used functions or classes that you have amassed from one project to the next. These utilities help you solve recurring problems, such as sanitizing database calls, authenticating users, and including pages dynamically. You might also have a predefined directory structure where these classes and the rest of your application code reside. However, all of this will exist in complete isolation; you would be solely responsible for the maintenance, inclusion of new features, and documentation. This can be a tedious and time-consuming task. Not to mention that if you were to collaborate with other developers on the project, they would first have to get acquainted with the way you build applications.

# Laravel to the rescue

This is exactly where a web application framework such as Laravel comes to the rescue. Laravel re-uses and assembles existing components to provide you with a cohesive layer upon which to build your web applications in a more structured and pragmatic way. Drawing inspiration from popular frameworks written in both PHP and other programming languages, Laravel offers a robust set of tools and an application architecture that incorporates many of the best features of CodeIgniter, Yii, ASP.NET MVC, Ruby on Rails, and Sinatra.

If you have already used one of those tools or a different framework that implements the **Model-View-Controller** (**MVC**) paradigm, you will find it very easy to get started with Laravel 4.

# A new approach to developing PHP applications

It is a great time to discover (or fall back in love with) PHP. Over the years, the language has earned itself a bad reputation amongst developers who were forced to work on and maintain badly coded applications. Moreover, at the language level, PHP is also notorious for its naming inconsistencies and questionable design decisions regarding its syntax. As a consequence, there has been an exodus to more credible frameworks written in **Ruby** and **Python**. Since these languages were nowhere as feature-rich for the Web as PHP, the creators of Ruby on Rails and Django, for instance, had to recreate some essential building blocks, such as classes, to represent HTTP requests and responses and were, therefore, able to avoid some of the mistakes that PHP had made before them. From the start, these frameworks also forced the developer to adhere to a predefined application architecture.

# A more robust HTTP foundation

A few years on , these ideas have found their way back into PHP. The **Symfony** project has adopted these principles to recreate a more solid, flexible, and testable HTTP foundation for PHP applications. Along with the latest version of Drupal and phpBB, Laravel is one of the many open source projects that use this foundation together with several other components that form the Symfony framework.

Laravel does not just rely on and extend Symfony components, it also depends on a variety of other popular libraries, such as **SwiftMailer** for more straightforward e-mailing, **Carbon** for more expressive date and time handling, **Doctrine** for its inflector and database abstraction tools, and a handful of other tools to handle logging, class loading, and error reporting. In short, rather than trying to do everything itself, Laravel stands on the shoulders of giants.

# Embracing PHP

Laravel requires a relatively recent version of PHP, 5.3.7, released in August 2011. This version provides some nifty features that you might not be aware of if you have been working with earlier versions of PHP, or if you're completely new to the language. In this book, and when reading code examples for Laravel applications online, you will encounter some of these new features. For this reason, we will quickly have a look at them to make sure they don't throw you off!

- **Namespaces**: It is used extensively in languages such as Java and C# and helps you to avoid name collisions that happen when the same function name is used by two completely different libraries. Namespaces are separated by backslashes, and this is mirrored by the directory structure, with the only difference being the use of slashes on Unix systems in accordance with the **PSR-0** conventions. They are declared at the top of the file as `<?php namespace Illuminate\Database\Eloquent`. To specify the namespaces in which PHP should look for classes, we insert `use` followed by the "namespaced" class, for example, `use Illuminate\Database\Eloquent\Model;`.

- **Interfaces**: They are also known as **Contracts** and are a way of defining the methods that a class should provide, if it implements that interface. Interfaces do not contain any implementation details; they are merely contracts. So, for instance, if a class implements `JsonableInterface`, it needs to have a `toJson()` method.

- **Anonymous functions**: They are also known as `closures` and were introduced in PHP 5.3. Somewhat reminiscent of JavaScript, they help you produce shorter code, and you will use them extensively when building Laravel applications to define routes, events, filters, and in many other instances. The following is an example of an anonymous function attached to a route: `Route::get('hi', function() { return 'hi'; });`.

- **Overloading**: Also called `dynamic` or `magic` methods, they allow you to call methods such as `whereUsernameOrEmail($name, $email)` that were not previously defined in a class. These calls are handled by the `__call()` method, which then tries to parse the name to execute one or more known methods. In this case `->where('username', $username)->orWhere('email', $email)`.

- **Shorter array syntax**: Since PHP 5.4, a shorter array syntax has been introduced. Instead of writing `array('primes' =>array(1,3,5,7))`, it is now possible to write `['primes'=>[1,3,5,7]]`. Although we will use the old syntax in this book, you will probably come across the new syntax on the Web. If your server supports PHP 5.4, there is no reason not to use them.

# Laravel's main features and sources of inspiration

Let us now look at what you get when you start a project with Laravel and how these features can help you boost your productivity:

- **Modularity**: Laravel was built on top of over 20 different libraries and is itself split up into individual modules. Tightly integrated with **Composer Dependency Manager**, it can be updated with ease.

- **Testability**: Built from the ground up to ease testing, Laravel ships with several helpers that let you visit routes from your tests, crawl the resulting HTML, ensure that methods are called on certain classes, and even impersonate authenticated users.

- **Routing**: Laravel gives you a lot of flexibility when you define the routes of your application. For example, you may manually bind a simple anonymous function to a route with an HTTP verb, such as GET, POST, PUT, or DELETE. This feature is inspired by micro-frameworks, such as **Sinatra** (Ruby) and **Silex** (PHP). Moreover, it is possible to attach filter functions that are executed on particular routes.

- **Configuration management**: More often than not, your application will be running in different environments, which means that the database or e-mail server credentials settings or the displaying of error messages will be different when your app is running on a local development server than when it is running on a production server. Laravel lets you define settings for each environment and then automatically selects the right settings depending on where the app is running.

- **Query builder and ORM**: Laravel ships with a fluent query builder, which lets you issue database queries with a PHP syntax where you simply chain methods instead of writing SQL. In addition to this, it provides you with an **Object relational mapper** (**ORM**) and ActiveRecord implementation, called Eloquent, which is similar to what you would find in Ruby on Rails to help you define interconnected models. Both the query builder and the ORM are compatible with different databases, such as PostgreSQL, SQLite, MySQL, and SQL Server.

- **Schema builder, migrations, and seeding**: Also inspired by Rails, these features allow you to define your database schema with PHP code and keep track of any changes with the help of database migrations. A migration is a simple way of describing a schema change and how to revert to it. Seeding allows you to populate selected tables of your database, for example, after running a migration.

**[ 11 ]**

- **Template engine**: Partly inspired by the Razor template language in ASP. NET MVC, Laravel ships with **Blade**, a lightweight template language with which you can create hierarchical layouts with predefined blocks where dynamic content is injected.

- **E-mailing**: With its `Mail` class, which wraps the popular `SwiftMailer` library, Laravel makes it very easy to send an e-mail, even with rich content and attachments, from your application.

- **Authentication**: Since user authentication is such a common feature in web applications, Laravel provides you with the tools to register, authenticate, and even send password reminders to users.

- **Redis**: It is an in-memory key-value store that has a reputation for being extremely fast. If you give Laravel a `Redis` instance that it can connect to, it can use it as a session and general-purpose cache and also give you the possibility to interact with it directly.

- **Queues**: Laravel integrates with several queue services, such as Amazon SQS and IronMQ, to allow you to delay resource-intensive tasks, such as the e-mailing of a large number of users, and run them in the background rather than keep the user waiting for the task to complete.

# Expressiveness and simplicity

At the heart of Laravel's philosophy is simplicity and expressiveness. This means that particular attention has been given to the naming of classes to effectively convey their actions in plain English. Consider the following code example:

```php
<?php

Route::get('area/{id}', function($id){
  if(51 == $area and !Auth::check()) {
    return Redirect::guest('login');
  } else {
    return "Welcome to Area " . $area;
  }
})->where('id, '[0-9]+');
```

**Downloading the example code**

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

Even though we have not even installed Laravel or presented its routing functions yet, you will probably have a rough idea of what this snippet of code does. Expressive code is more readable for someone new to a project, and it is probably also easier to learn and remember for you.

## Prettifying PHP

The authors of Laravel have gone on to apply these principles to existing functions as well. A prime example is the `File` class, which was created to make file manipulations:

- *more expressive*: To find out when a file was last deleted, use `File::lastModified($path)` instead of `filemtime(realpath($path))` To delete a file, use `File::delete($path)` instead of `@unlink($path)`, which is the standard PHP equivalent.

- *more consistent*: Some of the original file manipulation functions of PHP are prefixed with `file_`, while others just start with `file`; some are abbreviated and other are not.

- *more testable*: Many of the original functions can be tricky to use in tests due to the exceptions they throw and also because they are more difficult to mock.

- *more feature complete*: This is achieved by adding functions that did not exist before, such as `File::copyDirectory($directory, $destination)`.

There are very rare instances where expressiveness is sacrificed for brevity. This is the case for commonly used shortcut functions, such as `e()`, that escape HTML entities or `dd()` and which you can use to halt the execution of the script and dump the contents of one or more variables.

## Responsibilities, naming, and conventions

At the beginning of this chapter, we pointed out that one of the main issues with standard PHP applications was the lack of a clear separation of concerns; business logic becomes entangled with the presentation and data tier. Like many other frameworks that favor convention over configuration, Laravel gives you a scaffolding with predefined places to put code in. To help you to eliminate trivial decisions, it expects you to name your variables, methods, or database table names in certain ways. It is, however, far less opinionated than a framework like Ruby on Rails and in areas like routing, where there is often more than one way to solve a problem.

We have also pointed out that Laravel is an MVC framework. Do not worry if you have not used this architecture pattern before, in a nutshell, this is what you need to know about MVC in order to be able to build your first Laravel applications.

- **Models**: Just think of the models as `entities` of your system. Very often, but not always, they correspond to tables in your database. As we will see, all that is required to define a model is to create a new class that extends the `Eloquent` class. While the class name is defined with a singular noun in `CamelCase`, the corresponding table at the database level will by convention have to be the `pluralsnake_case` version of that class name. Thanks to the inflection libraries it uses, `Eloquent` will know that a model called `VisitedCountry` corresponds to the `visited_countries` table in the database. Laravel will also expect the primary key to be called `id`, and by default it will look for the `created_at` and `edited_at` fields that it updates automatically. Like every other part of `Eloquent`, and in accordance with the convention over configuration paradigm, if the default behavior is not quite working for you, you can always choose to override it. Models also contain information about how they relate to other models. Using the Active Record terminology, it is possible to define the `belongsTo`, `hasMany`, and `belongsToMany` relationships.

- **Controllers or routes**: There are two types of controllers in Laravel, **standard** controllers and **resource** controllers. Their job is to make sense of the incoming requests and to send an appropriate response. Both adhere to slightly different conventions. Traditional controllers are similar to what you would find in frameworks such as CodeIgniter, where a `detail` action that takes one parameter and which lives in the `Projects` controller could, by convention, be reached at `/projects/detail/123`. Resource controllers, on the other hand, allow you to define the `RESTful` controllers that respond to the different HTTP verbs, such as `GET`, `POST`, `PUT`, and `DELETE`. Lastly, for smaller and simpler applications, it is possible to bypass controllers altogether and write the entire application logic in routes.

- **Views or Templates**: Views are responsible for displaying the data that the controller received from the model. They can be conveniently built using the Blade template language or simply using standard PHP. The file extension of the view, either `.blade.php` or simply `.php`, determines whether or not Laravel treats your view as a `Blade` template when it renders it.

The following diagram illustrates the interactions between all the constituent parts:
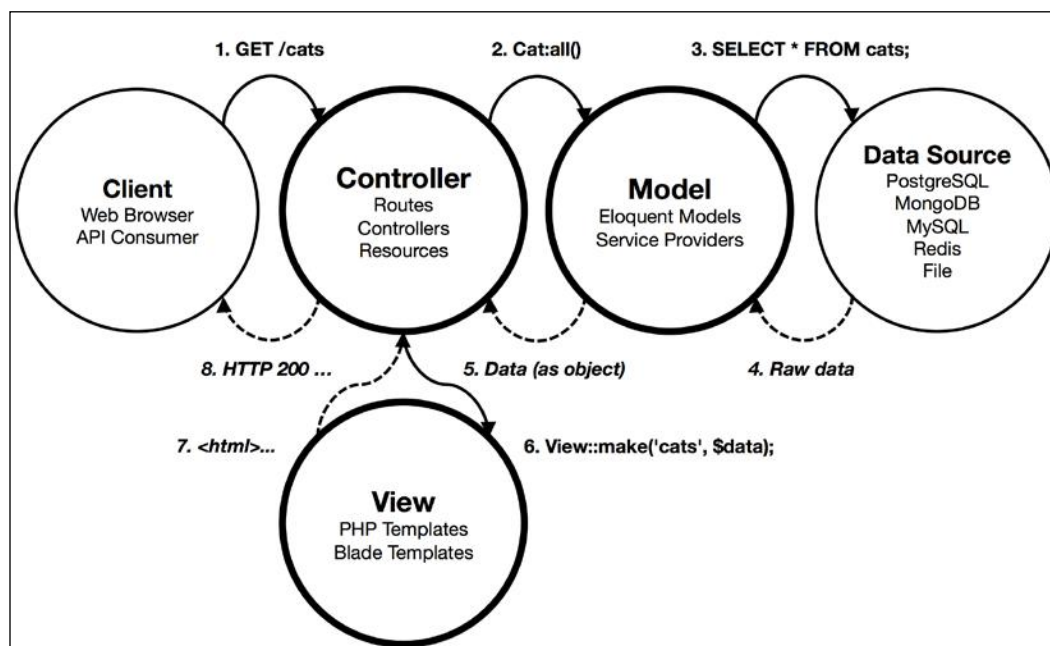


While it will still be possible to write unstructured code and go against the MVC paradigm and the framework's conventions, it will often involve more effort on the developer's part.

# Helping you become a better developer

Laravel's design decisions, and in particular, the way in which it inspires developers to write framework-agnostic code promises it a bright future. In addition to this, its community is probably one of its strongest assets; it is possible to get answers within minutes on forums, IRC, and Twitter.

However, frameworks come and go, and it is hard to predict when Laravel will lose its steam and be supplanted by a better or more popular framework. However, Laravel will not only make you more productive in the short term, it also has the potential to make you a better developer in the long run. By using it to build web applications, you will indirectly become more familiar with the following concepts, all of which are highly transferable to any other programming language or framework. These include the MVC paradigm and Object-oriented programming design patterns, the use of dependency managers, testing and dependency injection, and the power and limitations of ORMs and database migration.

It will also inspire you to write more expressive code with descriptive **DocBlock** comments that facilitate the generation of documentation as well as the future maintenance of the application.

# Structure of a Laravel application

In the next two chapters, we will be installing Laravel and creating our first application. Any new project already has a complete directory tree and even some placeholder files to get you up and running in very little time. This structure is a great starting point, but as we will see in the final chapter of this book, it is also customizable. Here is what it looks like:

```
./app/                      # Your Laravel application
  ./app/commands/           # - Command line scripts
  ./app/config/             # - Configuration files
  ./app/controllers/        # - Controllers
  ./app/database/           # - Database migrations and seeders
  ./app/lang/               # - Localisation variables
  ./app/models/             # - Classes used to represent entities
  ./app/start/              # - Startup scripts
  ./app/storage/            # - Cache and logs directory
  ./app/tests/              # - Test cases
  ./app/views/              # - Templates that are rendered to HTML
  ./app/filters.php         # - Filters executed before/after a request
  ./app/routes.php          # - URLs and actions

./bootstrap/                # Application bootstrapping scripts

./public/                   # Document root
  ./public/.htaccess        # - Sends incoming requests to index.php
  ./public/index.php        # - Starts Laravel application

./vendor/                   # Third-party dependencies installed
                              through Composer

./artisan*                  # Artisan command line utility

./composer.json             # Project dependencies

./phpunit.xml               # Test configuration file for PHPUnit

./server.php                # Local development server
```

Like the rest of Laravel, the naming is expressive, and it is easy to guess what each folder is for. On the first level, there are four directories, `app/`, `bootstrap/`, `public/`, and `vendor/`. All your server-side code will reside in the `app/` directory, inside which you will find the three directories that hold the files for the controllers, models, and views. We will explore the responsibilities of each directory further in the next chapters.

# The application container and request lifecycle

Whether you are a beginner in PHP or an experienced developer in a different language, it might not always be obvious how an HTTP request reaches a Laravel application. Indeed, the request lifecycle is fundamentally different from plain PHP scripts that are accessed directly by their URI (for example, `GET http://example.com/about-us.php`).

The `public/` directory is meant to act as the document root; in other words, the directory in which your web server starts looking after every incoming request. Once URL rewriting is properly set up, every request that does not match an existing file or directory hits the `/public/index.php` file. The job of this file is to register the `Composer` class autoloader, which tells PHP where to look for any classes that are called. It then bootstraps the application by setting its environment based on the host name and binding the different paths of your application. Once that is done, it simply instantiates a new application container, which is in turn responsible for dealing with the incoming request. This application container uses an HTTP verb and request URL (for example, `POST /comments`) and maps it to the correct controller action or route.

# Exploring Laravel

In this chapter, we are only covering the general mechanisms of how Laravel works, without looking at the detailed implementation examples. For the majority of developers who just want to get the job done, this is sufficient. Moreover, it is much easier to delve into the source code of Laravel once you have already built a few applications. Nevertheless, here are some answers to the questions that might crop up when exceptions are thrown or when you navigate through the source code. In doing so, you will come across some methods that are not documented in the official guide, and you might even be inspired to write better code.

Browsing the API (`http://laravel.com/api`) can be somewhat intimidating at first. But it is often the best way to understand how a particular method works under the hood. Here are a few tips:

- The `Illuminate` namespace does not refer to a third-party library. It is the namespace that the author of Laravel has chosen for the different modules that constitute Laravel. Every single one of them is meant to be reusable and used independently from the framework.

- When searching for a class definition, for example, `Auth`, in the source code or the API, you might bump into `Façade`, which hardly contains any helpful method and only acts as a proxy to the real class. This is because almost every dependency in Laravel is injected into the application container when it is instantiated.

- Most of the libraries that are included in the `vendor/` directory contain a *README* file, which details the functionality present in the library (for example, `vendor/nesbot/carbon/readme.md`).

# Moving from Version 3 to Version 4

The rise in popularity of Laravel started with its third version. Although most of the core features have been ported to version 4, if you have already written applications with Laravel 3, and you are reading this book to get up to speed with the changes and maybe migrate your app, here are the main changes that you need to be aware of:

- **Packages are the new bundles**: Laravel 3 had a thriving ecosystem of plug-ins, called bundles. The PHP community is trying to steer away from framework-specific packages since they complicate future maintenance, and release more generic packages instead. If you happen to stumble upon a Laravel bundle that seems to solve a problem, you are having in your applications, unfortunately it will not work with Laravel 4. To replace them, it now encourages the use of packages, many of which are framework-agnostic.

- **Composer all over**: The next chapter will explore this in more detail, but Laravel 4 uses `composer` to manage its various dependencies and keep them up to date.

- **New coding conventions**: As mentioned previously, Laravel 4 adheres to the `PSR-0` and `PSR-1` standards. The most notable change is the switch to `camelCase` methods and class names, where `User::where_email_and_active($email, true)` became `User::whereEmailAndActive($email,true)` and `User_Controller` became `UserController`.

- **Dependency injection**: Laravel 4 has been rewritten to heavily rely on dependency injection. This makes it easy to swap out entire classes and facilitates testing.

- **Documentation**: The new documentation has been simplified and the topics have been regrouped. Compared to the 3.0 documentation, it is slightly terser in some areas. The API, generated with **ApiGen**, also has a fresh look and is a joy to browse.

- **Renamed methods**: Other changes to consider, especially when migrating an existing application, is the renaming of certain methods `URL::to_route` was shortened to `URL::route`.

# Summary

In this chapter, we have introduced the main characteristics of Laravel 4 and how it can help you to write more structured applications while reducing the amount of boilerplate code. We have also explained the concepts and PHP features used by Laravel, and you should now be well equipped to get started and write your first application! In the next chapter we will learn how to install and use `Composer`, a dependency manager for PHP, which will install Laravel and its dependencies for you.

# 2
# Composer All Over

From the previous chapter, you now know that Laravel was built on top of several third-party packages. Rather than including these external dependencies in its own source code, Laravel uses **Composer**, a dependency manager, to download them and keep them up-to-date. Since it is a package itself, Laravel is also treated like a dependency is, therefore, very easy to install.

In this chapter, we will cover the following topics:

- The problems that dependency managers solve
- Installation instructions for Windows and Unix systems (Mac OS X, Linux)
- Creating a new Laravel project with Composer
- Finding and installing additional packages
- General advice for working with Composer

Strongly inspired by popular dependency managers, such as Bundler from the Ruby community, or npm, used by `node.js` and other JavaScript projects, Composer brings their features to the PHP world. However, by default it will not install packages globally; instead, it is meant to be used on a per-project basis. If you need to install PHP dependencies for your entire system, you should still use **PEAR**, the package manager that is bundled with PHP by default.

If you have not used one before, here are the main reasons to use a dependency manager in your workflow:

- It is quicker than searching for, downloading, and unzipping the different packages manually
- It helps you avoid version conflicts when upgrading individual dependencies
- The auto-loading of the different classes is done for you
- The discovery and selection of packages is greatly simplified thanks to a central repository

# Working with the command line

If you are just getting started with web development, you might not be completely familiar with the **command-line interface** (**CLI**). Working with a Composer, and later on with **Artisan**, Laravel's CLI utility, will require some interaction with it.

Here is how you can start it:

1. On Windows, look for the **Command Prompt** program. If you cannot find it, just click on **Start | Run** and type in `cmd.exe`.
2. On Mac OS X, it is called **Terminal** and it can be found inside `/Applications/Utilities`.
3. On Linux, depending on your distribution of Linux, it will be called Terminal or Konsole, but if you are running Linux, you are probably already familiar with it.

You do not need to have any advanced command-line skills to get through this book and build applications with Laravel. You will, however, need to be able to navigate to the right directory in your filesystem before running commands. To do this, just enter the `cd` command followed by the path to your code directory.

> On most systems you can also just enter `cd` followed by a space and then drag-and-drop the directory into the terminal.
> ```
> $ cd /path/to/your/code/directory
> ```

Or you can run the following command line on Windows:

```
> cd C:\path\to\your\code\directory
```

In the rest of this book, unless the example is specific to Windows, we will always use the `$` character to denote a shell command and use slashes as directory separators. Make sure you adapt the command accordingly if you are running Windows.

# How does Composer work?

Composer (`http://getcomposer.org/`) comes as a PHP executable, which is added to your PATH environment variable (that is, the list of locations that is searched when you run a command). When installed correctly, it can be executed in the command line from anywhere in your filesystem using the `composer` command. Your project, with its dependencies, is defined with a JSON file, called `composer.json`. Composer reads the contents of this file and connects to **Packagist**, (`https://packagist.org/`) an online repository, to resolve the different dependencies, recursively.

These dependencies are then downloaded to a local directory, in our case, `vendor/`, and the state of the dependencies is saved to a file called `composer.lock`. Composer also generates an `autoload.php` file at the root of the `vendor/` directory that wires up the auto-loading of classes when it is included in a PHP script (using `require 'vendor/autoload.php'`).

# Installation

Installing Composer on Unix or a Windows system is very straightforward thanks to its installer and installation scripts.

## Unix (Mac OS, Linux)

First of all, we need to make sure that the `php` executable can be called from the command line. To do so, simply open a new terminal window and enter:

```
$ php -v
```

This will show you the information related to the currently installed PHP version. If you get a **command not found** error or have a version that is inferior to the minimum requirement (which is, 5.3.7), refer to the distribution-specific installation guide at `http://php.net`. On Mac OS, whether you have installed PHP with **MAMP** or **Homebrew**, you will have to make sure that it is included and loaded first in your PATH variable.

Then, once you have made sure that PHP is reachable from the command line, to install Composer, enter the following command:

```
$ curl -sS https://getcomposer.org/installer | php
```

To make it available globally, just move it to a directory that is included in your PATH. If you get a permissions error, select a different directory for which you have write access or, if you can, prefix the command with `sudo`.

```
$ sudo mv composer.phar /usr/local/bin/composer
```

Finally, to ensure that it is installed correctly, open a new terminal window and enter the following:

```
$ composer
```

This will give you a list of all the available commands.

# Windows

To install Composer on Windows, and assuming that you already have a working version of PHP installed, simply head to `http://getcomposer.org/download/` and download the `Composer-Setup.exe` file.

The installer will ask you to locate the PHP executable. Common locations for `php.exe` include:

- Default location: `C:\PHP5\php.exe` or `C:\PHP\php.exe`
- If you are using XAMPP: `C:\xampp\php\php.exe`
- If you are using WAMP: `C:\wamp\bin\php\php5.x.x\php.exe`

If you cannot find it in these locations but you are certain that it is installed somewhere, simply use the Windows search to find `php.exe` on your hard drive.

The installer will then take care of the rest by installing Composer and adding the `php` and `composer` commands to your PATH.

To make sure that it is installed properly, open a new command prompt and enter the following two commands:

```
> php -v
> composer
```

Both commands should output the respective version information messages.

# Creating a new Laravel application

Once Composer is installed, it is extremely easy to create a new Laravel project. After navigating to the directory in which you want to start the project, simply run the following command line:

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

This will download the latest version of Laravel as well as its dependencies. Depending on your CPU and connection speed, this could take a few minutes. Once it is complete, you will find the complete directory structure that was presented in the previous chapter and you will be ready to start your first project.

If you are using Git for version control, this is a good time to run a `git init`; the root of the directory already contains a `.gitignore` file and the placeholder directories each have a `.gitkeep` file.

# Finding and installing new packages

Using the search on `http://packagist.org`, you can find packages to add common features, such as image manipulation or PDF generation to your application. Indicators of good packages beyond the number of downloads and the number of stars on GitHub are the quality of documentation, the test coverage, and the overall project activity. Before adding a new package, you can also browse the different versions of a package and its dependencies on Packagist:
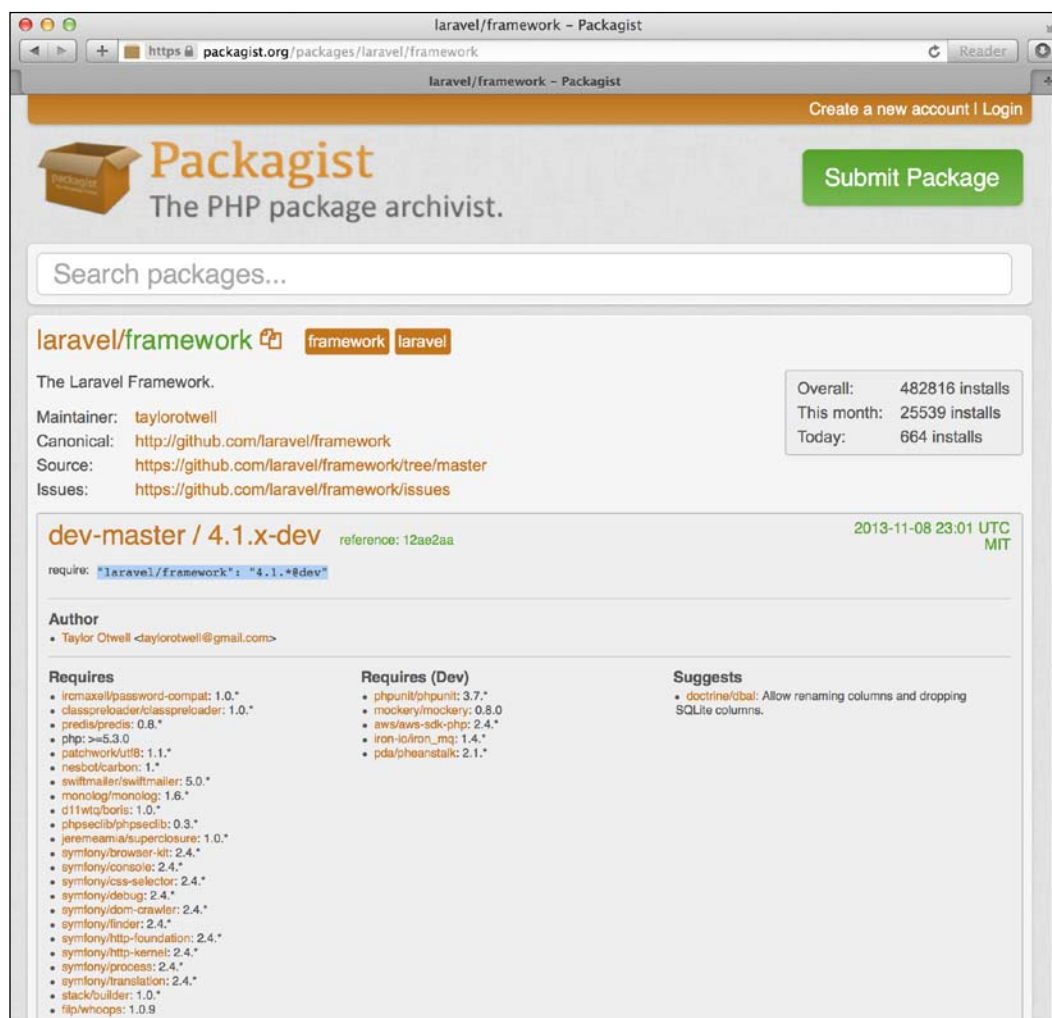


In development it is fine to use a `dev-master` branch, but in production, it is better to stick with a precise version number to avoid potential compatibility issues.

To install a package, open `composer.json` in a text editor and insert its name and the desired version in the require object:

```
"require": {
  "laravel/laravel": "v4.1.*",
  "intervention/image": "dev-master"
}
```

Since it is a JSON file, you need to be careful not to leave any trailing commas on the last line. To check if there are any errors in your `composer.json` file, you can use the `composer validate` command.

Then, simply run `composer install` and Composer will fetch the package and its dependencies. To update the `composer.lock` file and save the exact version numbers of the resolved dependencies, call `composer update`. If you deploy or distribute your application, the lock file allows everyone else to retrieve the exact same packages when they run `composer install`. The `update` command, on the other hand, will always check for the latest version of every package and if you run it in production, you risk running into compatibility issues.

# Additional advice

Before you go off and start writing your first Laravel application, here are some additional tips to work with Composer.

- Commit the `composer.lock` file to your VCS repository to bypass the dependency resolution and make sure that everyone you collaborate with works with the exact same versions of the dependencies.

- You are not meant to check the contents of the `vendor/` directory in version controller. It is already excluded in the `.gitignore` file. Including it would increase the size of your repository and commit messages. As long as there is a `composer.json` and `.lock` file, anyone who downloads your application will be able to resolve the dependencies.

- You are not meant to edit any files inside `vendor/` either, since these would be overwritten the next time you run `composer install`.

- Composer gives you two options, `--prefer-source` and `--prefer-dist`, when you install packages. The difference between these two options is that with the `dist` option, Composer will favor stable releases and avoid downloading the entire Git history if possible.

- The total size of the `vendor/` directory will be in the region of 25 MB with `--prefer-dist` and about three times of that with `--prefer-source`, since the complete Git history of each package is included. When you deploy a Laravel application, you are meant to run `composer install` on the server. If FTP deployment is your option, there are packages such as `barryvdh/ laravel-vendor-cleanup` that you can use to remove the non-essential files before uploading everything to your server.

- Composer's `diagnose` command and the verbosity flags (`-v|vv|vvv`) can help you identify common problems and will make it easier for people to help you on **IRC**, **Stack Overflow**, and forums.

# Summary

In this chapter we have explained the problems solved by dependency managers. We have installed Composer and created our first Laravel project. We have also learned about how to find and install packages and how to avoid common mistakes when working with Composer.

The next chapter is where the real fun begins! Now that you have a working installation of Composer, we will go through the different steps involved in creating a complete Laravel application.

# 3
# Your First Application

Having learned about the conventions and responsibilities in Laravel, and how to create a new project with Composer, you are now ready to build your first application!

In this chapter, you will use the concepts presented in the previous two chapters in a practical way and learn how to do the following:

- Sketch out the URLs and entities of your application
- Troubleshoot common issues when getting started
- Define the routes and their actions as well as the models and their relationships
- Prepare your database and learn how to interact with it using Eloquent
- Use the Blade template language to create hierarchical layouts

The first step in creating a web application is to identify and define its requirements. Then, once the main features have been formulated, we derive the main entities as well as the URL structure of the application. Having a well-defined set of requirements and URLs is also essential for other tasks such as testing; this will be covered later in the book.

A lot of new concepts are presented in this chapter. If you have trouble understanding something or if you are not quite sure where to place a particular snippet of code, you can download the annotated source code of the application on `http://packtpub.com/support`, which will help you to follow along.

# Sketching out the application

We are going to build a browsable database of cat profiles. Visitors will be able to create pages for their cats and fill in basic information such as the name, date of birth, and breed for each cat. This application will support the default **Create-Retrieve-Update-Delete** operations (**CRUD**). We will also create an overview page with the option to filter cats by breed. All of the security, authentication, and permission features are intentionally left out since they will be covered in the next chapter.

# Entities, relationships, and attributes

Firstly, we need to define the *entities* of our application. In broad terms, an entity is a thing (person, place, or object) about which the application should store data. From the requirements, we can extract the following entities and attributes:

- Cats, which have a numeric identifier, a name, a date of birth, and a breed
- Breeds, which only have an identifier and a name

This information will help us when defining the *database schema* that will store the entities, relationships, and attributes, as well as the *models*, which are the PHP classes that represent the objects in our database.

# The map of our application

We now need to think about the URL structure of our application. Having clean and expressive URLs has many benefits. On a usability level, the application will be easier to navigate and look less intimidating to the user. For frequent users, individual pages will be easier to remember or bookmark and, if they contain relevant keywords, they will often rank higher in search engine results.

To fulfill the initial set of requirements, we are going to need the following routes in our application:

| Method | Route | Description |
|--------|-------|-------------|
| GET | / | Index |
| GET | /cats | Overview page |
| GET | /cats/breeds/:name | Overview page for specific breed |
| GET | /cats/:id | Individual cat page |
| GET | /cats/create | Form to create a new cat page |

| Method | Route | Description |
| --- | --- | --- |
| POST | /cats | Handle creation of new cat page |
| GET | /cats/:id/edit | Form to edit existing cat page |
| PUT | /cats/:id | Handle updates to cat page |
| GET | /cats/:id/delete | Form to confirm deletion of page |
| DELETE | /cats/:id | Handle deletion of cat page |

We will shortly learn how Laravel helps us to turn this routing sketch into actual code. If you have written PHP applications without a framework, you can briefly reflect on how you would have implemented such a routing structure. To add some perspective, this is what the second to last URL could have looked like with a *traditional* PHP script (without URL rewriting): `/index.php?p=cats&id=1&_ action=delete&confirm=true`.

The preceding table can be prepared using a pen and paper, in a spreadsheet editor, or even in your favorite code editor using ASCII characters. In the initial development phases, this table of routes is an important prototyping tool that forces you to think about URLs first and helps you define and refine the structure of your application iteratively.

If you have worked with REST APIs, this kind of routing structure will look familiar to you. In RESTful terms, we have a `cats` resource that responds to the different HTTP verbs and provides an additional set of routes to display the necessary forms.

If, on the other hand, you have not worked with RESTful sites, the use of the `PUT` and `DELETE` HTTP methods might be new to you. Even though web browsers do not support these methods for standard HTTP requests, Laravel uses a technique that other frameworks such as Rails use, and emulates those methods by adding a `_method` input field to the forms. This way, they can be sent over a standard `POST` request and are then delegated to the correct route or controller method in the application.

Note also that none of the form submissions endpoints are handled with a `GET` method. This is primarily because they have side effects; a user could trigger the same action multiple times accidentally when using the browser history. Therefore, when they are called, these routes never display anything to the users. Instead, they redirect them after completing the action (for instance, `DELETE /cats/:id` will redirect the user to `GET /cats`).

# Starting the application

Now that we have the blueprints for the application, let's roll up our sleeves and start writing some code.

Start by opening a new terminal window and create a new project with Composer, as follows:

```
$ composer create-project laravel/laravel cats --prefer-dist
$ cd cats
```

Once Composer finishes downloading Laravel and resolving its dependencies, you will have a directory structure identical to the one presented in the first chapter.

# Using the built-in development server

To start the application, unless you are running an older version of PHP (5.3.*), you will not need a local server such as WAMP on Windows or MAMP on Mac OS since Laravel uses the built-in development server that is bundled with PHP 5.4 or later.

To start the development server, we use the following `artisan` command:

```
$ php artisan serve
```

Artisan is the command-line utility that ships with Laravel and its features will be covered in more detail in a future chapter.

Next, open your web browser and visit `http://localhost:8000`; you will be greeted with Laravel's welcome message.

If you get an error telling you that the `php` command does not exist or cannot be found, make sure that it is present in your PATH variable. If the command fails because you are running PHP 5.3 and you have no upgrade possibility, simply use your local development server (MAMP/WAMP) and set Apache's `DocumentRoot` to point to `cats-app/public/`.

# Writing the first routes

Let's start by writing the first two routes of our application inside `app/routes.php`. This file already contains some comments as well as a sample route. You can keep the comments but you must remove the existing route before adding the following routes:

```
Route::get('/', function(){
  return "All cats";
});

Route::get('cats/{id}', function($id){
  return "Cat #$id";
});
```

The first parameter of the `get` method is the URI pattern. When a pattern is matched, the closure function in the second parameter is executed with any parameters that were extracted from the pattern. Note that the slash prefix in the pattern is optional; however, you should not have any trailing slashes. You can make sure that your routes work by opening your web browser and visiting `http://localhost:8000/cats/123`.

If you are not using PHP's built-in development server and are getting a 404 error at this stage, make sure that Apache's `mod_rewrite` configuration is enabled and works correctly.

# Restricting the route parameters

In the pattern of the second route, `{id}` currently matches any string or number. To restrict it so that it only matches numbers, we can chain a `where` method to our route as follows:

```
Route::get('cats/{id}', function($id){
  return "Cat #$id";
})->where('id', '[0-9]+');
```

The `where` method takes two arguments: the first one is the name of the parameter and the second one is the regular expression pattern that it needs to match.

If you now try to visit an invalid URL, Laravel will throw a `NotFoundHttpException` and display a pretty and informative stack trace:



Do not be intimidated by the colors or the amount of information displayed; this is simply a list of functions that were executed before the error occurred. It also provides you with an overview of the data (that is, server and request data, `GET`/`POST` Data, Files, Cookies, and Session values) that the application receives and will therefore be indispensable when debugging an application.

# Catching the missing routes

Instead of displaying a detailed error message to your visitors, you can catch the "Not Found" exception and display a custom message by defining the following `missing` method for your application inside `app/start/global.php`:

```
App::missing(function($exception){
  return Response::make("Page not found", 404);
});
```

Here we are not merely returning a string, but a `Response` object with the message as its first parameter and an HTTP status code as the second parameter. In the first two routes that we wrote, Laravel automatically converted the string that we returned into a 200 OK HTTP response (for example, in the first route it is: `Response::make("All cats", 200)`). While the difference might not be obvious to the end users, the distinction between "404 Not Found" and "200 OK" is important for the search engines that crawl your site or when you are writing an API.

# Handling redirections

It is also possible to redirect visitors by returning a `Redirect` object from your routes. If for example, we wanted everyone to be redirected to `/cats` when they visit the application for the first time, we would write the following lines of code:

```
Route::get('/', function(){
  return Redirect::to('cats');
});

Route::get('cats', function(){
  return "All cats";
});
```

# Returning views

The most frequent object that you will return from your routes is the `View` object. Views receive data from a route (or controller) and inject it into a template, therefore, helping you to separate the business and presentation logic in your application.

To add your first view, simply create a file called `about.php` inside `app/views` and add the following content to it:

```
<h2>About this site</h2>
There are over <?php echo $number_of_cats; ?> cats on this site!
```

Then return the view using the `View::make` method with a variable, `$number_of_cats`:

```
Route::get('about', function(){
  return View::make('about')->with('number_of_cats', 9000);
});
```

Finally, visit `/about` in your browser to see the rendered view. This view was written with plain PHP; however, Laravel comes with a powerful template language called Blade, which will be introduced later in this chapter.

# Preparing the database

Before we can expand the functionality of our routes, we need to define the models of our application, prepare the necessary database schema, and populate the database with some initial data. To keep things simple and also show the flexibility of the Eloquent ORM, we are going to use SQLite, a lightweight file-based database.

To configure Laravel to use SQLite, open `app/config/database.php` and change the default database connection name from `mysql` to `sqlite`. Also make sure that the default database file, `app/database/production.sqlite`, exists.

# Creating the Eloquent models

The first and easiest step is to define the models with which our application is going to interact. At the beginning of this chapter, we identified two main entities, *cats* and *breeds*. Laravel ships with Eloquent, a powerful ORM that lets you define these entities, map them to their corresponding database tables, and interact with them using PHP methods rather than raw SQL. By convention, they are written in the singular form; a model named `Cat` will map to the `cats` table in the database, and a hypothetical `Mouse` model will map to the `mice`.

The `Cat` model, saved inside `app/models/Cat.php`, will have a `belongsTo` relationship with the Breed model, which is defined in the following code snippet:

```
class Cat extends Eloquent {
  protected $fillable = array('name','date_of_birth','breed_id');
  public function breed(){
    return $this->belongsTo('Breed');
  }
}
```

The `$fillable` array defines the list of fields that Laravel can fill by **mass assignment**, a convenient way to assign attributes to a model. By convention, the column that Laravel will use to find the related model has to be called `breed_id` in the database. The Breed model, `app/models/Breed.php` is defined with the inverse `hasMany` relationship as follows:

```
class Breed extends Eloquent {
  public $timestamps = false;
  public function cats(){
    return $this->hasMany('Cat');
  }
}
```

By default, Laravel expects a `created_at` and `updated_at` timestamp field in the database table. Since we are not interested in storing these timestamps with the breeds, we disable them in the model by setting the `$timestamps` property to `false`.

This is all the code that is required in our models for now. We will discover various other features of Eloquent as we progress in this book; however, in this chapter we will primarily use two methods: `all()` and `find()`. To illustrate their purpose, here are the SQL queries that they generate:

```
Breed::all()     => SELECT * FROM breeds;
Cat::find(1)     => SELECT * FROM cats WHERE id = 1;
```

In the views and controllers of our application, the properties of an Eloquent model can be retrieved with the `->` operator: `$cat->name`. The same goes for the properties of the related models, which are accessible as shown: `$cat->breed->name`. Behind the scenes, Eloquent will perform the necessary SQL joins.

# Building the database schema

Now that we have defined our models, we need to create the corresponding database schema. Thanks to Laravel's support for migrations and its powerful schema builder, you will not have to write any SQL code and you will also be able to keep track of any schema changes in a version control system. To create your first migration, open a new terminal window, and enter the following command:

**$ php artisan migrate:make add_cats_and_breeds_table**

This will create a new migration inside `app/database/migrations/`. If you open the newly created file, you will find some code that Laravel has generated for you. Migrations always have an `up()` and `down()` method that defines the schema changes when migrating up or down. By convention, the table and field names are written in "snake_case". Moreover, the table names are written in the plural form.

Our first migration is going to look like this:

```
public function up(){
  Schema::create('cats', function($table){
    $table->increments('id');
    $table->string('name');
    $table->date('date_of_birth');
    $table->integer('breed_id')->nullable();
    $table->timestamps();
  });
  Schema::create('breeds', function($table){
    $table->increments('id');
    $table->string('name');
  });
}
public function down(){
  Schema::drop('cats');
  Schema::drop('breeds');
}
```

The `date()` and `string()` methods create fields with the corresponding types
(in this case, `DATE` and `VARCHAR`) in the database, `increments()` creates an auto
incrementing `INTEGER` primary key, and `timestamps()` adds the `created_at` and
`updated_at` `DATETIME` fields that Eloquent expects by default. The `nullable()`
method specifies that the column can have `NULL` values.

To run the migration, enter the following command:

**$ php artisan migrate**

When it is run for the first time, this command will also create a `migrations` table
that Laravel uses to keep track of the migrations that have been run.

# Seeding the database

Rather than manually populating our database, we can use the seeding helpers
offered by Laravel. This time, there is no Artisan command to generate the file,
but all we need to do is create a new class called `BreedsTableSeeder.php` inside
`app/database/seeds/`. This class extends Laravel's `Seeder` class and defines the
following `run()` method:

```
class BreedsTableSeeder extends Seeder {
  public function run(){
    DB::table('breeds')->insert(array(
      array('id'=>1, 'name'=>"Domestic"),
```

```
        array('id'=>2,  'name'=>"Persian"),
        array('id'=>3,  'name'=>"Siamese"),
        array('id'=>4,  'name'=>"Abyssinian"),
    ));
    }
}
```

You can bulk insert an array but you could also insert arbitrary code in the `run()` method to load data from a CSV or JSON file. There are also third-party libraries that can help you generate large amounts of test data to fill your database.

To control the order of execution of the seeders, Laravel lets you call them individually inside `app/database/seeds/DatabaseSeeder.php`. In our case, since we only have one seeder, all we need to write is the line that follows:

```
$this->call('BreedsTableSeeder');
```

Then, we can seed the database by calling it using the following command:

```
$ php artisan db:seed
```

# Mastering Blade

Now that we have some information in our database, we need to define the templates that are going to display it. Blade is Laravel's lightweight template language and its syntax is very easy to learn. Here are some examples of how Blade can reduce the number of keystrokes and increase the readability of your templates:

| Standard PHP syntax | Blade syntax |
| --- | --- |
| `<?php echo $var; ?>` | `{{$var}}` |
| `<?php echo htmlentities($var); ?>` | `{{{$var}}}` |
| `<?php if($cond): ?>…<?php endif; ?>` | `@if($cond) … @endif` |

You should only use the double braces to output a variable if you trust the user or the value that is returned. In all other cases, make sure that you use the triple brace notation to avoid XSS vulnerabilities (explained in a bit more detail in the next chapter).

Blade supports all of PHP's major constructs to create loops and conditions: `@for`, `@foreach`, `@while`, `@if`, and `@elseif`; therefore, allowing you to avoid opening and closing the `<?php` tags everywhere in your templates.

---

**[ 39 ]**

# Creating a master view

Blade lets you build hierarchical layouts by allowing the templates to be nested and extended. The following code snippet is the "master" template that we are going to use for our application. We will save it as `app/views/master.blade.php`.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Cats DB</title>
    <link rel="stylesheet" href="{{asset('bootstrap-3.0.0.min.
css')}}">
  </head>
  <body>
    <div class="container">
      <div class="page-header">
        @yield('header')
      </div>
      @if(Session::has('message'))
        <div class="alert alert-success">
          {{Session::get('message')}}
        </div>
      @endif
    @yield('content')
    </div>
  </body>
</html>
```

The Bootstrap 3 CSS framework is included to speed up the prototyping of the application interface. You can download it from `http://getbootstrap.com` and place the minified CSS file inside the `public/` folder. To ensure that its path prefix is set correctly, even when Laravel is run from a subfolder, we use the `asset()` helper. In the following template, we will use other helpers such as `url` and `link_to`, which help us write more concise code and also escape from any HTML entities. To see the complete list of Blade template helpers that are available to you, visit: `http://laravel.com/docs/helpers`.

To inform the user about the outcome of certain actions, we have prepared a notification area between the header and the page content. This *flash data* (in other words, the session data that is only available for the next request) is passed and retrieved to and from the `Session` object.

The `@yield` directives act as placeholders for the different sections that a child view can populate and override. To see how a child template can re-use them, we are going to recreate the `About` view by changing its extension to `.blade.php` and extending our `master` template:

```
@extends('master')
@section('header')
  <h2>About this site</h2>
@stop
@section('content')
  <p>There are over {{$number_of_cats}} cats on this site!</p>
@stop
```

The `@section ... @stop` directives delimit the blocks of content that are going to be injected into the master template. You can see how this is done in the following diagram:

```
┌─────────────────────────────────────────────────────────────┐
│  master.blade.php              about.blade.php               │
│                                @extends('master')            │
│  ┌───────────────────┐        ┌──────────────────────┐      │
│  │   @yield('header') │◁──────│  @section('header')   │      │
│  └───────────────────┘        └──────────────────────┘      │
│  ┌───────────────────┐                                       │
│  │  Messages (if any) │                                       │
│  └───────────────────┘                                       │
│  ┌───────────────────┐        ┌──────────────────────┐      │
│  │   @yield('content')│◁──────│  @section('section')  │      │
│  └───────────────────┘        └──────────────────────┘      │
│                                                               │
└─────────────────────────────────────────────────────────────┘
```
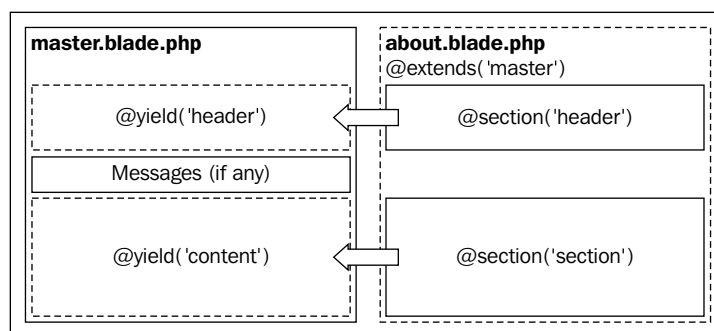
If you now reopen the `/about` route in your web browser, without changing anything in your previous route definition, you will see the new view. Laravel's view finder will simply use the new file, and since its name ends with `.blade.php`, treat it like a Blade template.

# Back to the routes

Now that we have a main template that we can extend and re-use, we can start to create the individual routes of our application inside `app/routes.php`, along with the different views that will display the application data.

# The overview page

This is the *index* page that is going to display all of the cats using the `cats.index` view. We will also re-use this view for the second route where the cats are filtered by their breed, since both the routes are almost identical. Note that Laravel expects you to use the dot notation (`cats.index` and not `cats/index`) to refer to a view located inside a subfolder:

```
Route::get('cats', function(){
  $cats = Cat::all();
  return View::make('cats.index')
    ->with('cats', $cats);
});

Route::get('cats/breeds/{name}', function($name){
  $breed = Breed::whereName($name)->with('cats')->first();
  return View::make('cats.index')
    ->with('breed', $breed)
    ->with('cats', $breed->cats);
});
```

The only novelty in these routes are the slightly more-advanced Eloquent queries. While we already know that the `all()` method in the first route loads all of the entries from the `cats` table, the second route uses a more complex query. The first method, `whereName`, is a dynamic method that translates into a `WHERE name = $name` SQL query. The `with()` method loads the related cat models and `first()` retrieves the first instance.

The template, saved inside `cats/index.blade.php`, will look like this:

```
@extends('master')

@section('header')
  @if(isset($breed))
    {{link_to('/', 'Back to the overview')}}
  @endif
<h2>
    All @if(isset($breed)) {{$breed->name}} @endif Cats

  <a href="{{url('cats/create')}}" class="btn btn-primary pull-right">
    Add a new cat
  </a>
</h2>
@stop
```

```
@section('content')
  @foreach($cats as $cat)
    <div class="cat">
      <a href="{{url('cats/'.$cat->id)}}">
        <strong> {{{$cat->name}}} </strong> - {{{$cat->breed->name}}}
      </a>
    </div>
  @endforeach
@stop
```

With the help of a `foreach` loop, the view iterates over the list of cats that it received from the route. Since we will be using this view to display both the index page (`/cats`) as well as the breed overview page (`/cats/breeds/breed`), we used the `@if` directives in two places to conditionally display more information.

# Displaying a cat's page

The next route is used to display a single cat. To find a cat by its ID, we use Eloquent's `find()` method:

```
Route::get('cats/{id}', function($id) {
  $cat = Cat::find($id);
  return View::make('cats.single')
    ->with('cat', $cat);
});
```

Since this is such a common pattern, Laravel provides you with a way to automatically bind a model to a route and, therefore, make your code shorter and more expressive. To bind the `$cat` variable to the `Cat` model, simply add the following declaration before your routes:

```
Route::model('cat', 'Cat');
```

This allows you to shorten your route and pass a `Cat` object to it instead:

```
Route::get('cats/{cat}', function(Cat $cat) {
  return View::make('cats.single')
    ->with('cat', $cat);
});
```

The view, `cats/single.blade.php`, does not contain any new directives. It simply displays the name of the cat with the links to edit or delete it. In the `content` section, we return its age and breed if the breed is set; this is shown in the following snippet:

```
@extends('master')

@section('header')
  <a href="{{url('/')}}">Back to overview</a>
  <h2>
      {{{$cat->name}}}
  </h2>
  <a href="{{url('cats/'.$cat->id.'/edit')}}">
    <span class="glyphicon glyphicon-edit"></span> Edit
  </a>
  <a href="{{url('cats/'.$cat->id.'/delete')}}">
    <span class="glyphicon glyphicon-trash"></span> Delete
  </a>
  Last edited: {{$cat->updated_at}}
@stop

@section('content')
  <p>Date of Birth: {{$cat->date_of_birth}} </p>
  <p>
      @if($cat->breed)
       Breed:
       {{link_to('cats/breeds/' . $cat->breed->name,
       $cat->breed->name)}}
      @endif
  </p>
@stop
```

# Adding, editing, and deleting cats

The next series of routes and views will be used to create, edit, and delete a cat page. Since we are going to use the same view for all of the three actions, we pass a `method` variable that the form will have to use to complete the action:

```
Route::get('cats/create', function() {
  $cat = new Cat;
  return View::make('cats.edit')
    ->with('cat', $cat)
    ->with('method', 'post');
});
```

Since we cannot bind a model before it exists, we simply pass a new and empty instance of a `Cat` model to the view. For the edit and delete routes, however, we can take advantage of the route model binding that we introduced earlier, as follows:

```
Route::get('cats/{cat}/edit', function(Cat $cat) {
  return View::make('cats.edit')
    ->with('cat', $cat)
    ->with('method', 'put');
});

Route::get('cats/{cat}/delete', function(Cat $cat) {
  return View::make('cats.edit')
    ->with('cat', $cat)
    ->with('method', 'delete');
});
```

The next set of routes will handle the three different types of actions and redirect the user using a flash data message. This message is then retrieved in the `Session` object using `Session::get('message')` in the master template. Any input data that is received by the application and that you would normally access via the `$_GET` or `$_POST` variables is instead retrievable using the `Input::get()` method. It is also possible to retrieve an array of all the input data with `Input::all()`. They also use more features of Eloquent that we have not seen before. The `POST /cats` and `PUT /cats/{cat}` routes, respectively, use the `create()` and `update()` methods from Eloquent with `Input::all()` as their argument. This is only possible because we specified the specific fields that are *fillable* in the `Cat` model beforehand:

```
Route::post('cats', function(){
  $cat = Cat::create(Input::all());
  return Redirect::to('cats/' . $cat->id)
    ->with('message', 'Successfully created page!');
});

Route::put('cats/{cat}', function(Cat $cat) {
  $cat->update(Input::all());
  return Redirect::to('cats/' . $cat->id)
    ->with('message', 'Successfully updated page!');
});

Route::delete('cats/{cat}', function(Cat $cat) {
  $cat->delete();
  return Redirect::to('cats')
    ->with('message', 'Successfully deleted page!');
});
```

The view used in all those different routes, `views/cats/edit.blade.php`, will be slightly more complex since it requires a few additional conditions. However, it remains more maintainable than the two or three individual files that would be required to display the different forms to add, edit, or delete a resource. You will also come across several form helpers, such as `Form::label()` or `Form::text()`, which essentially generate the equivalent HTML code for you.

```
@extends('master')

@section('header')
  <a href="{{('cats/'.$cat->id.'')}}">&larr; Cancel </a>
  <h2>
      @if($method == 'post')
        Add a new cat
      @elseif($method == 'delete')
        Delete {{$cat->name}}?
      @else
        Edit {{$cat->name}}
      @endif
  </h2>
@stop

@section('content')
  {{Form::model($cat, array('method' => $method, 'url'=>
  'cats/'.$cat->id))}}

  @unless($method == 'delete')
    <div class="form-group">
      {{Form::label('Name')}}
      {{Form::text('name')}}
    </div>
    <div class="form-group">
      {{Form::label('Date of birth')}}
      {{Form::text('date_of_birth')}}
    </div>
    <div class="form-group">
      {{Form::label('Breed')}}
      {{Form::select('breed_id', $breed_options)}}
    </div>

    {{Form::submit("Save", array("class"=>"btn btn-default"))}}
  @else
    {{Form::submit("Delete", array("class"=>"btn btn-default"))}}
  @endif

  {{Form::close()}}
@stop
```

In this view, we used another nifty feature of Laravel that lets us bind a model to a form. This is achieved with the `Form::model()` method, which expects an instance of a model as its first parameter. Once a model is bound to a form, any fields that are displayed are automatically populated based on the contents of the model. This will be particularly helpful once we add validation to the form.

The `Form::select()` helper builds a `<select>` dropdown with the different choices. It expects the list of choices to be passed to a multidimensional array. Rather than binding this array to each route, we can use **view composers**, another feature of Laravel, which allows you to bind a variable to a specific view each time. To create your first view composer, simply insert the following lines of code at the bottom of your `app/routes.php` file:

```
View::composer('cats.edit', function($view)
{
  $breeds = Breed::all();
  if(count($breeds) > 0){
    $breed_options = array_combine($breeds->lists('id'),
                                   $breeds->lists('name'));
  } else {
    $breed_options = array(null, 'Unspecified');
  }
  $view->with('breed_options', $breed_options);
});
```

With this final touch, the first version of our application is now complete! Anyone who visits it can create a page for their cat, and edit or delete it.

# Summary

We have covered a lot in this chapter. We learned how to define routes, prepare the models of the application, and interact with them. Moreover, we have had a glimpse at the many powerful features of Eloquent, Blade, as well as the other convenient helpers in Laravel to create forms and input fields: all of this in under 200 lines of code!

In the next chapter, we will learn how to use Laravel's built-in user registration, authentication, and security features to improve our existing application.

# 4
# Authentication and Security

In this chapter, we will improve the application we built in the previous chapter by adding a simple authentication mechanism and addressing any security issues with the existing code base. In doing so, you will learn about:

- How to configure and use the `Auth` class
- Filters and how to apply them to specific routes
- The most common security vulnerabilities in web applications
- How Laravel can help you write more secure code

## Authenticating users

Allowing users to register and sign in is an extremely common feature in web applications. Yet, PHP does not dictate in any way how it should be done, nor does it give you any helpers to implement it. This has led to the creation of disparate, and sometimes insecure, methods of authenticating users and restricting access to specific pages. In that respect, Laravel provides you with different tools to make these features more secure and easier to integrate. It does so with the help of its `Auth` class and a functionality that we have not covered yet, **route filters**.

## Creating the user model

First of all, we need to define the model that is going to be used to represent the users of our application. Laravel already provides you with sensible defaults inside `app/config/auth.php`, where you change the model or table that is used to store your user accounts.

It also comes with an existing `User` model inside `app/models/User.php`. For the purposes of this application, we are going to simplify it slightly, remove certain class variables, and add new methods so that it can interact with the `Cat` model:

```php
use Illuminate\Auth\UserInterface;
class User extends Eloquent implements UserInterface {
  public function getAuthIdentifier() {
    return $this->getKey();
  }
  public function getAuthPassword() {
    return $this->password;
  }
  public function cats(){
    return $this->hasMany('Cat');
  }
  public function owns(Cat $cat){
    return $this->id == $cat->owner;
  }
  public function canEdit(Cat $cat){
    return $this->is_admin or $this->owns($cat);
  }
}
```

The first thing to note is that this model implements the `UserInterface`. Remember that an interface does not give any implementation details. It is nothing more than a `contract` that specifies the names of the methods that a class should define when it implements the interface, in this case, `getAuthIdentifier()` and `getAuthPassword()`. These methods are used internally by Laravel when authenticating a user. The next method, `cats()`, simply defines the `has many` relationship with the `Cat` model. The last two methods will be used to check whether a given `Cat` instance is owned or editable by the current `User` instance.

# Creating the necessary database schema

Now that we have defined a `User` model, we need to create the database schema for it and alter the existing `cats` table to add information about the owner. Start by creating a new migration:

**$ php artisan migrate:make create_users**

And then define the `up` method with the necessary database columns:

```php
public function up(){
  Schema::create('users', function($table){
    $table->increments('id');
```

```
    $table->string('username');
    $table->string('password');
    $table->boolean('is_admin');
    $table->timestamps();
  });

Schema::table('cats', function($table){
  $table->integer('user_id')->nullable()->references('id')-
    >on('users');
});
}
```

With the preceding code, we create a new table for the users of our application. This table has a username, a password, and a flag to indicate whether the user is an administrator. We have also altered the existing `cats` table to add a `user_id` column that stores `id` of the owner of `Cat`. This time, we are also using the `references` and `on` methods that are used to create a **foreign key constraint** in the table. Foreign keys help you to enforce the consistency of data (for example, you would not be able to assign `Cat` to a non-existent user).

The code to reverse this migration will simply have to remove the foreign key constraint and the column and then drop the `users` table:

```
public function down(){
  Schema::table('cats', function($table){
    $table->dropForeign('cats_user_id_foreign');
    $table->dropColumn('user_id');
  });
  Schema::drop('users');
}
```

Next, we prepare a database seeder to create two users for our application, one of which will be an administrator.

```
class UsersTableSeeder extends Seeder {
  public function run(){
    User::create(array(
      array('username' =>'admin',
        'password' => Hash::make('hunter2'), 'is_admin' => true));

    User::create(array(
      'username' =>'scott', 'password' => Hash::make('tiger'),
        'is_admin' => false)
    ));
  }
}
```

Once you have saved this code inside a new file named `app/database/seeds/` `UsersTableSeeder.php`, do not forget to call it inside the main `DatabaseSeeder` class.

> Laravel expects all passwords to be hashed with the `Hash::make` helper, which uses **Bcrypt** to create a strong hash. You should never store passwords in *cleartext* or hash them with weak algorithms, such as md5 or sha1.

To run the migration and seed the database at the same time, enter:

```
$ php artisan migrate && php artisan db:seed
```

# Authentication routes and views

Let's now look at the new routes and views. We will start by making some amends to the master layout (`app/views/master.blade.php`) to display the login link to guests and the logout link to users who are logged in. To check whether a visitor is logged in, we use the `Auth::check()` method:

```
<div class="container">
  <div class="page-header">
    <div class="text-right">
      @if(Auth::check())
        Logged in as
        <strong>{{Auth::user()->username}}</strong>
        {{link_to('logout', 'Log Out')}}
      @else
        {{link_to('login', 'Log In')}}
      @endif
    </div>
  @yield('header')
  </div>
  @if(Session::has('message'))
    <div class="alert alert-success">
      {{Session::get('message')}}
    </div>
  @endif

  @if(Session::has('error'))
    <div class="alert alert-warning">
      {{Session::get('error')}}
    </div>
  @endif
  @yield('content')
</div>
```

This code replaces the contents of the `<body>` tag in our previous template file. A section for any error messages was also included below the header.

The route to display this login form could not be easier:

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

> If you were curious as to where and why Laravel uses the `make` methods in various places, it is only there to maintain PHP 5.3 compatibility, which does not support *class member access on instantiation*, and therefore, does not let you write `return new View('login');`.

The route that handles login attempts will simply pass the `username` and `password` input values to the `Auth::attempt` method. When this method returns `true`, we simply redirect the visitor to the intended location. If this fails, we redirect the user back to where he came from with `Redirect::back()` with the input values and an error message.

```
Route::post('login', function(){
  if(Auth::attempt(Input::only('username', 'password'))) {
    return Redirect::intended('/');
  } else {
  return Redirect::back()
    ->withInput()
    ->with('error', "Invalid credentials");
  }
});
```

But how does Laravel know what our intended location was? If you open `app/filters.php` and look at the `auth` filter, you will see that it redirects guests to the login route with the `Redirect::guest()` method. This method stores the requested path in a session variable, which is then used by the `intended()` method. The parameter passed to this method is the fallback route to which users should be redirected if there is no request path in the session information.

> Note also that there is a filter called `guest` that does the opposite of `auth`; you could use it on the login route if you wanted to prevent logged-in users from accessing it. With this filter in place, logged-in users will be redirected to the home page instead. You can change this behavior inside `app/filters.php`.

The login view, inside `app/views/login.blade.php`, is a simple form:

```
@extends('master')
@section('header')<h2>Please Log In</h2>@stop
@section('content')
  {{Form::open()}}
  <div class="form-group">
      {{Form::label('Username')}} {{Form::text('username')}}
  </div>
  <div class="form-group">
      {{Form::label('Password')}} {{Form::password('password')}}
  </div>
    {{Form::submit()}}
  {{Form::close()}}
@stop
```

Here is what the login form, as well as the updated master template, will look like:
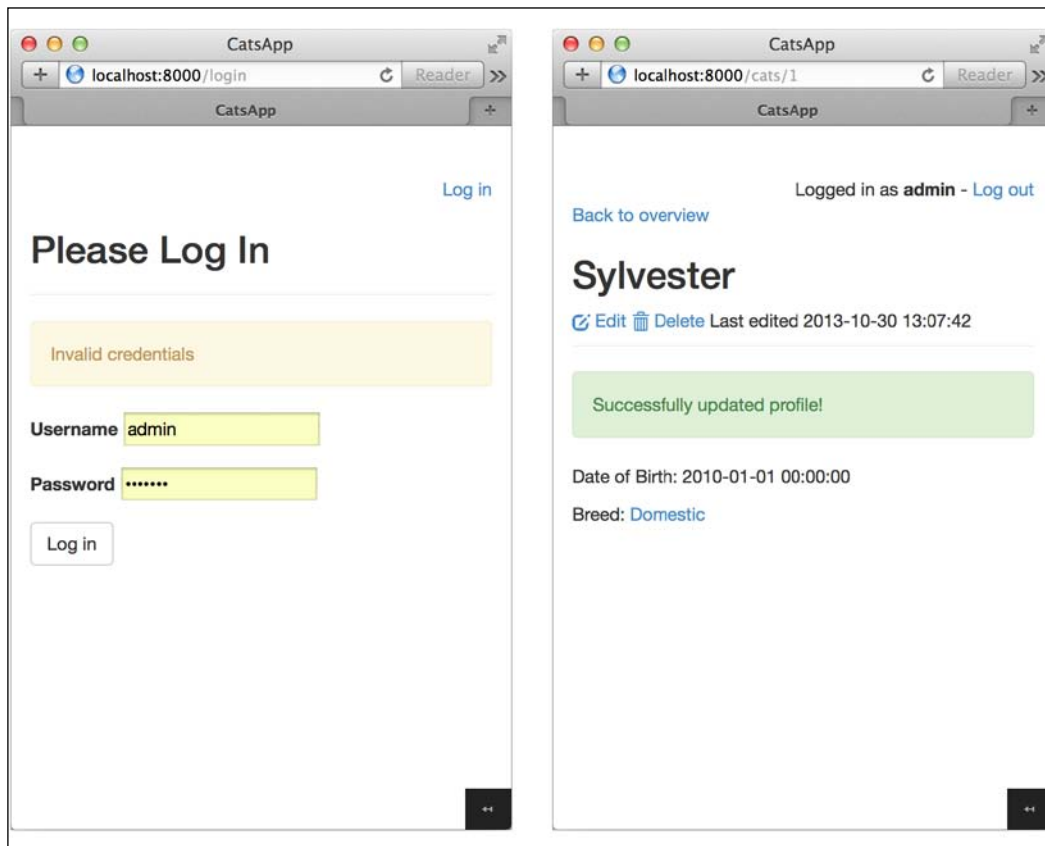
The last route we need to create is the one that is going to handle the logout action. All it needs to do is call `Auth::logout()` and then redirect the user to the home page with a message:

```
Route::get('logout', function(){
  Auth::logout();
  return Redirect::to('/')
    ->with('message', 'You are now logged out');
});
```

Then, we need to wrap the routes that require authentication inside a route group the following:

```
Route::group(array('before'=>'auth'), function(){
  Route::get('cats/create', function(){...});
  Route::post('cats', function(){...});
  ...
});
```

Every request to these routes will first execute the `auth` filter. We also need to protect the PUT and POST routes by adding a condition that checks whether the currently logged-in user is allowed to edit the page:

```
Route::put('cats/{cat}', function(Cat $cat) {
  if(Auth::user()->canEdit($cat)){
    $cat->update(Input::all());
    return Redirect::to('cats/' . $cat->id)
      ->with('message', 'Successfully updated profile!');
  } else {
    return Redirect::to('cats/' . $cat->id)
      ->with('error', "Unauthorized operation");
  }
});
```

In the views, we can use the following condition to determine whether or not a user should see the edit and delete links:

```
@if(Auth::check() and Auth::user()->canEdit($cat))
  Edit link | Delete link
@endif
```

Lastly, we need to alter the POST/cats route to make sure we save the identifier of the current user when a new Cat instance is created:

```
Route::post('cats', function(){
  $cat = Cat::create(Input::all());
  $cat->user_id = Auth::user()->id;
  if($cat->save()){
    return Redirect::to('cats/' . $cat->id)
      ->with('message', 'Successfully created profile!');
  } else {
    return Redirect::back()
      ->with('error', 'Could not create profile');
  }
});
```

The call to $cat->save() returns either true if the object was inserted into the database or false if there was an error, so that we can use it to redirect the user accordingly.

# Validating user input

Our application still has a major flaw: it does not perform any validation on the data submitted by users. While you might end up with a series of conditions with regular expressions here and there if you were to do this in plain PHP, Laravel offers a far more straightforward and robust way to achieve this.

Validation is performed by passing an array with the input data and an array with the validation rules to the Validator::make($data, $rules) method. In the case of our application, here are the rules we could write:

```
$rules = array(
  'name' => 'required|min:3', // Required, > 3 characters
  'date_of_birth' => array('required, 'date') // Must be a date
)
```

Multiple validation rules are separated by pipes, but they can also be passed in an array. Laravel provides over 30 different validation rules, and they are all documented in here:

http://laravel.com/docs/validation#available-validation-rules

Here is how we would check these rules with the data submitted in the form:

```
$validation_result = Validator::make($rules, Input::all());
```

You can then make your application act based on the output of `$validation_result->fails()`. If this method call returns `true`, you would retrieve an object containing all error messages with `$validation_result->messages()`, and this object is attached to a redirection that sends the user back to the form:

```
return Redirect::back()
  ->with('messages', $validation_result->messages());
```

Since each field can have zero or more validation errors, you would use a condition and a loop with the following methods to display those messages:

```
if($messages->has('name')){
  foreach ($messages->get('name') as $message){
    echo $message;
  }
}
```

You may also use a tool such as Ardent, which extends Eloquent and lets you to write validation rules directly inside the model:

`https://github.com/laravelbook/ardent`

# Securing your application

Before you deploy your application in a hostile environment, full of merciless bots and malicious users, there are a number of security considerations that you must keep in mind. In this section, we are going to cover several common attack vectors for web applications and learn about how Laravel protects your application against them. Since a framework cannot protect you against everything, we will also look at the common pitfalls to avoid.

# Cross-site request forgery

**Cross-site request forgery** (**CSRF**) attacks are conducted by targeting a URL that has side-effects (that is, it is performing an action and not just displaying information). We have already partly mitigated CSRF attacks by avoiding the use of `GET` for routes that have permanent effects such as `DELETE/cats/1`, since it is not reachable from a simple link or embeddable in an `<iframe>` element. However, if an attacker is able to send his victim to a page that he controls, he can easily make the victim submit a form to the target domain. If the victim is already logged in on the target domain, the application would have no way of verifying the authenticity of the request.

The most efficient countermeasure is to issue a token whenever a form is displayed and then check that token when the form is submitted. `Form::open` and `Form::model` both automatically insert a hidden `_token` input element.

Our application in its current form has several vulnerable endpoints. First of all, all the URLs that handle user input are not checking this CSRF token. To address this, we will group the POST, PUT, and DELETE routes inside their own route group with a `csrf` filter:

```
Route::group(array('before'=>'csrf'), function(){ ...} );
```

We can also protect the individual GET routes by adding the token in the URL with the `csrf_token()` function:

```
<a href="{{URL::to('logout?_token='.csrf_token())}}">Log out</a>
```

To attach a filter to an individual route, simply turn the second parameter of the route into an array, and add the name of the filter:

```
Route::get('logout', array('before'=>'csrf', function(){...}));
```

Multiple filters can be passed as a string separated by the pipe symbol:

```
Route::get('foo', array('before'=>'auth|csrf', function(){...});
```

# Escaping content to prevent cross-site scripting – XSS

**XSS** attacks happen when attackers are able to place client-side JavaScript code in a page viewed by other users. In our application, assuming that the name of Cat has not escaped, if we enter the following snippet of code as the value for the name, every visitor will be greeted with an alert message everywhere the name of Cat is displayed:

```
Evil Cat <script>alert('Meow!')</script>
```

While this is a rather harmless script, it would be very easy to insert a longer script or link to an external script that steals the session or cookie values. To avoid this kind of attack, you should never trust any user-submitted data or escape any dangerous characters. To do this, simply wrap your variables in three curly braces in your Blade templates.

```
{{{$cat->name}}}
```

Instead of executing the script, this time the `<script>` tag is displayed on the page since the angle brackets have escaped and displayed with their HTML entities (`&lt;` and `&gt;`).

# Avoiding SQL injection

An **SQL injection** vulnerability exists when an application inserts arbitrary and unfiltered user input in an SQL query. This user input can come from cookies, server variables, or, most frequently, through GET or POST input values. These attacks are conducted to access or modify data that is not normally available and sometimes to disturb the normal functioning of the application.

By default, Laravel will protect you against this type of attack since both the query builder and Eloquent use PHP's Data Objects class (PDO) behind the scenes. PDO uses **prepared statements**, which allow you to safely pass any parameters without having to escape and sanitize them.

In some cases, you might want to write more complex or database-specific queries in SQL. This is possible using the DB::raw method. When using this method, you must be very careful not to create any vulnerable queries like the following one:

```
Route::get('sql-injection-vulnerable', function(){
  $name = "'Bobby' OR 1=1";
  return DB::select(
    DB::raw("SELECT * FROM cats WHERE name = $name") );
});
```

To protect this query from SQL injection, you need to rewrite it by replacing the parameters with question marks in the query and then pass the values in an array as a second argument to the raw method:

```
Route::get('sql-injection-not-vulnerable', function(){
  $name = "'Bobby' OR 1=1";
  return DB::select(
    DB::raw("SELECT * FROM cats WHERE name = ?", array($name)));
});
```

# Using mass-assignment with care

In the previous chapter, we used mass-assignment, a convenient feature that allows us to create a model based on the form input without having to assign each value individually.

This feature should, however, be used carefully. A malicious user could alter the form on the client side and add a new input to it:

```
<input name="admin" value="1" >
```

Then, when the form is submitted and we attempt to create a new model using:

```
Cat::create(Input::all())
```

[ 59 ]

Thanks to the `$fillable` array, which defines a white list of fields that can be filled through mass assignment, this method call will throw a mass-assignment exception.

It is also possible to do the opposite and define a blacklist with the `$guarded` property. However, this option can be potentially dangerous since you might forget to update it when adding new fields to the model.

# Cookies – secure by default

Laravel makes it very easy to create, read, and expire cookies with its `Cookie` class.

You will also be pleased to know that all cookies are automatically signed and encrypted. This means that if they are tampered with, Laravel will automatically discard them. This also means that you will not be able to read them from the client side using JavaScript.

# Forcing HTTPS when exchanging sensitive data

If you are serving your application over HTTP, you need to bear in mind that every bit of information that is exchanged, including passwords, is sent in *cleartext*. An attacker on the same network could therefore intercept private information, such as session variables, and log in as the victim. The only way we can prevent this is to use HTTPS. If you already have an SSL certificate installed on your web server, Laravel comes with a number of helpers to switch between `http://` and `https://` and restrict access to certain routes. You can, for instance, define an `https` filter that will redirect the visitor to the secure route as shown in the following code snippet:

```
Route::filter('https', function() {
  if (!Request::secure())
    return Redirect::secure(URI::current());
});
```

# Summary

In this chapter, we have learned how to make use of many of Laravel's tools to add authentication features to a website, validate data, and avoid common security problems. In the next chapter, we will cover another important aspect of modern web applications: testing, which is another area where Laravel shines.

# 5
# Testing – It's Easier Than You Think

Testing is an often-neglected part in PHP development. Compared to languages such as Java and Ruby, where testing is strongly ingrained into the mindsets of developers, PHP has been lagging behind. This is mainly because simple PHP applications tend to be tightly coupled and are, therefore, difficult to test. However, thanks to standardization and modularization efforts and frameworks that encourage the separation of concerns, PHP testing has become more accessible and the mentality towards it is slowly changing.

Laravel 4 is a framework that was built from the ground up to facilitate testing. It comes with all the necessary files to get started along with different helpers to test your application, thus helping beginners to overcome some of the biggest obstacles.

In this chapter, we will demonstrate how Laravel makes it very simple to get started with testing without forcing you to go for a test-first approach, or making you aim for a complete test coverage. In this gentle introduction to testing, we will look at the following topics:

- The advantages of writing tests for your application
- How to prepare your tests
- The software design patterns that Laravel fosters
- How to use Mockery to test objects in isolation
- The built-in features and helpers that facilitate testing

# The benefits of testing

If you have not written tests for your web applications before, the advantages of testing might not always be obvious to you. After all, preparing and writing tests is a significant time investment, and for short-lived prototypes or hackathon projects, they can even seem to be a complete waste of time. However, in almost all of the other cases, when your project is likely to grow in complexity, or when you collaborate with other developers, tests have the potential to save you and other people a lot of time and headaches.

Tests also introduce some changes to your workflow. In the development stage, you will no longer have to switch back and forth between your code editor and your web browser. Instead, if you bind the test runner to a keyboard shortcut, most of the testing will take place inside your editor or IDE.

Once you have proven that a certain bit of functionality works, you will have a way of quickly ensuring that it continues to work as expected if the source code is changed at a later date. In addition to this, it forces you to clearly and unambiguously define the expected behavior of your application and can therefore complement or replace a significant part of the documentation. This can be particularly helpful, not only for new developers who start collaborating on the project, but also for yourself, if you have not touched the project for a while.

# The anatomy of a test

Your application tests will reside in `app/tests/`. In this directory, you will find a base test case inside `TestCase.php`, which is responsible for bootstrapping the application in the testing environment. This class extends Laravel's main `TestCase` class, which in turn extends the `PHPUnit_Framework_TestCase` class along with many helpful testing methods that we will cover later in this chapter. All of your tests will extend this first `TestCase` class and define one or more methods that are meant to test one or more features of your application.

In every test, we generally perform the following three distinct tasks:

1.  We *arrange* or initialize some data.
2.  We execute a function to *act* on this data.
3.  We *assert* or verify that the output matches what we expected.
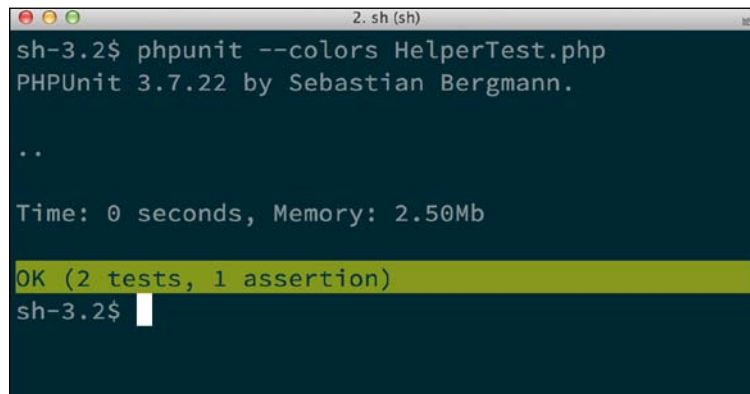
Here is an example test case, `HelperTest.php`, which illustrates the three preceding steps:

```php
class Helper {
  public static function sum($arr){ return array_sum($arr); }
}
class HelperTest extends PHPUnit_Framework_TestCase{
  public function testSum(){
    $data = array(1,2,3);              // 1) Arrange
    $result = Helper::sum($data);      // 2) Act
    $this->assertEquals(6, $result);   // 3) Assert
  }
  public function testSomethingElse(){
    // ...
  }
}
```

When the preceding code snippet is executed, PHPUnit will run each method within the test case and keep track of how many tests failed or passed. With PHPUnit installed on your system, you can run this test using the following command:

```
$ phpunit --colors HelperTest.php
```

This will produce the following output:



> Most code editors also provide ways to run this directly within the editor by pressing a shortcut key, and it is even possible to run them automatically before each commit or before you deploy your code to a remote server.

# Unit testing with PHPUnit

A positive effect of testing is that it forces you to split your code into manageable dependencies so you can test them in isolation. The testing of these individual classes and methods is referred to as **unit testing**. Since it relies on the PHPUnit testing framework, which already provides a large number of tools to set up test suites, Laravel does not need to provide any additional helpers for this type of testing.

A great way to learn about any framework, and at the same time learn about the different ways in which it can be tested, is to look at how its authors have written tests for it. Therefore, our next examples will be taken directly from Laravel's test suite, which is located at `vendor/laravel/framework/tests/`.

# Defining what you expect with assertions

Assertions are the fundamental components of unit tests. Simply put, they are used to compare the *expected output* of a function with its *actual output*.

To see how assertions work, we will examine the test for the `Str::is()` helper, which checks whether a given string matches a given pattern.

The following test can be found at the bottom of the `Support/SupportStrTest.php` file:

```php
use Illuminate\Support\Str;
class SupportStrTest extends PHPUnit_Framework_TestCase {
  // ...
  public function testIs()
  {
    $this->assertTrue(Str::is('/', '/'));
    $this->assertFalse(Str::is('/', ' /'));
    $this->assertFalse(Str::is('/', '/a'));
    $this->assertTrue(Str::is('foo/*', 'foo/bar/baz'));
    $this->assertTrue(Str::is('*/foo', 'blah/baz/foo'));
  }
}
```

The preceding test performs five assertions that test whether the method is indeed returning the expected value when called with different parameters.

PHPUnit provides many other assertion methods that can, for example, help you test for numerical values with `assertGreaterThan()`, equality with `assertEquals()`, types with `assertInstanceOf()`, or existence with `assertArrayHasKey()`. While there are many more possible assertions, these are the ones you will probably end up using most frequently. In total, PHPUnit provides around 40 different assertion methods, all of which are described in the official documentation at `http://phpunit.de/manual/`.

# Preparing the scene and cleaning up objects

If you need to run a function before each test method to set up some test data or reduce code duplication, you can use the `setUp()` method. If, on the other hand, you need to run some code after each test to clear any objects that were instantiated in your tests, you can define it inside the `tearDown()` method.

# Expecting exceptions

It is also possible to test for exceptions by decorating your function with an `@expectedException` DocBlock, like Laravel does inside `Database/DatabaseEloquentModelTest.php`:

```
/**
 * @expectedException Illuminate\Database\Eloquent\
MassAssignmentException
 */
public function testGlobalGuarded()
{
  $model = new EloquentModelStub;
  $model->guard(array('*'));
  $model->fill(array('name' => 'foo', 'age' => 'bar',
                     'votes' => 'baz'));
}
```

In this test function, there is no assertion, but the code is expected to throw an exception when it is executed. Also note the use of an `EloquentModelStub` object. A stub creates an instance of an object that provides or simulates the methods that our class needs—in this case, an Eloquent model on which we can call the `guard()` and `fill()` methods. If you look at the definition of this stub further down in the test, you will see that it does not actually interact with a database but it provides canned responses instead.

# Testing interdependent classes in isolation

In addition to stubs, which we looked at in the previous section, there is another way in which you can test one or more interdependent classes in isolation. It is by using **mocks**. In Laravel, mocks are created using the Mockery library, and they help define the methods that should be called during the test, the arguments they should receive, and their return values as well. Laravel heavily relies on mocks in its own tests. An example can be found in the tests for the `Paginator` class where the pagination `Environment` class is mocked. This class is responsible for interfacing the view with the current HTTP request.

```php
use Illuminate\Pagination\Paginator;

class PaginationPaginatorTest extends PHPUnit_Framework_TestCase {

  public function tearDown(){
    Mockery::close();
  }

  public function testPaginationContextIsSetupCorrectly(){
    $p = new Paginator(
        $env = Mockery::mock('Illuminate\Pagination\Environment'),
        array('foo', 'bar', 'baz'), 3, 2);

    $env->shouldReceive('getCurrentPage')
->once()->andReturn(1);
    $p->setupPaginationContext();

    $this->assertEquals(2, $p->getLastPage());
    $this->assertEquals(1, $p->getCurrentPage());
  }
}
```

As opposed to stubs, mocks allow us to define which methods need to be called, how many times they should be called, which parameters they should receive, and which parameters they should return. If any of these preconditions are not met, the test will fail.

In the previous example, the `getCurrentPage()` method is called when the pagination context is set up. This method will call `getCurrentPage()` on the mocked `Environment` class, which will always return `1` instead of retrieving this value from the request query string (for example, `?page=2`).

To ensure that we do not have an instance of a mocked object that persists and interferes with future tests, Mockery provides a `close()` method that needs to be executed after each test. Thanks to this mock, the class can be tested in complete isolation.

# End-to-end testing

When we are confident that all of the interdependent classes work as expected, we can conduct another type of testing. It consists of simulating the kind of interaction that a user would have through a web browser. This user would, for example, visit a specific URL, perform certain actions, and expect to see some kind of feedback from the application.

This is perhaps the most straightforward type of testing, as it mimics the kind of testing that you manually perform each time you refresh your browser after a code change. When you get started, it is absolutely fine to only perform this type of testing. However, you must bear in mind that if any errors occur, you will still have to drill deep down into your code to find the exact component that caused the error.

# Testing – batteries included

When you start a new project with Laravel, you are provided with both a configuration file with sensible defaults for PHPUnit at the root of the project inside `phpunit.xml` as well as a directory, `app/tests/`, where you are expected to save your tests. This directory even contains an example test that you can use as a starting point.

With these settings in place, from the root of our project, all we need to enter is:

```
$ phpunit
```

This command will read the XML configuration file and run our tests. If at this stage you get an error message telling you that PHPUnit cannot be found, you either need to add the `phpunit` command to your `PATH` variable or install it locally with Composer.

If you need to install it via Composer, you can insert it in a `require-dev` block as follows, since it is only a development dependency:

```
"require-dev": {
  "phpunit/phpunit": "3.7.*"
}
```

After running `composer update`, you will be able to call PHPUnit using the following command:

```
$ vendor/bin/phpunit
```

# Framework assertions

Now that we know about the two major types of tests and have PHPUnit installed, we are going to write a few tests for the application that we developed in the previous two chapters.

This first test will verify whether visitors are redirected to the correct page when they first visit our site:

```
public function testHomePageRedirection() {
  $this->call('GET', '/');
  $this->assertRedirectedTo('cats');
}
```

Here, we made use of the `call()` method that simulated a request to our application. Then we used one of the assertion methods provided by Laravel to make sure that the response is a redirection to the new location. If you now run the `phpunit` command, you should see the following output:

```
OK (1 test, 2 assertions)
```

Next, we can try to write a test to make sure that the creation form is not accessible to the users that are not logged in; this is shown in the following code snippet:

```
public function testVisitorIsRedirected() {
  $this->call('GET', '/cats/create');
  $this->assertRedirectedTo('login');
}
```

Unfortunately, this time the test will fail. This is because Laravel does not apply filters on routes when they are called from within the testing environment. This is entirely intentional; it encourages you to test the filter and the action that uses it in isolation. However, it is possible to override this behavior. By calling `Route::enableFilters()` in our test's `setUp()` method, we can tell Laravel that it has to apply the filters regardless.

# Impersonating users

Sometimes, you may wish to run a test as if you were a registered user of the application. This is possible by using the `be()` method and passing a `User` instance to it or whichever Eloquent model you use along with Laravel's authentication class:

```
public function testLoggedInUserCanCreateCat() {
  Route::enableFilters();
  $user = new User(array('name'=>'John Doe',
                         'is_admin'=>false));
  $this->be($user);
  $this->call('GET', '/cats/create');
  $this->assertResponseOk();
}
```

# Testing with a database

While some developers would advise you against writing tests that hit the database, it can often be a simple and effective way of making sure that all the components work together as expected. However, it should only be done once each individual unit has been tested. Let's also not forget that Laravel has support for migrations and seeding; in other words, it has all of the tools that are required to recreate an identical data structure from scratch before each test.

To write tests that depend on a database, we need to override the `setUp()` method in our tests to migrate and seed the database each time a test is run. It is also important to run the parent `setUp()` method, otherwise, the test case will not be able to start properly:

```
public function setUp(){
  parent::setUp();
  Artisan::call('migrate');
  $this->seed();
}
```

Then, we need to configure the database in the testing environment, `app/config/testing/database.php`; if the application does not contain any database-specific queries, we can use SQLite's in-memory feature by setting `:memory:` instead of a path to the database file. The following configuration also has the potential to speed up our tests:

```
'sqlite' => array(
  'driver'   => 'sqlite',
  'database' => ':memory:',
),
```

And lastly, since we are going to test the editing and deletion features, we are going to need at least one row in the `cats` table of our database, so we prepare a seeder that will insert a cat with a forced `id` of value `1`:

```
class CatsTableSeeder extends Seeder {
  public function run(){
    Cat::create(array('id'=>1, 'name'=>'Figaro', 'user_id'=>1));
  }
}
```

Once this is done, we can test the deletion feature as follows:

```
public function testOwnerCanDeleteCat() {
  $user = new User(array('id'=>1, 'name'=>'User #1', 'is_
admin'=>false));
  $this->be($user);
  $this->call('DELETE', '/cats/1');
  $this->assertRedirectedTo('/cats');
  $this->assertSessionHas('message');
}
```

Note that this time, we did not need to enable the filters since the permissions are checked by a method in the `User` model. Since the database is wiped and reseeded after each test, we do not need to worry about the fact that the previous test deleted that particular cat. We can also write a test to ensure that a user who is not an administrator cannot edit someone else's cat profile:

```
public function testNonAdminCannotEditCat() {
  $user = new User(array('id'=>2, 'name'=>'User #2', 'is_
admin'=>false));
  $this->be($user);
  $this->call('DELETE', '/cats/1');
  $this->assertRedirectedTo('/cats/1');
  $this->assertSessionHas('error');
}
```

# Inspecting the rendered views

Since Laravel ships with Symfony's `DomCrawler` and `CssSelector` components, it is possible to inspect the contents of a rendered view. By issuing a request through the test client instance with `$this->client->request()`, you can filter its contents with CSS queries as follows:

```
public function testAdminCanEditCat() {
  $user = new User(array('id'=>3, 'name'=>'Admin',
'is_admin'=>true));
    $this->be($user);
    $new_name = 'Berlioz';
    $this->call('PUT', '/cats/1', array('name' => $new_name));
    $crawler = $this->client->request('GET', '/cats/1');
    $this->assertCount(1, $crawler
                    ->filter('h2:contains("'.$new_name.'")'));
}
```

The complete documentation for the DomCrawler component can be found at `http://symfony.com/doc/current/components/dom_crawler.html`. If you are already familiar with jQuery, its syntax will look familiar to you.

# Summary

While the main ideas behind testing are easy to grasp, it is often their implementation that can prove to be an obstacle, especially when working with a new framework. However, after reading this chapter, you should have a good overview of how you can test your Laravel applications. The techniques presented in this chapter will enable you to write more robust and future-proof applications.

In the next chapter, we will explore the possibilities offered by Artisan, Laravel's command-line utility.

# 6
# A Command-line Companion
# Called Artisan

In the last few chapters we have used Artisan for various tasks, such as starting a development server and running database migrations. However, as we will see in this chapter, Laravel's command-line utility has far more capabilities and can be used to run and automate all sorts of tasks. In the next pages, you will learn how Artisan can help you:

- Inspect and interact with your application
- Enhance the overall performance of your application
- Install and use third-party commands released by the community
- Speed up your workflow by using generators
- Automate and ease deployment
- Write your own commands

By the end of this tour of Artisan's capabilities, you will understand how it can become an indispensable companion in your workflow.

## Keeping up with the latest changes

New features are constantly added to Laravel. If a few days have passed since you first installed it, try running a `composer update` from your terminal. Provided you are using the `dev-master` version, you should see the latest versions of Laravel and its dependencies being downloaded. Since you are already in the terminal, finding out about the latest features is just one command away:

```
$ php artisan changes
```

This saves you from going online to find a change log or reading through a long history of commits on GitHub. It can also help you learn about features that you were not aware of. You can also find out which version of Laravel you are running by entering the following:

```
$ php artisan --version
Laravel Framework version 4.1-dev
```
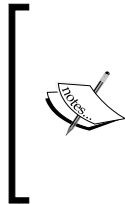
> All Artisan commands have to be run from your project's root directory.
>
> With the help of a short script like `Artisan Anywhere`, available at `https://github.com/antonioribeiro/artisan-anywhere`, it is also possible to run artisan from any subfolder in your project.

# Inspecting and interacting with your application

With the `routes` command, you can see at a glance as to which URLs your application will respond to, what their names are, and if any filters will be applied before and after each request. This is probably the quickest way to get acquainted with a Laravel application that someone else has built.

To display a table with all the routes, all you have to do is enter:

```
$ php artisan routes
```

For example, here is how the application we built in *Chapter 3*, *Your First Application* looks like:

```
sh-3.2$ php artisan routes
+--------+---------------------------+------+---------+----------------+---------------+
| Domain | URI                       | Name | Action  | Before Filters | After Filters |
+--------+---------------------------+------+---------+----------------+---------------+
|        | GET /                     |      | Closure |                |               |
|        | GET /about                |      | Closure |                |               |
|        | GET /cats                 |      | Closure |                |               |
|        | GET /cats/breeds/{name}   |      | Closure |                |               |
|        | GET /cats/{id}            |      | Closure |                |               |
|        | GET /cats/create          |      | Closure | auth           |               |
|        | GET /cats/{cat}/edit      |      | Closure | auth           |               |
|        | GET /cats/{cat}/delete    |      | Closure | auth           |               |
|        | POST /cats                |      | Closure | auth, csrf     |               |
|        | PUT /cats/{cat}           |      | Closure | auth, csrf     |               |
|        | DELETE /cats/{cat}        |      | Closure | auth, csrf     |               |
|        | GET /login                |      | Closure | guest          |               |
|        | POST /login               |      | Closure |                |               |
|        | GET /logout               |      | Closure | csrf           |               |
+--------+---------------------------+------+---------+----------------+---------------+
sh-3.2$
```

In some applications, you might see /{v1}/{v2}/{v3}/{v4}/ {v5} appended to particular routes. This is because the developer has registered a controller with implicit routing, and Laravel will try to match and pass up to five parameters to the controller. We will look at controllers more closely in the next chapter and see why it is better to define routes explicitly.

# Fiddling with the internals

When developing your application, you will sometimes need to run short, one-off commands to inspect the contents of your database, insert some data into it, or check the syntax and results of an Eloquent query. One way you could do this is by creating a temporary route with a closure that is going to trigger these actions. However, this is less than practical since it requires you to switch back and forth between your code editor and your web browser.

To make these small changes easier, Artisan provides a command called `tinker`, which boots up the application and lets you interact with it. Just enter:

```
$ php artisan tinker
```

This will start a **Read-Eval-Print Loop** (**REPL**) similar to what you get when running the `php -a` command, which starts an interactive shell. In this REPL, you can enter PHP commands in the context of the application and immediately see their output:

```
> $cat = 'Garfield';
> Cat::create(array('name'=>$cat,'date_of_birth'=>new DateTime));
> echo Cat::whereName($cat)->get();
[{"id":"4","name":"Garfield 2","date_of_birth":…}]
> var_dump(Config::get('database.default'));
> string(6) "sqlite"
```

As of Version 4.1, Laravel leverages Boris, a PHP-specific REPL that provides a more robust shell with support keyboard shortcuts and history. However, since Boris only supports **POSIX** compliant systems, Laravel will fall back to a more basic shell if you are running it on Windows.

# Turning the engine off

Whether it is because you are upgrading a database or waiting to push a fix for a critical bug to production, you may want to manually put your application on hold to avoid serving a broken page to your visitors. You can do this by entering:

```
$ php artisan down
```

This will put your application into `maintenance` mode. To define what happens when a visitor lands on your application when it is in this state, simply edit the `App:down` handler inside `app/global/start.php` to add a custom message, render a view, or redirect the user. To exit the maintenance mode, simply run:

```
$ php artisan up
```

# Fine-tuning your application

For every incoming request, Laravel has to load many different classes and this can slow down your application, particularly if you are not using a PHP accelerator such as **APC**, **eAccelerator**, or **XCache**. In order to reduce disk I/O and shave off precious milliseconds from each request, you can run:

```
$ php artisan optimize
```

This will trim and merge many common classes into one single file located inside `bootstrap/compiled.php`. The optimize command is something you could, for example, include in a deployment script.
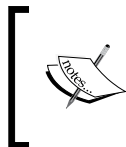
> By default, Laravel will not compile your classes if `app.debug` is set to `true`. You can override this by adding the `--force` flag to the command but bear in mind that this will make your error messages less readable.

# Installing third-party commands

We will see later in this chapter that it is very easy to create your own Artisan commands to automate recurring tasks. Moreover, the more common this task is, the more likely it will be that someone else has already written a command for it. Indeed, the Laravel community is coming up with many extremely useful commands and we are going to discuss a few here.

# Speeding up your workflow with generators

As you add more and more features to an existing codebase or create new applications, you will find yourself writing similar code over and over again. Whether it is a migration class or a form to edit a model, the underlying code will have a fair bit of boilerplate.

Luckily, *Jeffrey Way*, a tutorial author and prominent member of the Laravel community, has created a suite of generators to automate the creation of these classes. Greatly inspired by what you could find in a framework like Ruby on Rails, these generators facilitate the creation of migrations, seeds, models, views, controllers, tests, and forms.

Rather than manually copying and pasting classes from another project or from the documentation, they allow you to directly create all the necessary files in the right locations.

To install the generators package, simply run:

```
$ composer require way/generators:dev-master --dev
```

This will add the dependency to the `require-dev` block in your `package.json` file and download the package to your `vendor/` directory.

To enable the commands in Artisan, you have to open `app/config/app.php` and add `Way\Generators\GeneratorsServiceProvider` to the array of service providers. Although we have not covered service providers before, just think of them as a way of rapidly enabling or disabling features in your application. They are used to register classes that are used in your application or, in our case, new Artisan commands.

Once this is done, to see the full list of available generators from the terminal, run:

```
$ php artisan
```

Each generator will also tell you the parameters it expects if you call it without any arguments. Here is an example of the output of the `form` command when it does not receive any arguments:

```
$ php artisan generate:form

  [RuntimeException]
  Not enough arguments.
generate:form [--method[="..."]] [--html[="..."]] model
```

# Generating migrations

To demonstrate the power of generators, we will use the `generate:migration` command to recreate the database migrations we created in *Chapter 3*, *Your First Application*.

```
$ php artisan generate:migration create_cats_table
--fields="name:string, date_of_birth:date:nullable"
```

This is all it takes! If you now open `app/database/migrations`, you will find the newly created migration file, which is practically identical to the one you wrote previously.

Provided you name your migrations according to certain conventions, the migration will be set up to either create a new table, add a column, or delete it. The generator understands the following keyword prefixes:

- `create` or `add` to create a table:

    ```
    $php artisan generate:migration create_foo_table
    ```

- `add` or `insert` to add a field:

    ```
    $php artisan generate:migration add_bar_to_foo_table --
        fields="bar:text"
    ```

- `remove`, `drop`, or `delete` to remove a field:

    ```
    $php artisan generate:migration drop_bar_from_foo_table --
        fields="bar:text"
    ```

Note that in the last example, even though we are dropping a field, we are still passing a value to the `--fields` parameter to guarantee that the `down` method reverts the migration as expected.

# Generating HTML forms

With `generate:form`, you can instantaneously produce the necessary HTML markup for a form for a given model:

```
$ php artisan generate:form user
```

The generator will even perform some database introspection and automatically pick an adequate field type for each property; `<textarea>` for text fields, `<input type="checkbox">` for booleans, and a standard `<input type="text">` parameter for almost all the other fields. Even though the markup might require some tweaking before being included in your views, this can greatly speed up your workflow.

## Generating everything else

Jeffrey Way's package also offers several other generators, which will not be covered in this book since their use is relatively straightforward and comparable to the two generators presented. The author is also constantly improving the package and chances are high that new generators will have emerged since this book was published. Make sure you visit `https://github.com/JeffreyWay/Laravel-4-Generators` to find out about the entire list of generators and some instructions on how to use them.

# Deploying with a single command

If you have used languages like Ruby or Python, you might have used deployment tools such as **Capistrano** or **Fabric**. Up to this day, these tools are used even by PHP developers since there was no robust equivalent written in pure PHP. However, with the advent of tools like **Rocketeer**, it is now possible to write deployment scripts exclusively with PHP and, therefore, reduce the learning curve as well as the number of external dependencies.

Fully compatible with Laravel 4.1 but also usable with Version 4.0 with a few additional installation steps, Rocketeer provides a handful commands under the `deploy` namespace to upload new releases or roll back to previous ones.

The package's installation guide as well the full documentation are available at: `http://rocketeer.autopergamene.eu/`

After installing the package with Composer, you will be asked to run a `config:publish` command, which will copy the necessary configuration files from the package into your application's configuration directory. Artisan provides similar `:publish` commands for assets and views, and you will come across many packages that depend on that step in the installation process.

The easiest way to learn how to use Rocketeer is to read through the newly created configuration file to get a feel of what is possible, and then to incrementally adapt it to your own needs.

## Deployment, the old-school way

Whenever possible, you should use web hosts that give you SSH access so that you can make use of Laravel's Remote package and use Rocketeer or any other deployment tool.

In some situations, however, you might not have this luxury and the only alternative will be to upload everything via FTP. While we do not have to worry about the number of files and their size when Composer fetches the packages on the server, with FTP each upload can take a non-negligible amount of time, especially on slow or tunneled connections.

Luckily, there is a simple package, `barryvdh/laravel-vendor-cleanup`, that eases the pain by providing you with a `vendor-cleanup` command that does exactly what it says on the tin. It cleans up the `vendor/` directory, which is full of files such as tests and documentation that are completely superfluous in a production environment. When used with a default Laravel 4.0 installation, this cuts the size of the vendor directory by half (from 28 MB to 14 MB). The command also knows about the unnecessary files in common PHP packages such as `dompdf` or `Assetic`.

# Rolling out your own artisan commands

Maybe at this stage you might be thinking about writing your own bespoke commands. As you will see, this is surprisingly easy to do with Artisan. If you have used Symfony's Console component, you will be pleased to know that an Artisan command is simply an extension of it with a slightly more expressive syntax. This means that the various helpers will prompt for input, show a progress bar, or format a table, all available from within Artisan.

The command that we are going to write depends on the application we built in *Chapter 3*, *Your First Application*. It will allow you to export all cat records present in the database as a CSV with or without a header line. If no output file is specified, the command will simply dump all records on the screen, in a formatted table.

# Creating the command

There are only two required steps to create a command. Firstly, you need to create the command itself and then you need to register it manually.

This time, the generator to create the class is bundled with Laravel directly:

```
$ php artisan command:make ExportCatsCommand
```

This will generate a class inside `app/commands/`, which you then need to register inside `app/start/artisan.php` by adding the following line to it:

```
Artisan::add(new ExportCatsCommand);
```

If you now run `php artisan`, you should see a new command called `command:name`. This command does not do anything yet, but before we start writing the functionality, let's briefly look at how it works internally.

# The anatomy of a command

Inside the newly created command class, you will find some code that was generated for you. We will walk through the different properties and methods and see what their purpose is.

The first two properties are the name and description of the command. Nothing exciting here; this is only the information that will be shown in the command line when you run Artisan. The colon is used to namespace the commands.

```
$name = 'export:cats';
$description = 'Export all cats';
```

Then you will find the `fire` method. This is the method that gets called when you run a particular command. From there, you may retrieve the arguments and options passed to the command or run other methods.

```
function fire()
```

Lastly, there are two methods that are responsible to define the list of arguments or options that are passed to the command.

```
function getArguments() { /* Array of arguments */ }
function getOptions() { /* Array of options*/ }
```

Each argument or option can have a name, a description, and a default value which can be required or optional. Additionally, options can have a shortcut.

To understand the difference between arguments and options, consider the following command, where options are prefixed with two dashes:

```
$ command --option_one=value --option_two -v=1 argument_one
  argument_two
```

In this example, `option_two` does not have a value; it is only used as a flag. The `-v` only has one dash since it is a shortcut.

Arguments can be retrieved with `$this->argument($arg)`, and options—you guessed it—with `$this->option($opt)`. If these methods do not receive any parameters, they simply return the full list of parameters.

# Writing the command

We are going to start by writing a method that retrieves all cats from the database and returns them as an array:

```
protected function getCatsData(){
  $cats = Cat::with('breed')->get();
  foreach($cats as $cat){
    $output[] = array($cat->name,
      $cat->date_of_birth,
      $cat->breed->name);
  }
  return $output;
}
```

There should not be anything new here; we could have used the `toArray()` method, which turns an Eloquent collection into an array, but we would have had to flatten the array and exclude certain fields.

Then we need to define what arguments and options our command expects:

```
protected function getArguments(){
  return array(
    array('file', InputArgument::OPTIONAL,
      'The output file', null),
  );
}

protected function getOptions(){
  return array(
    array('headers', 'h', InputOption::VALUE_NONE,
      'Display headers?', null),
  );
}
```

The last parameter is the default value that the argument and option should have if it is not specified. In both cases we want it to be `null`.

Lastly, we write the logic for the `fire` method:

```
public function fire(){
  $output_path = $this->argument('file');

  $headers = array('Name', 'Date of Birth', 'Breed');
  $rows = self::getCatsData();

  if($output_path){
    $handle = fopen($output_path, 'w');
```

```
        if($this->option('headers')){
          fputcsv($handle, $headers);
        }
        foreach($rows as $row){
          fputcsv($handle, $row);
        }
        fclose($handle);
          $this->info("Exported list to $output_path");
      } else {
          $table = $this->getHelperSet()->get('table');
          $table->setHeaders($headers)->setRows($rows);
          $table->render($this->getOutput());
      }
    }
```

While the bulk of this method is relatively straightforward, there are a few novelties. The first one being the use of the `$this->info()` method, which writes an information message to the output. If you need to show an error message in a different color, you can use the `$this->error()` method.

Further down in the code, you will see some functions that are used to generate a table. As we mentioned previously, an Artisan command extends the `Symfony Console` component and, therefore, inherits all of its helpers. These can be accessed with `$this->getHelperSet()`. Then it is only a matter of passing arrays for the header and rows of the table and calling the `render` method.

To see the output of our command, we run:

```
$ php artisan export:cats
$ php artisan export:cats --headers file.csv
```

# Summary

In this chapter, we have learned about the different ways in which Artisan can assist you in the development, debugging, and deployment process. We have also seen how easy it is to build a custom Artisan command and adapt it to your own needs.

If you are relatively new to the command line, you will have had a glimpse into the power of command-line utilities. If, on the other hand, you are a seasoned user of the command line and you have written scripts with other programming languages, you can surely appreciate the simplicity and expressiveness of Artisan.

In the next chapter, we will take a second look at several of the features that were presented in *Chapter 3*, *Your First Application*, and *Chapter 4*, *Authentication and Security*, and see how you would use them in more complex applications.

# 7
# Architecting Ambitious Applications

Now that you have had the first taste of most of Laravel's main components, we will take a second look at their more advanced capabilities and how you can use them when writing larger applications. You will find that as your application grows in size and complexity, you will not have to sacrifice the structure and readability of your code.

In the next couple of pages, you will learn how to:

- Split up your routes into controllers and resources
- Write more complex and efficient queries with **Eloquent**
- Use different configuration settings depending on where the application runs
- Include your own PHP classes and functions
- Use Laravel's automatic JSON serialization and deserialization features

By the end of this chapter, you will have a better understanding of how to scale a small- to medium-sized Laravel application, improve its performance, and tweak it to better suit your needs.

# Moving from simple routing to powerful controllers

Up until this point, we have intentionally left out the C in MVC in the code examples, and defined all of our application endpoints using simple routes. As you can imagine, having all the routes and their actions defined in a single file works really well for lightweight projects. But this will rapidly turn into a maintenance nightmare if you exceed a handful of routes. To avoid this, we will look at how you can split them up into controllers while keeping `routes.php` as the map of your application.

Controllers are classes that reside inside `app/controllers/`. In addition to any private methods, they will have one or more methods called controller actions, which essentially serve the same purpose as the anonymous functions we attached to our routes.

Consider the following example:

```
Route::get('user/{nickname}/photos', array('before'=>'auth',
  function($nickname){
  // Perform some operations…
  return "something";
}));
```

To achieve the same functionality with a controller and remove the business logic from the routes, simply create a new controller, `UserController.php`:

```
class UserController {
  public function __construct(){
    $this->beforeFilter('auth');
  }
  public function photos($nickname){
    $this->doSomething();
    return "something";
  }
   private function doSomething(){ /* Any business logic */}
}
```

This approach can greatly improve the reusability and testability of your code, especially if `doSomething()` is used in more than one controller action. You could test it just once in isolation and then rely on it. When you venture into more advanced topics such as dependency injection and start using repositories, you can even swap out entire classes when you instantiate the controller, but we will not cover this here.

> Also note that filters can now be attached globally or selectively inside the controller's constructor. If you want to restrict it to specific actions or HTTP verbs, please refer to the official documentation that shows you how to do that at:
>
> `http://laravel.com/docs/controllers#controller-filters`

Finally, to tell Laravel which controller action to use, simply shorten the route declaration as follows:

```
Route::get('user/{nickname}/photos',
  array('uses'=>'UserController@photos'));
```

# Favoring explicit routing

Many MVC frameworks, such as CodeIgniter or earlier versions of Ruby on Rails, encouraged the use of implicit routing, which, as its name implies, allows you to implicitly route URLs to their corresponding controller action.

While it is possible and even very easy to automatically route `/GET foo/bar/baz/1` to the `getBar` action of a `FooController` (and passing `baz` and `1` as its two parameters) with Laravel, it is often better to spend a little bit more time wiring up the routes explicitly. Even though implicit routing works relatively well for simple applications, it limits the flexibility and precision of your routes. For instance, you would have no way of defining the number and format of the expected arguments to the controller action. `/foo/bar/1234` and `/foo/bar/baz/0` would both hit the same controller action. This controller action would then be responsible for handling routing exceptions, something it should not have to do. On the other hand, with explicit routing, you can precisely define the name and expected pattern of each argument.

Another argument in favor of explicit routing is that it makes it much easier to browse a codebase and allows Artisan to generate a cleaner table when using `php artisan routes`.

# Straightforward REST routing

Laravel greatly simplifies the creation of REST APIs with its resource controllers. Since they adhere to conventions, there is only a limited defined set of actions that can be performed from the controller's actions; `POST /cats` to create a resource, `PUT /cats/1` to update it, and so on.

In fact, almost all of the routes from *Chapter 3*, *Your First Application*, could be rewritten as follows:

```
Route::resource('cat', 'CatController');
```

This will register the following routes:

| Verb | Path | Action |
| --- | --- | --- |
| GET | /cat | index |
| GET | /cat/create | create |
| POST | /cat | store |
| GET | /cat/{id} | show |
| PUT | /cat/{id}/edit | edit |
| PUT | /cat/{id} | update |
| DELETE | /cat/{id} | destroy |

Then, in a `CatController` class, you would have all the different actions: `index`, `store`, `show`, `update`, and `destroy`. This is really all you need since they are then wired up to respond to the correct route and verb! Resource controllers also have two methods, `create` and `edit`, that are used to display a form to edit the resource.

# Supercharging your models

In *Chapter 3*, *Your First Application*, we learned how to use Eloquent to create models that represent the entities of our applications. We defined simple foreign key relationships and we used basic methods such as `all()`, `whereField($value)`, and `find($id)`. While these are the methods you will use frequently, Eloquent has a ton of other useful features and we will now look at some of the most useful ones.

# Simple performance tricks

Just like with any ORM that issues SQL queries for you, with Eloquent you can sometimes shoot yourself in the foot if you do not understand what it does behind the scene. For this reason, we will start by looking at a couple of things to consider improving the overall performance of your application.

# Eager loading records

When displaying several records that have one or more foreign key relationships, Eloquent will execute a new SQL query to resolve the related records every single time. This means that if you have 10 cats in your database and you want to display their breed, which is stored as a foreign key, there will be 10+1 database queries. This is referred as an *n+1* problem; the first query retrieves the record and the *n* others go and find the related records. As *n* or the number of related fields increase, the impact on the performance of the application will become more noticeable.

To avoid this, make sure you tell Laravel the relations it should preload by using the `with` method:

```
Cat::with('breed', 'owner')->get();
```

This will perform a first query to load all cats and then perform only two more queries to retrieve the owners and breeds. If you were displaying 50 cats per page, for example, eager loading would bring the total number of SQL queries down from 101 to 3!

# Selecting only what you need

If you are working with large datasets or tables with many columns, you should avoid loading the values of all columns into memory. To only select the columns you need, pass an array to the `get` method in your query:

```
Cat::get(array('name'));
```

This query will simply turn into a `SELECT name FROM cats` SQL statement.

# Profiling your queries

If you are curious about what happens behind the scene when you execute a query on an Eloquent object, you should install a profiler package. You can choose between `https://github.com/loic-sharma/profiler`, which is a port of Laravel 3's profiler, or `https://github.com/barryvdh/laravel-debugbar`, which integrates the generic PHP Debug Bar package. Once installed, they add a toolbar at the bottom of each rendered view, which shows the SQL queries that were issued on a given page and the amount of memory used by a request to your application. This is often the best way to identify potential bottlenecks in your code when you work with smaller data sets in your local development database.

# Foolproof models with soft deletes

Usually, when a record is deleted from the database, it is gone forever. If you want to give the users of your application the ability to recover accidentally deleted data, simply enable soft deletes on your models by giving them a `protected $softDelete = true;` property. You will also need to add a `DATETIME` column called `deleted_at` to the model. This column then remains empty until the record is removed with the `delete()` method.

Don't worry about adapting any of your queries to exclude deleted records; Eloquent takes care of all of this for you by adding the necessary `WHERE` clause to each SQL query. If you want to override this behavior, you can chain the `withTrashed()` method or, alternatively use `onlyTrashed()` to get only the records that were deleted:

```
Cat::onlyTrashed()->get();
```

To force the deletion of a model, you need to use the `forceDelete()` method.

# More control with SQL

If you are used to writing SQL by hand, you will probably appreciate Eloquent's expressiveness, but at the same time, in some instances, you will miss the fine-grained control you had with SQL. Fortunately, it is possible to run lower-level commands by using Laravel's query builder, which allows you to call methods to issue advanced `WHERE` clauses and `JOIN` or `GROUP BY` statements. The overhead in terms of performance is negligible and you still have the benefit to have a more readable syntax. Not to mention that your code will be free from SQL injection vulnerabilities.

If you do need to execute raw SQL in a `select` clause, for example, because you need to use a feature that does not exist in Laravel's query builder or because the SQL code is specific to a particular database engine, you can use the `DB::raw` method.

```
DB::table('cats')->select(DB::raw('count(*) as cats_count'))
```

If you need to execute a complete statement and not just a `SELECT` statement, you can use `DB::statement()`:

```
DB::statement("TRUNCATE cats");
```

# Listening for model events

The last really useful feature of Eloquent that we will look at is *model events*. Eloquent models fire events before and after they are created, saved, updated, deleted, or restored. It is very easy to listen for those events and execute a function when they occur. You could, for example, add an event listener to log changes to database records or send yourself an e-mail whenever a new user signs up.

These event listeners can be defined in a separate file, for example, `app/events.php`, which is loaded when the app starts (we will see how to do that later in this chapter). In simple cases, however, it is absolutely fine to keep them inside the model.

If, for example, we wanted to prevent an administrator from deleting a user who still has some cats in the database, we would intercept the deletion as follows:

```php
class User extends Eloquent {

  public static function boot(){
    parent::boot();

    static::deleting(function($user){
      if($user->cats->count())
        return false;
    });
  }
}
```

Since we are overriding `boot()`, we need to make sure we call this method on the parent class. This method is responsible for booting the model in a static context. By using the boot method instead of the constructor method, we avoid binding the event for each new instance.

# The handy paginator class

As you would expect, it is possible to only return portions of a dataset. Rather than the SQL `LIMIT` and `OFFSET` terms, Eloquent prefers two verbs that are slightly more expressive, `skip()` and `take()`:

```php
Cat::skip(100)->take(10)->get();
```

With these methods, along with `count()`, which are all inherited from the query builder class, you have all the building blocks to write a paginator to browse the results.

---

**[ 91 ]**

However, Laravel does not expect you to write the pagination logic by yourself. Instead, it allows you to attach a `paginate` method at the end of any query. Thus, the previous query could simply be rewritten as follows:

```
Cat::paginate(10);
```

The only thing missing here is the number of records to skip. Laravel's paginator retrieves this information from the page query string in the URL and then automatically passes it to the paginator. Therefore, the URL that returns the same results as our first query would be: `/cats?page=11`.

To display the previous and next links and the page numbers in your Blade templates, just call the `links` method on your Eloquent collection:

```
{{ $cats->links() }}
```

This will output the complete HTML markup, which is compatible with Twitter Bootstrap.

# Environment configuration made easy

You will rarely have identical settings in each environment in which your application runs. You should, for example, always turn off error reporting on a production server. Perhaps you will have a database running somewhere else than on `localhost` in your production environment or maybe you want to intercept outgoing e-mail with a local SMTP server in the development environment. To achieve this, the last thing you would want to do is to add a series of conditions inside your configuration files!

Luckily, Laravel makes it very easy to override the default configuration settings found inside `app/config/`. The mechanism to detect and apply the correct settings depending on the environment is very simple. All you have to do is define the names of the different environments along with a list of corresponding host names or IP addresses (for example, `localhost`, `127.0.0.1`, or `dev.example.com`) inside `bootstrap/start.php`:

```
$environments = array(
  'development' => array('127.0.0.1', 'localhost'),
  'production' => array('*.com'),
);
```

Then, if the app is served from `example.com`, Laravel will automatically apply any settings that are defined inside the `app/config/production/` folder. Remember that you only have to define the settings that differ. Therefore, if all you need to do is disable error messages, you could have an `app/config/production/app.php` file that only contains:

```
return array('debug' => false,);
```

In the case of the `app.php` configuration file, this allows you to avoid duplicating the array of service providers and aliases.

> Should you ever need to retrieve the name of the current environment in your application, you can use the `App::environment()` method.

# Environments and Artisan

From *Chapter 6*, *A Command-line Companion Called Artisan*, you know that all of Artisan's commands start an instance of the Laravel application in your terminal. This time, since the application is not served from a web server with a hostname, you will have to explicitly tell Laravel which environment to use by setting the `env` flag in your command. For example, if you have a development environment named `local`, to migrate your database, you would run:

```
$ php artisan migrate --env=local
```

Omitting the `--env` flag is a frequent mistake when getting started with Laravel and it can be the source of confusing error messages.

Alternatively, if you want to avoid appending the `env` flag to every command, you can add the hostname of your machine to the list in `bootstrap/start.php`. To find it out, run Artisan's `tinker` command and enter:

```
>var_dump(gethostname());
string(10) "macbook.local"
```

In this case, the value that we need to add to the array of local environment hostnames is `macbook.local`.

# Adding your own configuration settings

Configuration files are nothing more than simple files that return associative arrays. If your application depends on some API credentials or any other configuration values, do not leave them in your `routes.php` file. Instead, create a new configuration file and store the settings there. This has several benefits; it is cleaner, it allows you to swap them out depending on the environment (like any other Laravel configuration value), and if you need to, you can also exclude them from version control.

To retrieve and use them in your code, Laravel provides a `Config::get($setting, $default)` method that accepts an optional second parameter for a fallback value if the parameter is not set.

# Bringing in your own classes

Whether it is to add a third-party library or incorporate legacy code, new users of Laravel are often confronted with the problem of not knowing where to place new classes so that they can be called from their application.

Maybe this will come as a surprise to you, but you are actually free to place the source files wherever you deem appropriate, be it at the root of `app/` or in a subfolder. This only leaves you with the problem of making sure that the code is callable. It turns out that there are at least three ways in which this can be achieved, all of which require you to edit `composer.json`.

The first method uses the `file` array, where you simply list the files that should be loaded when the application starts:

```
files:[
  "app/helpers.php"
]
```

This method is well suited to load individual files that only contain functions and do not depend on object-oriented properties or namespacing.

The second way to load custom code is to add the name of the directory to the `classmap` array without removing the existing values:

```
"autoload": {
  "classmap": [
    "app/libraries/",
    "…"
  ]
}
```

You can use this method if you want to make all the classes from that directory callable.

The last method adds a `"psr-0"` object. As its name implies, it is best suited with namespaced classes of `PSR-0`:

```
"autoload": {
  "...",
  "psr-0": {
    "Cats\\": "app"
  }
}
```

> 📝 Since this is a JSON configuration file, we are using a second slash to escape the namespace separator.

You can now place your namespaced library in the `app/` folder and load it from anywhere in your application:

```php
<?php
use Cats\Presenters\BreedPresenter;
```

This will look for a file called `app/Cats/Presenters/BreedPresenter.php`.

With every method described, do not forget to run `composer dump-autoload` to refresh the classmap and make sure that PHP can find your classes.

# Playing nice with the frontend

JSON has become the de facto format to exchange data with APIs and web browsers. Out of the box, Laravel provides several features that make working with JSON a breeze. It automatically serializes arrays and Eloquent collections to JSON when they are returned from a route. It even serializes paginated results, as you can see in the example that follows:

```php
Route::get('json-test', function(){ return Cat::paginate(2); });
```

This will return the following JSON response:

```
{"total":2,"per_page":2,"current_page":1,"last_page":1,"from":1,
  "to":2,"data":[{...},{...}]}
```

When receiving a request with data in the JSON format, Laravel automatically deserializes it and merges it with the other input it receives. This means that you can retrieve it with `Input::get('key')`, the same method you use to retrieve GET or POST data.

If you have ever had to build a backend for a `Backbone.js` application or simply created a script to receive form data submitted via jQuery, you will undoubtedly find these features extremely convenient.

# Summary

In this final chapter, we have covered a few of the slightly more advanced features of Laravel. While there are many more topics to explore, you should now have a good understanding of the possibilities offered by Laravel and you will probably find it easier to read up on any features that are specific to the type of applications you will be building in the real world.

In the appendix, you will be presented with a handy reference for many of the other helpful features that Laravel offers out of the box.

# An Arsenal of Tools

Laravel comes with several utilities that help you perform specific tasks, such as sending e-mails, queuing functions, and manipulating files. It ships with a ton of handy utilities that it uses internally; the good news is that you can also use them in your applications. This appendix will present the most useful utilities so you do not end up rewriting a function that already exists in the framework!

The structure of this appendix is partly based on *Jesse O'Brien's* cheat sheet, which is accessible at `http://cheats.jesse-obrien.ca/`. The examples are based on Laravel's tests as well as its official documentation and API.

## Array helpers

Arrays are the bread and butter of any web application that deals with data. PHP already offers nearly 80 functions to perform various operations on arrays, and Laravel complements them with a handful of practical functions that are inspired by certain functions found in Python and Ruby.
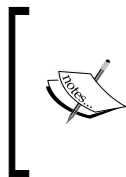
> Several of Laravel's classes, including Eloquent collections, implement the PHP `ArrayAccess` interface. This means that you can use them like a normal array in your code and, for instance, iterate over the items in a `foreach` loop or use them with the array functions described here.

Most of the functions support a **dot notation** to refer to nested values, which is similar to JavaScript objects. For example, rather than writing `$arr['foo']['bar']['baz']`, you can use the `array_get` helper and write `array_get($arr, 'foo.bar.baz');`.

In the following usage examples, we will use three dummy arrays and assume that they are reset for each example:

```
$associative =  array(
  "foo" => 1,
  "bar" => 2,
);
$multidimensional = array(
  "foo" => array(
      "bar" => 123,
  ),
);
$list_key_values = array(
  array("foo" => "bar"),
  array("foo" => "baz"),
);
```

# The usage examples of array helpers

We will now have a look at how we can use Laravel's array helper functions to extract and manipulate the values of those arrays:

- To retrieve a value with a fallback value if the key does not exist, we use the `array_get` function:

  ```
  array_get($multidimensional, 'foo.bar', 'default');
  // Returns 123
  ```

- To remove a value from an array using the dot notation, we use the `array_forget` function:

  ```
  array_forget($multidimensional, 'foo.bar');
  // $multidimensional == array( 'foo' => array() );
  ```

- To remove a value from an array and return it, we use the `array_pull` function:

  ```
  array_pull($multidimensional, 'foo.bar');
  // Returns 123 and removes the value from the array
  ```

- To set a nested value using the dot notation, we use the `array_set` function:

  ```
  array_set($multidimensional, 'foo.baz', '456');
  // $multidimensional == array( 'foo' => array( 'bar' =>
  123, 'baz' => '456' ) );
  ```

- To flatten a multidimensional associative array, we use the `array_dot` function:

```
array_dot($multidimensional);
// Returns array( 'foo.bar' => 123 );
array_dot($list_key_values);
// Returns array( '0.foo' => 'bar', '1.foo' => 'baz' );
```

- To return all of the the keys and their values from the array except for the ones that are specified, we use the `array_except` function:

```
array_except($associative, array('foo',));
// Returns array('bar' => 2);
```

- To only extract some keys from an array, we use the `array_only` function:

```
array_only($associative, array('bar'));
// Returns array('bar' => 2);
```

- To return a flattened array containing all of the the nested values (the keys are dropped), we use the `array_fetch` function:

```
array_fetch($list_key_values, 'foo');
// Returns array('bar', 'baz')
```

- To iterate over the array and return the first value for which the closure returns true, we use the `array_first` function:

```
array_first($associative, function($key, $value){
    return $key == "foo";
});
// Returns 1
```

- To generate a one-dimensional array containing only the values that are found in a multidimensional array, we use the `array_flatten` function:

```
array_flatten($multidimensional);
// Returns array(123)
```

- To extract an array of values from a list of key-value pairs, we use the `array_pluck` function:

```
array_pluck($list_key_values, 'foo');
// Returns array('bar', 'baz');
```

- To get the first or last item of an array (this also works with the values returned by functions), we use the `head` and `last` functions:

```
head($array); // Aliases to reset($array)
last($array); // Aliases to end($array)
```

# String and text manipulation

The string manipulation functions are found in the `Illuminate\Support` namespace and are callable on the `Str` object.

Most of the functions also have shorter `snake_case` aliases. The `Str::endsWith()` method is, for example, identical to the global `ends_with()` function. We are free to use whichever one we prefer in our application.

## Boolean functions

The following functions return `true` or `false` values:

- The `is` method checks whether a value matches a pattern. The asterisk can be used as a wildcard character:

  ```
  Str::is('projects/*', 'projects/php/'); // Returns true
  ```

- The `contains` method checks whether a string contains a given substring:

  ```
  Str::contains('Getting Started With Laravel', 'Python');
  // returns false
  ```

- The `startsWith` and `endsWith` methods check whether a string starts or ends with one or more substrings:

  ```
  Str::startsWith('.gitignore', '.git'); // Returns true
  Str::endsWith('index.php', array('html', 'php')); //
  Returns true
  ```

## Transformation functions

In some cases you need to transform a string before displaying it to the user or using it in a URL. Laravel provides the following helpers to achieve this:

- This function generates a URL-friendly string:

  ```
  Str::slug("A/B testing's fun!");
  // Returns "ab-testings-fun"
  ```

- This function generates a title where every word is capitalized:

  ```
  Str::title('getting started with laravel');
  // Returns "Getting Started With Laravel"
  ```

- This function caps a string with an instance of a given character:

  ```
  Str::finish('/one/trailing/slash', '/');
  Str::finish('/one/trailing/slash/', '/');
  // Both will return "/one/trailing/slash/"
  ```

- This function limits the number of characters in a string:

```
Str::limit($value, $limit = 100, $end = '...')
```

- This function limits the number of words in a string:

```
Str::words($value, $words = 100, $end = '...')
```

# Inflection functions

The following functions help you find out the plural or singular form of a word, even if it is irregular:

- This function finds out the plural form of a word:

```
Str::plural('fish');
// Returns "fish"
```

- This function finds out the singular form of a word:

```
Str::singular('elves');
// Returns "elf"
```

# Dealing with files

Far more elegant and consistent than its native counterparts in PHP, Laravel's file manipulation functions make it easier to write the web and console applications that deal with the uploads along with the filesystem.

# File uploads

The following functions will make it easier for you to deal with file uploads in your application:

- This function creates a form to send files (by adding `enctype='multipart/form-data'` to the `<form>` element):

```
Form::open(array('files' => true))
```

- This function creates a file upload field:

```
Form::file('avatar');
```

- This function retrieves the uploaded file and saves it to an existing folder inside `app/storage/`:

```
$avatar = Input::file('avatars')->move(storage_path() .
"/uploads/avatars");
```

- This function retrieves the path of the uploaded file:

```
$path = Input::file('avatar')->getRealPath();
```

- This function retrieves the original name of an uploaded file:

```
$name = Input::file('avatar')->getClientOriginalName();
```

- This function retrieves the extension of the uploaded file:

```
$extension = Input::file('avatar')-
>getClientOriginalExtension();
```

- This function retrieves the size of an uploaded file:

```
$size = Input::file('photo')->getSize();
```

# File manipulation methods

The `File` class also exposes several methods to retrieve and manipulate files:

- The following methods check for the existence or type:

```
File::exists('path/to/file/or/directory');
File::isDirectory('path/to/directory');
File::isWritable('path/to/directory');
File::isFile('path/to/file');
```

- This method gets the contents of a file:

```
File::get('path/to/file');
```

- This method creates a file and writes to it (if the file exists, it is overwritten):

```
File::put('path/to/file', 'contents');
```

- The following methods append or prepend to a file:

```
File::append('path/to/file', contents');
File::prepend('path/to/file', contents');
```

- This method deletes a file:

```
File::delete('path/to/file');
```

- The following methods move or copy files:

```
File::move('path/to/file', 'new/path/to/file');
File::copy('path/to/file', 'path/to/file-copy');
```

- The following methods help you get the basic information about a file:

```
File::extension('path/to/file');
File::type('path/to/file');
File::size('path/to/file');
File::lastModified('path/to/file');
```

- This method gets an array of all the subdirectories within a given directory:

  ```
  File::directories('path/to/directory');
  ```

- This method gets an array of all the files in a directory:

  ```
  File::files('path/to/directory');
  ```

- This method recursively lists all of the files in a directory and its subdirectories:

  ```
  File::allFiles('directory');
  ```

- These methods perform common operations on directories:

  ```
  File::makeDirectory('path/to/new/directory', $mode,
  $recursive);
  File::copyDirectory('source', 'destination', $options =
  null);
  File::deleteDirectory('path/to/directory');
  ```

- This method empties a directory but does not delete it:

  ```
  File::cleanDirectory('path/to/directory');
  ```

# Sending e-mails

Laravel's `Mail` class extends the popular Swift Mailer package, which makes sending e-mails a breeze. The e-mail templates are loaded in the same way as views, which means that you can use the Blade syntax and inject data into your templates.

- To inject some data into a template located inside `app/views/email/view.blade.php`, we use the following function:

  ```
  Mail::send('email.view', $data, function($message){});
  ```

- To send both an HTML and a plain text version, we use the following function:

  ```
  Mail::send(array('html.view', 'text.view'), $data,
  $callback);
  ```

- To delay the e-mail by 5 minutes (this requires a queue), we use the following function:

  ```
  Mail::later(5, 'email.view', $data, function($message){});
  ```

Inside the `$callback` closure that receives the message object, we can call the following methods to alter the message that is to be sent:

- `$message->subject('Welcome to the Jungle');`
- `$message->from('email@example.com', 'Mr. Example');`
- `$message->to('email@example.com', 'Mr. Example');`

Some of the less common methods include:

- `$message->sender('email@example.com', 'Mr. Example');`
- `$message->returnPath('email@example.com');`
- `$message->cc('email@example.com', 'Mr. Example');`
- `$message->bcc('email@example.com', 'Mr. Example');`
- `$message->replyTo('email@example.com', 'Mr. Example');`
- `$message->priority(2);`

To attach or embed files, you can use the following methods:

- `$message->attach('path/to/attachment.txt');`
- `$message->embed(''path/to/attachment.jpg');`

If you already have the data in memory, and you do not want to create additional files, you can use either the `attachData` or the `embedData` method:

- `$message->attachData($data, 'attachment.txt');`
- `$message->embedData($data, 'attachment.jpg');`

Embedding is generally done with image files, and you can use either the `embed` or the `embedData` method directly inside the body of a message as shown in the following code snippet:

```
<p>Product Screenshot:</p>
<p>{{$message->embed('screenshot.jpg')}}</p>
```

# Easier date and time handling with Carbon

Laravel bundles Carbon (`https://github.com/briannesbitt/Carbon`), which extends and augments PHP's native `DateTime` object with more expressive methods. Laravel uses it mainly to provide more expressive methods on the date and time properties (`created_at`, `edited_at`, and `deleted_at`) of an Eloquent object. However, since the library is already there, it would be a shame not to use it elsewhere in the code of your application.

# Instantiating Carbon objects

Carbon objects are meant to be instantiated like normal DateTime objects. They do, however, support a handful of more expressive methods.

- Carbon objects can be instantiated using the default constructor that will use the current date and time:
    - `$now = new Carbon();`

- They can be instantiated using the current date and time in a given timezone:
    - `$jetzt = new Carbon('Europe/Berlin');`

- They can be instantiated using expressive methods:
    - `$yesterday = Carbon::yesterday();`
    - `$demain = Carbon::tomorrow("Europe/Paris");`

- They can be instantiated using exact parameters:
    - `Carbon::createFromDate($year, $month, $day, $tz);`
    - `Carbon::createFromTime($hour, $minute, $second, $tz);`
    - ```
      Carbon::create($year, $month, $day, $hour,
      $minute, $second,
      $tz);
      ```

# Outputting user-friendly timestamps

We can generate human-readable relative timestamps such as *5 minutes ago, last week,* or *in a year* with the `diffForHumans()` method:

```
$post = Post::find(123);
echo $post->created_at->diffForHumans();
```

# Boolean methods

Carbon also provides a handful of simple and expressive methods that will come in handy in your controllers and views:

- `$date->isWeekday();`
- `$date->isWeekend();`
- `$date->isYesterday();`
- `$date->isToday();`
- `$date->isTomorrow();`

- `$date->isFuture();`
- `$date->isPast();`
- `$date->isLeapYear();`

# Carbon for Eloquent DateTime properties

To be able to call Carbon's methods on attributes stored as DATE or DATETIME types in the database, you need to list them in a `getDates()` method in the model:

```
class Post extends Eloquent {
  // ...
  public function getDates() {
    return array('created_at', 'edited_at', 'published_at',);
  }
}
```

# Don't wait any longer with queues

Queues allow you to defer the execution of functions without blocking the script. They can be used to run all sorts of functions, from e-mailing a large number of users to generating PDF reports.

As of Version 4.1, Laravel is compatible with the following queue drivers:

- Beanstalkd, with the `pda/pheanstalk` package
- Amazon SQS, with the `aws/aws-sdk-php` package
- IronMQ, with the `iron-io/iron_mq` package

Each queue system has its advantages. Beanstalkd can be installed on your own server; Amazon SQS might be more cost-effective and require less maintenance, as will IronMQ, which is also cloud-based. The latter also lets you set up *push queues*, which are great if you cannot run background jobs on your server.

# Creating a job and pushing it onto the queue

A **job** is nothing more than a class with a `fire` method that accepts two parameters for the name of the job and the data and that either deletes the job or releases it back to the queue:

```
class Job {
  public function fire($job, $data){
    // Perform job…
```

[ **106** ]

```
        // If the job was successful, delete it
        $job-delete();
        // Put it back onto the queue and try to execute it again
        $job->release($seconds);
    }
}
```

Push a job onto the queue. When called, this will execute the `fire` method:

```
Queue::push('Job', $data);
```

Push a job onto the queue but execute a different method:

```
Queue::push('Job@method', $data);
```

You can store your jobs anywhere in your application folder as long as the class is resolvable (see *Chapter 7, Architecting Ambitious Applications*). A potential location would be `app/jobs`. When developing, it is also fine to push anonymous functions onto the queue:

```
Queue::push(function($job) use ($vars, $from, $outer, $scope){
  // Perform job
});
```

# Listening to a queue and executing jobs

The following are the functions used for listening to a queue and executing jobs:

- We can listen to the default queue:

  ```
  $ php artisan queue:listen
  ```

- We can specify the connection on which to listen:

  ```
  $ php artisan queue:listen connection
  ```

- We can specify multiple connections in the order of their priority:

  ```
  $ php artisan queue:listen important,not-so-important
  ```

The `queue:listen` command has to run in the background in order to process the jobs as they arrive from the queue. To make sure that it runs permanently, you have to use a process control system such as forever (`https://github.com/nodejitsu/forever`) or supervisor (`http://supervisord.org/`).

# Getting notified when a job fails

To get notified when a job fails, we use the following functions and commands:

- The following event listener is used for finding the failed jobs:

```
Queue::failing(function($job, $data) {
  // Send email notification
});
```

- Any of the failed jobs can be stored in a database table and viewed with the following commands:

```
$ php artisan queue:failed-table // Create the table
$ php artisan queue:failed // View the failed jobs
```

# Queues without background processes

Push queues do not require a background process but they only work with the iron.io driver. After signing up for an account on `iron.io` and adding your credentials to `app/config/queue.php`, you use them by defining a POST route that receives all the incoming jobs. This route calls `Queue::marshal()`, which is the method responsible for firing the correct job handler:

```
Route::post('queue/receive', function() {
  return Queue::marshal();
});
```

This route then needs to be registered as a subscriber with the `queue:subscribe` command:

```
$ php artisan queue:subscribe queue_name http://yourapp.example.com/
queue/receive
```

Once the URL is subscribed on `http://iron.io/`, any newly created jobs with `Queue::push()` will be sent from Iron back to your application via a POST request.

# Where to go next?

The following is a list of the resources and sites that you can visit to keep up with the latest changes in Laravel:

- `http://twitter.com/laravelphp` on Twitter for regular updates
- `http://laravel.com/docs` for the complete documentation
- `http://laravel.com/api` for the browsable API
- `http://laravel.io` for weekly updates and the occasional podcast
- `http://laracasts.com` for screencast tutorials

# Index

**frameworks**
  homemade tools, limitations  8
  Laravel, using  8
  need for  8

## G

**generate:migration command  78**
**generators**
  generators, speeding up with  77
  HTML forms, generating  78
  migrations, generating  78
**getAuthIdentifier() method  50**
**getAuthPassword() method  50**
**getCurrentPage() method  66**
**getDates() method  106**
**GET method  31**
**guard() method  65**

## H

**homemade tools**
  limitations  8
**HTML forms**
  generating  78
**HTTP foundation  9**
**HTTPS**
  using  60

## I

**inflection functions  101**
**Input::get() method  45**
**intended() method  53**
**interdependent classes**
  testing, in isolation  66
**interfaces  10**
**internals**
  fiddling with  75

## J

**job**
  creating  106
  executing  107
  pushing, onto queue  106
**job fail**
  notifying  108

**JSON serialization features**
  using  96

## L

**Laravel**
  controllers  14
  exploring  17, 18
  features  11-15
  models  14
  moving, to version 4  18, 19
  templates  14
  URL  108
  using  8
  views  14
**Laravel application**
  application container  17
  creating  24
  request lifecycle  17
  structure  16, 17

## M

**mass-assignment**
  using  59
**master view**
  creating  40, 41
**method variable  44**
**migrations**
  generating  78
**missing method  35**
**missing routes**
  identifying  35
**mocks  66**
**Model-View-Controller (MVC)  9**

## N

**namespaces  10**
**nullable() method  38**

## O

**objects**
  cleaning up  65
**overloading  10**
**overview page  42**

## P

**package installation guide**
 URL  79
**packages**
 finding  25, 26
 installing  25, 26
 URL  25
**Packagist**
 URL  22
**PATH variable  67**
**PHP**
 embracing  10
 prettifying  13
**PHP application**
 developing  9
 HTTP foundation  9
**PHP classes**
 including  94, 95
**PHPUnit**
 URL  65
**phpunit command  67, 68**
**PHPUnit_Framework_TestCase class  62**

## Q

**queue**
 job, pushing onto  106
 listening to  107
 without background processes  108
**Queue::marshal() method  108**
**queue:listen command  107**
**queue:subscribe command  108**

## R

**raw method  59**
**Read-Eval-Print Loop (REPL)  75**
**Redirect::guest() method  53**
**redirections**
 handling  35
**Redirect object  35**
**relationships  30**
**rendered views**
 inspecting  71
**render method  83**
**request lifecycle  17**
**resource controllers  14**

**Response object  35**
**route parameters**
 restricting  33, 34
**routes**
 authenticating  52,-56
 cat page, adding  44-47
 cat page, deleting  44-47
 cat page, displaying  43
 cat page, editing  44-47
 missing routes, identifying  35
 overview page  42
 redirections, handling  35
 route parameters, restricting  33, 34
 views, returning  35, 36
 writing  33
**routes command  74**
**run() method  38**

## S

**scene**
 preparing  65
**setUp() method  65-69**
**shorter array syntax  10**
**single command**
 deploying  79
**SQL injection**
 avoiding  59
**standard controllers  14**
**Str::endsWith() method  100**
**string() method  38**
**Str object  100**

## T

**tearDown() method  65**
**test anatomy  62, 63**
**TestCase class  62**
**testing**
 benefits  62
**third-party commands**
 installing  76
 single command, deploying  79
 workflow, speeding up with generators  77
**tinker command  93**
**toArray() method  82**
**toJson() method  10**
**transformation functions  100**

## U

**unit testing**
  about  64
  with PHPUnit  64
**unit testing, PHPUnit**
  assertions  64
  exceptions, expecting  65
  interdependent classes, testing in isolation
     66
  objects, cleaning up  65
  scene, preparing  65
**Unix**
  Composer, installing on  23
**update() method  45**
**user-friendly timestamps**
  outputting  105
**user input**
  validating  56, 57
**user model**
  creating  49, 50
**users**
  authenticating  49
  impersonating  69

**users, authenticating**
  database schema, creating  50-52
  routes, authenticating  52-56
  user input, validating  56, 57
  user model, creating  49, 50
  views, authenticating  52-56

## V

**vendor-cleanup command  80**
**View::make method  36**
**View object  35**
**views**
  authenticating  52-56
  returning  35, 36

## W

**Windows**
  Composer, installing on  24
**with() method  42**
**withTrashed() method  90**
**workflow**
  speeding up, with generators  77

**Thank you for buying**
# Getting Started with Laravel 4

## About Packt Publishing

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

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

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

## About Packt Open Source

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

## Writing for Packt

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

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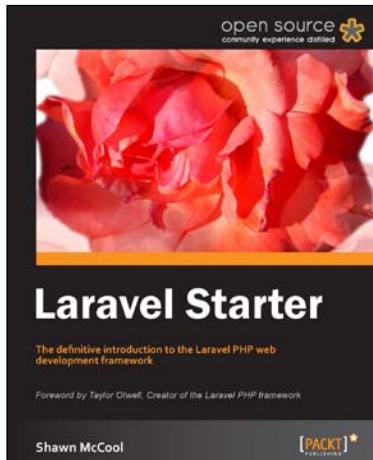
## Instant Laravel Starter

ISBN: 978-1-78216-090-8    Paperback: 64 pages

The definitive introduction to the Laravel PHP web development framework

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.

2. Create databases using Laravel's migrations

3. Learn how to implement powerful relationships with Laravel's own "Eloquent" ActiveRecord implementation
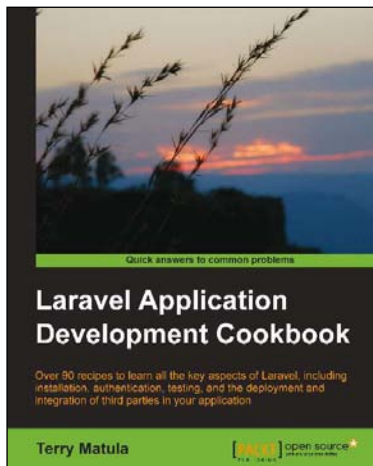
## Laravel Application Development Cookbook

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

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

1. Install and set up a Laravel application and then deploy and integrate third parties in your application

2. Create a secure authentication system and build a RESTful API

3. Build your own Composer Package and incorporate JavaScript and AJAX methods into Laravel

Please check **www.PacktPub.com** for information on our titles
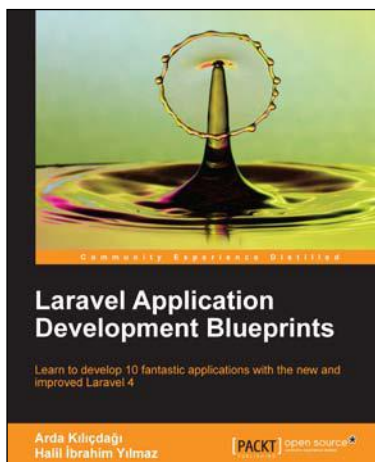
## Laravel Application Development Blueprints

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

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

1. Learn how to integrate third-party scripts and libraries into your application

2. With different techniques, learn how to adapt different methods to your needs

3. Expand your knowledge of Laravel 4 so you can tailor the sample solutions to your requirements
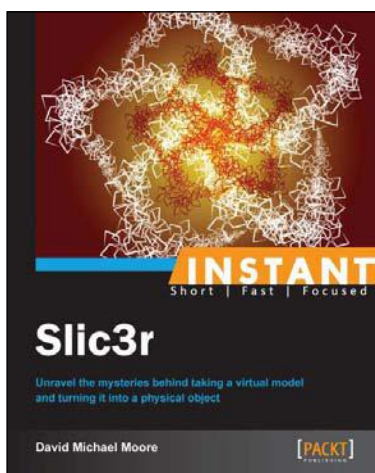
## Instant Slic3r

ISBN: 978-1-78328-497-9          Paperback: 68 pages

Unravel the mysteries behind taking a virtual model and turning it into a physical object

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results

2. Use Slic3r to make your printed objects the best quality possible

4. Make Slic3r work for you, automating tasks and doing post processing on Slic3r output

Please check **www.PacktPub.com** for information on our titles