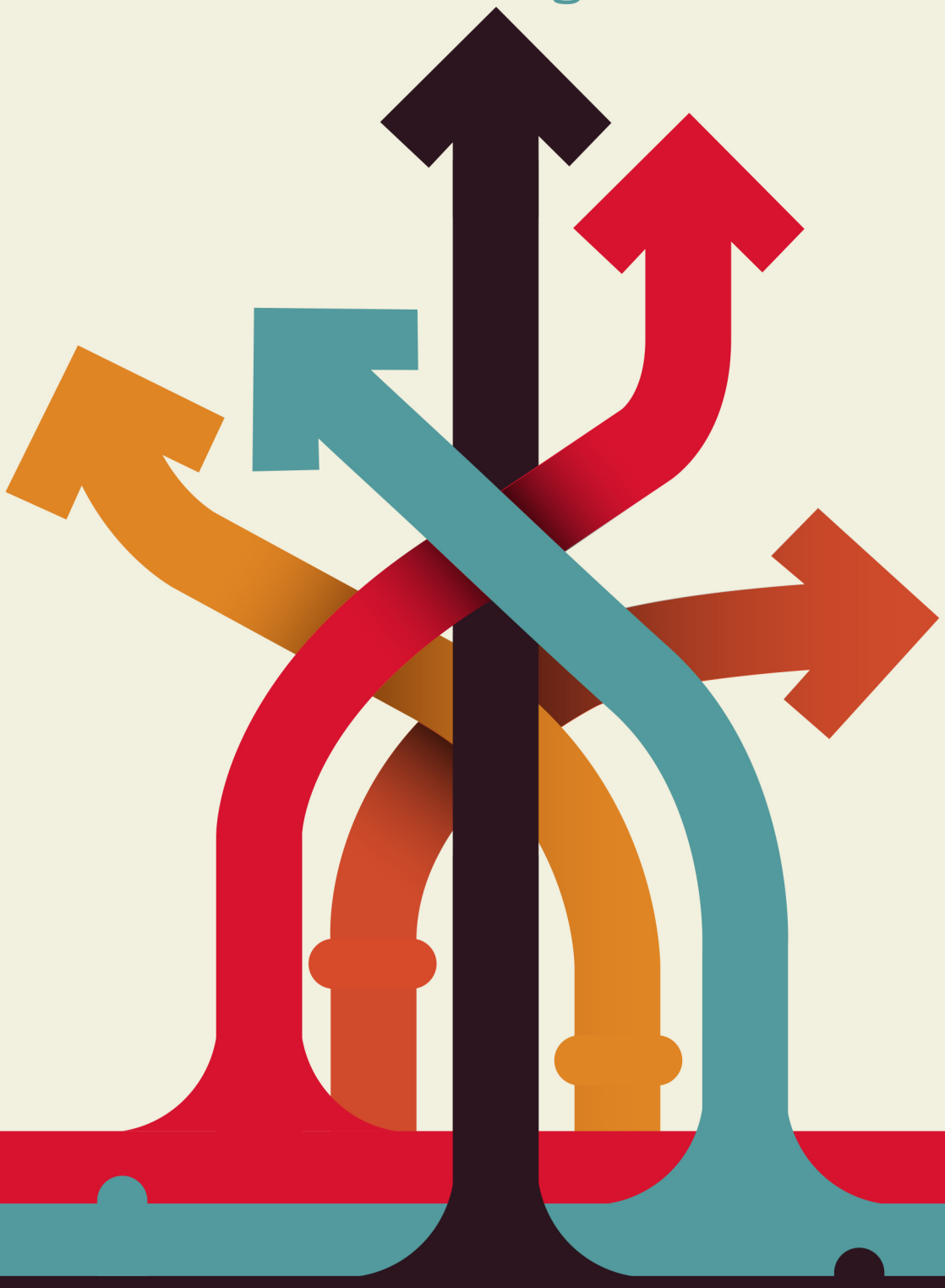
EASY LARAVEL 5

A Hands On Introduction Using a Real-World Project



W. JASON GILMORE

easylaravelbook.com

Easy Laravel 5

A Hands On Introduction Using a Real-World Project

W. Jason Gilmore

This book is for sale at http://leanpub.com/easylaravel

This version was published on 2018-02-16

Also By W. Jason Gilmore

Easy Active Record for Rails Developers

Easy E-Commerce Using Laravel and Stripe

Easy React

Dedicated to The Champ, The Princess, and Little Winnie. Love, Daddy

Contents

Introduction

I've spent the vast majority of my professional career (20 years and counting) immersed in the PHP language. During this time I've written eight PHP-related books, including a few bestsellers. Along the way I've worked on dozens of PHP-driven applications for clients ranging from unknown startups to globally-recognized companies, penned hundreds of articles about PHP and web development for some of the world's most popular print and online publications, and personally trained hundreds of developers on various PHP-related topics. In short, over the span of two decades I've pretty much seen it all when it comes to PHP.

So it might come as a surprise to some that I've never been more immersed in the language than right now. The PHP community and project ecosystem has never been stronger than it is today, offering an incredible number of libraries, frameworks, and tools which allow PHP developers to build more complicated web applications faster than they ever have before. And at least at the time of this writing there is no more popular PHP project on the planet than the *Laravel framework*.

Over the past several years I've worked on multiple large Laravel projects for a variety of clientele. Among others these projects include a REST API for an incredibly popular iOS and Android app, an e-commerce application for selling subscription-based services, and a huge intranet application for a South American agricultural concern. I can say without hesitation that these projects have ranked among the most entertaining and fulfilling in my career, and that sentiment has largely to do with the incredible power and productivity Laravel bestows upon its users.

This book summarizes all of the hard-won knowledge and experience I've amassed building these Laravel projects, and indeed that accrued building web applications of all shapes and sizes over the past two decades. By its conclusion, you'll have gained a well rounded understanding of Laravel's many features, and along the way will have been introduced to many best practices pertaining to code organization, testing, and deployment. I can't wait to get started!

Introducing the HackerPair Companion Project

Too many programming tutorials skew far more heavily in favor of academic exercises than real-world practicalities. Not so in this book. A significant amount of the material found herein is based upon the development of a project called HackerPair (http://hackerpair.com) which you can interact with *right now* by heading over to the HackerPair website (http://hackerpair.com).

HackerPair incorporates many, if not all, of the Laravel features you'll want to be acquainted with when building your own applications. I'll highlight just a few of the features here. Keep in mind however that not all of these features are currently available in the beta release, although they'll be added soon!

- **Comprehensive Database Seeding**: Many beginning developers tend to skip over the generation of real-world data for the development environment. HackerPair includes extensive data generation scripts (known as seeds) for users, events, categories, and locations.
- **Rigorous Form Integration and Validation**: HackerPair uses the powerful LaravelCollective/HTML package for forms generation, and relies on formalized Laravel procedures for input validation including use of the native Laravel validators and form requests.
- **Extensive Model Relationships**: HackerPair offers numerous examples of model relationships by including features such as event creation (events are owned by users), event favorites (users can favorite many events), event locations (events belong to states, states have many events), and so on.
- **User Authentication and Profile Management**: Laravel offers great off-the-shelf support for user registration and login, however developers will quickly outgrow the defaults. HackerPair extends the registration form to include several additional fields, extensively modifies the default registration and login view formatting, and adds account profile management.
- **Social Login**: In addition to standard user registration and authentication, users can instead opt to login using a third-party service such as GitHub and Twitter.
- **Vue.js Features**: Vue.js is Laravel's de facto JavaScript library. HackerPair includes a number of cool Vue.js features, including AJAX-driven event favoriting, event attendance management, and notifications.
- **Bootstrap 4 Integration**: Although Bootstrap 4 is still in beta at the time of this writing, I wanted to give it a spin and am glad I did. Although I will make clear I'm not exactly a CSS guru, and you'll probably find some of my styling to be repulsive. At any rate, in the book you'll also learn how to integrate Bootstrap 3 and the new Tailwind CSS frameworks.
- **Extensive Automated Testing**: One of my favorite Laravel features is the practically push button automated test integration. The HackerPair project includes extensive testing of numerous aspects of the code, including unit tests, model tests, and integration tests using Laravel Dusk.
- **A REST API**: We want to give developers the chance to build their own cool HackerPair applications, and so have exposed a REST API which allows information about events to be retrieved for display in a variety of formats.

Best of all, all interested Laravel developers are able to peruse and download the HackerPair GitHub repository for free! Head on over to http://github.com/wjgilmore/hackerpair to view the code.

About this Book

This book is broken into 12 chapters, each of which is briefly described below. Remember, the book is currently in beta, which is why not all of these chapters are included in your download! Many are under development and almost complete, so in the coming weeks I'll be regularly pushing up new versions and notifying readers.

Chapter 1. Introducing Laravel

In this opening chapter you'll learn how to create and configure your Laravel project using your existing PHP development environment, a virtual machine known as Laravel Homestead, and a minimal development environment known as Valet (OSX users only). I'll also show you how to configure your environment in order to effectively debug your Laravel applications, and how to extend Laravel's capabilities by installing several popular third-party packages. We'll conclude the chapter with an introduction to PHPUnit, showing you how to create and execute your first automated Laravel test!

Chapter 2. Managing Your Project Controllers, Layout, Views, and Other Assets

In this chapter you'll learn how to create controllers and actions, and define the routes used to access your application endpoints. You'll also learn how to create the pages (views), work with variable data and logic using the Blade templating engine, and reduce redundancy using layouts and view helpers. I'll also introduce Laravel Elixir, a new feature for automating otherwise laborious tasks such as JavaScript transpiling and CSS minification. You'll also learn how to integrate several popular CSS frameworks, including Bootstrap 3 and 4, and Tailwind, and how to use Laravel Dusk for integration testing.

Chapter 3. Talking to the Database

In this chapter we'll turn our attention to the project's data. You'll learn how to integrate and configure the database, manage your database schema using migrations, and easily populate your database using seeds. From there we'll move on to creating models, and how to query the database through these models using the Eloquent object relational mapper. I'll also introduce the concept of resourceful controllers, and we'll generate a controller which will be used to view and manage the example project's events. You'll also learn how how to use Laravel's Query Builder to query the database when Eloquent isn't possible or practical.

Chapter 4. Customizing Your Models

Laravel models are incredibly powerful tools, and can be customized in a variety of ways to meet your project's specific needs. In this chapter you'll learn how to override model defaults to create custom accessors and mutators, add instance methods, and use scopes to easily filter database results with minimal code redundancy. You'll also learn how to create sluggable URLs using the eloquent-sluggable package. The chapter concludes with an introduction to testing your models using Laravel's amazing database-specific test features.

Chapter 5. Creating, Updating, and Deleting Data

Chapters 3 and 4 were primarily focused upon the many different ways you can query the database. In this chapter we'll turn our attention to creating, updating, and deleting data. In addition to a review of the Eloquent syntax used to perform these tasks, we'll continue building out the resourceful controller created in chapter 3. You'll also learn how to incorporate flash notifications into your controllers and views to keep users updated regarding request outcomes, and how to use Laravel Dusk to test your forms.

Chapter 6. Validating User Input

For reasons of simplicity, chapter 5 focused exclusively on what it must be like to live in a world in which error prone or malicious users didn't exist. That is to say I momentarily punted on the matter of user input validation. But data validation is so crucial to successful web application development that it can be put off no longer, and so this chapter is devoted entirely to the topic. In this chapter you'll learn all about Laravel's native validators, and how to incorporate form requests into your project to add form validation while ensuring your controller code remains lean.

Chapter 7. Creating and Managing Model Relationships

Building and navigating table relations is a standard part of the development process even when working on the most unambitious of projects, yet this task is often painful when working with many web frameworks. Fortunately, using Laravel it's easy to define and traverse these relations. In this chapter I'll show you how to define, manage, and interact with one-to-one, one-to-many, many-to-many, has many through, and polymorphic relations.

Chapter 8. Sending E-mails

Whether for requiring newly registered users to confirm their e-mail address, or notifying event attendees of scheduling changes, web applications such as HackerPair rely heavily on using e-mail as an efficient means of communication. In this chapter you'll learn about Laravel's `Mailable` class, a fantastic solution for generating e-mails within your application. You'll also learn how to test e-mail generation and delivery in a sane fashion. Just for added measure, I'll walk you through the steps I took to incorporate a contact form into HackerPair, which when submitted, sends inquiring users' messages and contact details to a support address.

Chapter 9. Authenticating and Managing Your Users

Most modern applications offer user registration and preference management features in order to provide customized, persisted content and settings. In this chapter you'll learn how to integrate user registration, login, and account management capabilities into your Laravel application. I'll also show you how to add social authentication to your application, allowing users to authenticate using a variety of popular OAuth providers such as Twitter, Facebook, and GitHub.

Chapter 10. Creating an Administration Console

Most web applications incorporate a restricted administration console accessible by the project developers and support team. In this chapter I'll show you an easy solution for designating certain users as administrators, and how to grant access to a restricted console using prefixed route grouping and custom middleware.

Chapter 11. Introducing Vue.js

Vue.js[1] has become the Laravel community's de facto JavaScript library, and for good reason; it shares many of the practical, productive attributes Laravel developers have come to love. Chapter 13 introduces Vue.js' fundamental features, and shows you how to integrate highly interactive and eye-appealing interfaces into your Laravel application.

Chapter 12. Creating an Application API

These days a web interface is often only one of several available vehicles for interacting with the underlying data. Popular services such as GitHub, Amazon, and Google also offer an API (Application Programming Interface) which allows enterprising developers to dream up and implement new ways to view, mine, and update these companies' vast data stores. In this chapter you'll learn how to create your own API, and provide registered users with an API key which they'll use to authenticate when interacting with the API.

About W. Jason Gilmore

I'm W. Jason Gilmore[2], a software developer, consultant, and bestselling author. I've spent much of the past 17 years helping companies of all sizes build amazing technology solutions. Recent projects include an API for one of the world's highest volume robocall blockers, a SaaS for the interior design and architecture industries, an intranet application for a major South American avocado farm, an e-commerce analytics application for a globally recognized publisher, and a 10,000+ product online store for the environmental services industry.

I'm the author of eight books, including the bestselling *Beginning PHP and MySQL, Fourth Edition*, *Easy E-Commerce Using Laravel and Stripe* (with co-author and Laravel News founder Eric L. Barnes), and *Easy Active Record for Rails Developers*.

Over the years I've published more than 300 articles within popular publications such as Developer.com, JSMag, and Linux Magazine, and instructed hundreds of students in the United States and Europe. I'm also cofounder of the wildly popular CodeMash Conference[3], the largest multi-day developer event in the Midwest.

[1] http://vuejs.org/

[2] http://www.wjgilmore.com

[3] http://www.codemash.org

Away from the keyboard, you'll often find me playing with his kids, thinking about chess, and having fun with DIY electronics.

I love talking to readers and invite you to e-mail me at wj@wjgilmore.com.

Errata and Suggestions

Nobody is perfect, particularly when it comes to writing about technology. I've surely made some mistakes in both code and grammar, and probably completely botched more than a few examples and explanations. If you would like to report an error, ask a question or offer a suggestion, please e-mail me at wj@wjgilmore.com.

Chapter 1. Introducing Laravel

Laravel is a web application framework that borrows from the very best features of other popular framework solutions, among them Ruby on Rails and ASP.NET MVC. For this reason, if you have any experience working with other frameworks then I'd imagine you'll make a pretty graceful transition to Laravel. Newcomers to framework-driven development will have a slightly steeper learning curve due to the introduction of new concepts. I promise Laravel's practical and user-friendly features will make your journey an enjoyable one.

In this chapter you'll learn how to install the Laravel Installer and how to manage your projects using either the Homestead virtual machine or Valet development environment. We'll also create the companion project which will serve as the basis for introducing new concepts throughout the remainder of the book. I'll also introduce you to several powerful debugging and development tools crucial to efficient Laravel development. Finally, you'll learn a bit about Laravel's automated test environment, and how to write automated tests to ensure your application is operating precisely as expected.

Installing the Laravel Installer

A Laravel package known as the Laravel Installer is indispensable for generating new Laravel project skeletons. The easiest way to install Laravel is via PHP's Composer package manager (https://getcomposer.org). If you're not already using Composer to manage your PHP application dependencies, it's easily installed on all major platforms (OS X, Linux, and Windows among them), so head over to the Composer website and take care of that first before continuing.

With Composer installed, run the following command to install Laravel:

```
1  $ composer global require laravel/installer
```

After installing the Laravel installer, you'll want to add the directory `~/.composer/vendor/bin` to your system path so you can execute the `laravel` command anywhere within the operating system. The process associated with updating the system path is operating system-specific but a quick Google search will produce all of the instructions you need.

With the system path updated, open a terminal and execute the following command:

```
1  $ laravel -V
2  Laravel Installer 1.4.1
```

With that done, let's create the book's companion project skeleton. To generate a Laravel 5.5 project or newer you'll need to be running PHP 7 or newer on your development machine. Also, for reasons that will be apparent later in this chapter, I suggest creating a new directory in your development machine's account home directory named `code`. You don't have to do this, and can certainly manage your Laravel projects anywhere you desire within the file system, however I'll be referring to this directory throughout the next few sections and so it would probably save you some additional thinking to just play along:

```
$ cd ~
$ mkdir code
$ cd code
```

Now that you're inside the `code` directory, let's create the project skeleton. You'll primarily use the `laravel` CLI to generate new Laravel projects, which you can do with the `new` command:

```
$ laravel new hackerpair
Crafting application...
Loading composer repositories with package information
Installing dependencies (including require-dev) from lock file
Package operations: 68 installs, 0 updates, 0 removals
  - Installing doctrine/inflector (v1.2.0): Loading from cache
...
> @php artisan package:discover
Discovered Package: fideloper/proxy
Discovered Package: laravel/tinker
Package manifest generated successfully.
Application ready! Build something amazing
```

If you peek inside the `hackerpair` directory you'll see all of the files and directories which comprise a Laravel application! While I know diving into this code will undoubtedly be a very tantalizing prospect, please be patient and finish reading this chapter in its entirety before getting your hands dirty.

Managing Your Local Laravel Project Hosting Environment

If your development machine is already configured to host PHP applications (and meets a minimum set of requirements itemized here https://laravel.com/docs/master/installation#server-requirements), then you're free to configure your local web server and database to serve the project (use the `public` directory as the project's document root). Even if you've been managing your PHP projects

in this manner for years, I urge you to take this opportunity to at least try one of the local hosting solutions I'll introduce in this chapter.

Like any typical PHP-based web application, Laravel requires a web server such as NGINX or Apache, and in most cases a database such as MySQL or PostgreSQL for hosting application data. Further, modern Laravel applications require PHP 7 or newer. Beyond this, you'll need to update your web server's configuration file to recognize the Laravel application's document root, ensure various required PHP extensions have been installed (see the aforementioned link for a complete set of requirements), and deal with the ongoing system administration-related matters necessary to ensure this software stack plays nicely together. It gets even worse. Your Laravel application may require additional software such as Redis[4] and the npm package manager[5], only adding to the list of third-party technologies you'll have to manage.

In the past dealing with these sorts of distractions was basically a requirement, and along the way a bunch of packaged solutions such as XAMPP (https://www.apachefriends.org/) and MAMP (https://www.mamp.info) were offered as alternatives to manually installing and configuring each part of this stack. In time, a far more convenient and practical solution known as a *virtual machine* came along. A virtual machine is a software-based implementation of a computer that can be run inside the confines of another computer (such as your laptop), or even inside another virtual machine. This is incredible technology, because you can use a virtual machine to run an Ubuntu Linux server on your Windows 10 laptop, or vice versa. Further, it's possible to create a customized virtual machine image preloaded with a select set of software. This image can then be distributed to fellow developers, who can run the virtual machine and take advantage of the custom software configuration. This is precisely what the Laravel developers have done with Homestead[6], a virtual machine which bundles everything you need to get started building Laravel-driven websites.

In this section you'll learn all about Homestead, including how to install and configure it to host your Laravel projects (you can incidentally host all sorts of other PHP projects using Homestead, among them WordPress and Drupal). If you're using OSX, then I recommend you additionally carefully read the subsequent section introducing *Valet*, a streamlined hosting solution which allows you to make new Laravel applications available via your web browser in mere seconds.

Introducing Homestead

Homestead is currently based on Ubuntu 16.04, and includes everything you need to get started building Laravel applications, including PHP 7.1, NGINX, MySQL, PostgreSQL and a variety of other useful utilities such as Redis and Memcached. It runs flawlessly on OS X, Linux and Windows, and the installation process is very straightforward, meaning in most cases you'll be able to begin managing Laravel applications in less than 30 minutes.

[4] http://redis.io/

[5] https://www.npmjs.com/

[6] http://laravel.com/docs/homestead

Installing Homestead

Homestead requires Vagrant[7] and VirtualBox[8] (in lieu of VirtualBox you may use VMware Fusion or Parallels; see the Laravel documentation for more details). User-friendly installers are available for all of the common operating systems, including OS X, Linux and Windows. Take a moment now to install Vagrant and VirtualBox. Once complete, open a terminal and execute the following command:

```
$ vagrant box add laravel/homestead
==> box: Loading metadata for box 'laravel/homestead'
    box: URL: https://vagrantcloud.com/laravel/homestead
This box can work with multiple providers! The providers that it
can work with are listed below. Please review the list and choose
the provider you will be working with.

1) parallels
2) virtualbox
3) vmware_desktop

Enter your choice: 2
==> box: Adding box 'laravel/homestead' (v4.0.0) for provider: virtualbox
    box: Downloading: https://vagrantcloud.com/laravel/boxes/homestead/...
==> box: Successfully added box 'laravel/homestead' (v4.0.0) for 'virtualbox'!
```

Throughout the book I'll use the $ symbol to represent the terminal prompt.

This command installs the Homestead *box*. A box is just a term used to refer to a Vagrant package. Packages are the virtual machine images that contain the operating system and various programs. The Vagrant community maintains hundreds of different boxes useful for building applications using a wide variety of technology stacks, so check out this list of popular boxes[9] for an idea of what else is available.

Once the box has been added, you'll next want to install Homestead. To do so, you'll ideally use Git to clone the repository. If you don't already have Git installed you can easily do so by heading over to the Git website[10] or using your operating system's package manager.

Next, open a terminal and enter your home directory:

[7] http://www.vagrantup.com/
[8] https://www.virtualbox.org/
[9] https://vagrantcloud.com/discover/popular
[10] https://git-scm.com/downloads

```
$ cd ~
```

Then use Git's `clone` command to clone the Homestead repository:

```
$ git clone https://github.com/laravel/homestead.git Homestead
Cloning into 'Homestead'...
remote: Counting objects: 1497, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 1497 (delta 0), reused 0 (delta 0), pack-reused 1492
Receiving objects: 100% (1497/1497), 241.74 KiB | 95.00 KiB/s, done.
Resolving deltas: 100% (879/879), done.
Checking connectivity... done.
```

If you're not familiar with Git, what you've just done is downloaded the Homestead project repository, which means you not only now possess a copy of the code, but additionally the entire project's history of changes and releases. When you cloned the repository in this fashion, you're currently using what is known as the `master` branch, which may not always be stable. Logically you'll want to use the latest stable release, and so you'll want to `check it out`. At the time of this writing that latest stable release is `v6.5.0`. Check that version out like so:

```
$ cd Homestead
$ git checkout v6.5.0
```

When you'll run this command you'll receive a scary sounding response about being in a "detached head" state. This is irrelevant since you're just going to use Homestead as an end user, so don't worry about it.

You'll see this has resulted in the creation of a directory named `Homestead` in your home directory which contains the repository files. Next, you'll want to enter this directory and execute the following command:

```
$ bash init.sh
Homestead initialized!
```

If you're on Windows you'll instead want to run the following command:

```
$ init.bat
```

Running this script added a few new files to your `Homestead` directory, including `Homestead.yaml`, `after.sh`, and `aliases`. While all three are useful configuration files, for the purposes of just running our newly created project inside the virtual machine we'll only worry about `Homestead.yaml` for now.

Configuring the Homestead.yaml File

Open the `Homestead.yaml` file and you'll find the following contents:

```
---
ip: "192.168.10.10"
memory: 2048
cpus: 1
provider: virtualbox

authorize: ~/.ssh/id_rsa.pub

keys:
    - ~/.ssh/id_rsa

folders:
    - map: ~/code
      to: /home/vagrant/code

sites:
    - map: homestead.test
      to: /home/vagrant/code/public
```

If you happen to be using VMware Fusion or Parallels then you'll want to update the `provider` property to either `vmware_fusion` or `parallels`, respectively.

Next you'll want to ensure the `authorize` property is pointed to your public SSH key. If you're running Linux or OS X, then chances are high you've generated a public key at some point in the past, and the default `~/.ssh/id_rsa` path is correct. If you're running Linux or OS X and haven't yet generated a key pair then you should be able to do so by running the following command:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/wjgilmore/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/wjgilmore/.ssh/id_rsa.
Your public key has been saved in /Users/wjgilmore/.ssh/id_rsa.pub.
```

When using keys for reasons of automation, there's really no need to protect the key with a passphrase and so when prompted to provide one just press enter. However, there are very sound reasons for using a passphrase when using keys for other purposes, so be sure to read up on the matter if you're new to key-based authentication!

Windows users don't currently have native key generation capabilities. To my understanding the most straightforward way to generate keys is via the popular PuTTY SSH client. You can learn more about using PuTTY to do so via this link[11].

[11] https://docs.joyent.com/public-cloud/getting-started/ssh-keys/generating-an-ssh-key-manually/manually-generating-your-ssh-key-in-windows

With the `provider` and `authorize` properties sorted, we'll turn attention to the `folders` and `sites` properties. These properties cause quite a bit of confusion among newcomers so pay particular attention to the following explanation.

The `folders` property makes known to the virtual machine the location of one or more applications *residing on your local file system*, and identifies the location on the virtual machine to which these application files should be synchronized. Consider the following default mapping:

```
folders:
    - map: ~/code
      to: /home/vagrant/code
```

This means anything residing in a directory named `code` which is found in your home directory will *automatically* be synchronized with the virtual machine's file system, specifically within the directory `/home/vagrant/code`. You're free to synchronize multiple projects by defining additional `map-to` pairs like so:

```
folders:
    - map: ~/code
      to: /home/vagrant/code
    - map: ~/code/hackerpair
      to: /home/vagrant/code/hackerpair
    - map: ~/code/wjgilmore
      to: ~/code/wjgilmore
```

Further, you're not required to use `code` as the local base project directory! For instance if you manage projects in your `Documents/Software` directory, then just change the `folders` property accordingly:

```
folders:
    - map: ~/Documents/software/hackerpair
      to: /home/vagrant/code/hackerpair
    - map: ~/Documents/software/wjgilmore
      to: ~/code/wjgilmore
```

Just keep in mind you don't want to change the `to` reference to the `~/code/` path prefix, because this is where Homestead expects the files to reside. You can actually change this default but there's certainly no reason to do so now.

With your project file system mappings defined, it's time to tell Homestead how the web server should recognize those project directories. This is where the `sites` property comes in. Referring back to the following `folders` configuration:

```
- map: ~/code/hackerpair
  to: /home/vagrant/code/hackerpair
```

We're telling Homestead the `hackerpair` project root directory will be synchronized to `home/vagrant/code/hackerpair`. But this is *not* where a Laravel project's web (also known as *document*) root resides! Laravel project's are always served from the `public` directory, meaning Homestead's web server (known as NGINX) needs to point to that `public` directory when responding to requests. So in the `sites` property you'll want to define the `hackerpair` project like so:

```
folders:
    - map: ~/code/hackerpair
      to: /home/vagrant/code/hackerpair

sites:
    - map: hackerpair.test
      to: /home/vagrant/code/hackerpair/public
```

With these changes in place, you'll be able to reference `http://hackerpair.test` in your browser, and the HackerPair application will be served via Homestead! Not quite, because one minor but important detail remains; you need to tell your local operating system how to resolve references to the `hackerpair.test` domain, because otherwise your browser will reach out to the actual network in an effort to find this site, which in actuality only exists locally.

To ensure proper resolution, you'll need to update your development machine's `hosts` file. If you're running OSX or Linux, this file is found at `/etc/hosts`. If you're running Windows, you'll find the file at `C:\Windows\System32\drivers\etc\hosts`. Open up this file and add the following line:

```
192.168.10.10  hackerpair.test
```

Save these changes, and then run the following command from within your `Homestead` directory:

```
$ vagrant up
Bringing machine 'homestead-7' up with 'virtualbox' provider...
==> homestead-7: Importing base box 'laravel/homestead'...
==> homestead-7: Matching MAC address for NAT networking...
==> homestead-7: Checking if box 'laravel/homestead' is up to date...
==> homestead-7: Setting the name of the VM: homestead-7
==> homestead-7: Clearing any previously set network interfaces...
==> homestead-7: Preparing network interfaces based on configuration...
...
$
```

Your Homestead virtual machine is up and running! With that done, open your browser and navigate to `http://hackerpair.test`. You should see the words "Laravel 5" just as depicted in the following screen shot.

The Laravel 5.5 Splash Screen

Congratulations! From here on out any changes you make to the project will be immediately reflected via the browser. However, there still remains plenty to talk about regarding Homestead and virtual machine management. In the sections that follow I discuss several important matters pertaining to this topic. For the moment I suggest jumping ahead to the section "Perusing the HackerPair Skeleton Code" and returning to the below sections later.

Managing Your Virtual Machine

There are a few administrative tasks you'll occasionally need to carry out regarding management of your virtual machine. For example, if you'd like to shut down the virtual machine you can do so using the following command:

```
1 $ cd ~/Homestead
2 $ vagrant halt
3 ==> homestead-7: Attempting graceful shutdown of VM...
4 $
```

To later boot the machine back up, you can execute `vagrant up` as we did previously:

```
1 $ vagrant up
```

If you'd like to delete the virtual machine (including all data within it), you can use the `destroy` command:

```
$ vagrant destroy
homestead-7: Are you sure you want to destroy the 'homestead-7' VM? [y/N] y
==> homestead-7: Destroying VM and associated drives...
```

I stress executing the `destroy` command this *will delete* not only the virtual machine and also all of its data! Executing this command is very different from shutting down the machine using `halt`. I'm not warning this command will delete your application code, because that is synchronized from your local file system to the virtual machine. It would however delete any data residing in your project databases, since the database is hosted inside the virtual machine.

If you happen to have installed more than one box (it can be addictive), use the `box list` command to display them:

```
$ vagrant box list
laravel/homestead (virtualbox, 4.0.0)
```

Finally, if you make any changes to your `Homestead.yaml` file, you'll need to run the following command in order for Homestead to recognize those changes:

```
$ vagrant reload --provision
```

These are just a few of the many commands available to you. Run `vagrant --help` for a complete listing of what's available:

```
$ vagrant --help
```

SSH'ing Into Your Virtual Machine

Because Homestead is a virtual machine running Ubuntu, you can SSH into it just as you would any other server. For instance you might wish to configure NGINX or MySQL, install additional software, or make other adjustments to the virtual machine environment. If you're running Linux or OS X, you can SSH into the virtual machine using the `ssh` command:

```
$ vagrant ssh
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-92-generic x86_64)
vagrant@homestead:~$
```

Windows users will need to install an SSH client in order to SSH into the Homestead VM. A popular Windows SSH client is PuTTY[12].

In either case, you'll be logged in as the user `vagrant`, and if you list this user's home directory contents you'll see the `Code` directory defined in the `Homestead.yaml` file:

[12] http://www.putty.org/

```
vagrant@homestead:~$ ls
code
```

If you're new to Linux be sure to spend some time nosing around Ubuntu! This is a perfect opportunity to get familiar with the Linux operating system without any fear of doing serious damage to a server because if something happens to break you can always reinstall the virtual machine.

Transferring Files Between Homestead and Your Laptop

If you create a file on a Homestead and would like to transfer it to your laptop, you have two options. The easiest involves SSH'ing into Homestead and moving the file into one of your shared directories, because the file will instantly be made available for retrieval via your laptop's file system. For instance if you're following along with the `hackerpair` directory configuration, you can SSH into Homestead, move the file into `/home/vagrant/hackerpair`, and then logout of SSH. Then using your local terminal, navigate to `~/code/hackerpair` and you'll find the desired file sitting in your local `hackerpair` root directory.

Alternatively, you can use `sftp` to login to Homestead, navigate to the desired directory, and transfer the file directly:

```
$ sftp -P 2222 vagrant@127.0.0.1
Connected to 127.0.0.1.
sftp>
```

Connecting to Your Database

Although this topic won't really be relevant until we discuss databases in chapter 3, this nonetheless seems a logical place to show you how to connect to your project's Homestead database. If you return to `Homestead.yaml`, you'll find the following section:

```
databases:
    - homestead
```

This section is used to define any databases you'd like to be automatically created when the virtual machine is first booted (or re-provisioned; more about this in the next section). As you can see, a default database named `homestead` has already been defined. You can sign into this database now by SSH'ing into the machine and using the `mysql` client:

```
$ vagrant ssh
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-92-generic x86_64)
```

After signing in, enter the database using the `mysql` client, supplying the default username of `homestead` and the desired database (also `homestead`). When prompted for the password, enter `secret`:

```
vagrant@homestead:~$ mysql -u homestead homestead -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

At this point there are no tables in the database (we'll create a few in chapter 3), but feel free to have a look anyway:

```
mysql> show tables;
Empty set (0.00 sec)
```

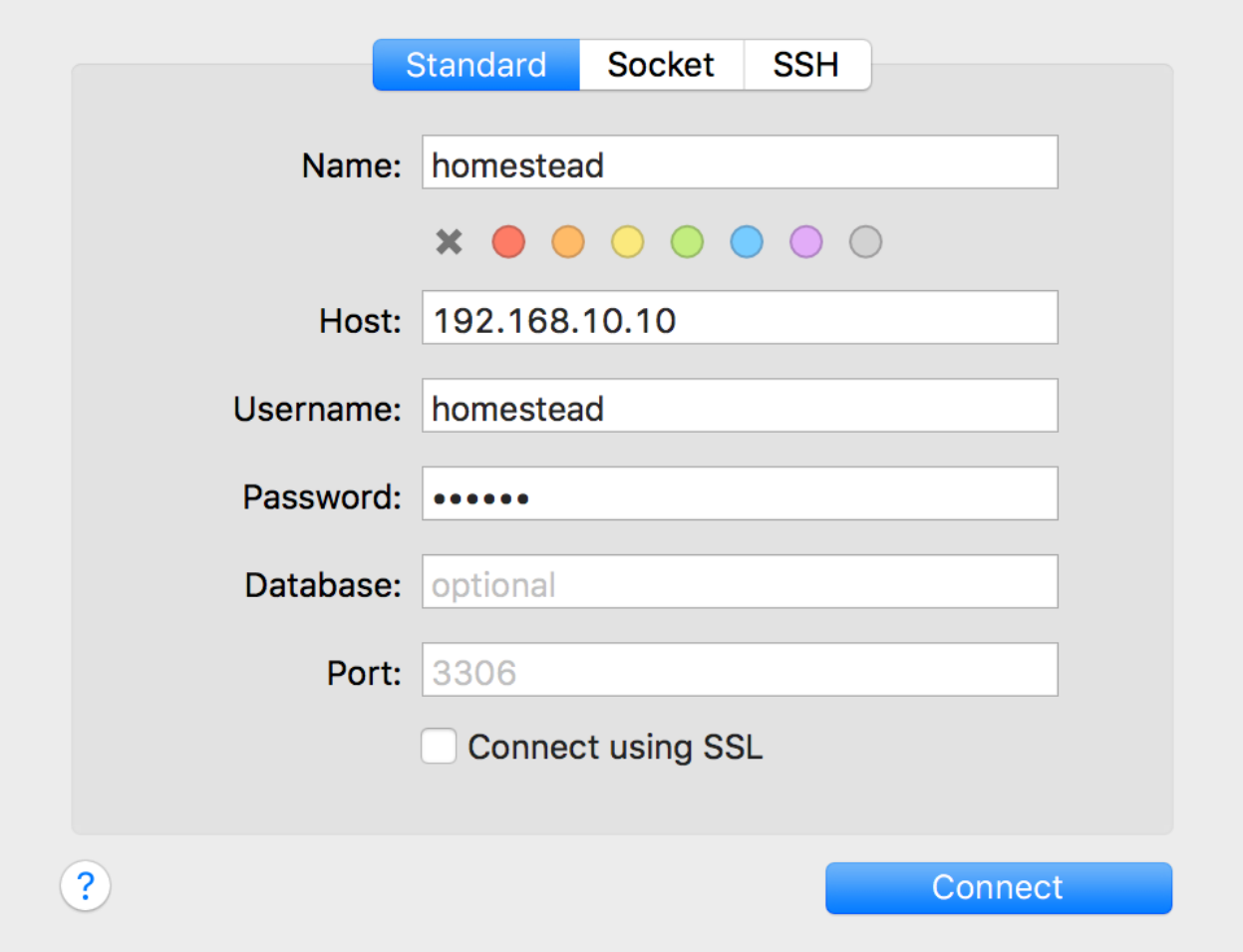
To exit the `mysql` client, execute `exit`:

```
mysql> exit;
Bye
vagrant@homestead:~$
```

Chances are you prefer to interact with your database using a GUI-based application such as Sequel Pro[13] or phpMyAdmin[14]. You'll connect to the `homestead` database like you would any other, by supplying the username (`homestead`), password (`secret`), and the host, which is `192.168.10.10`. For instance, the following screenshot depicts my Sequel Pro connection window:

[13] http://www.sequelpro.com/

[14] https://www.phpmyadmin.net/



The Sequel Pro connection window

You may want to change the name of this default database, or define additional databases as the number of projects you manage via Homestead grows in size. I'll show you how to do this next.

Defining Multiple Homestead Sites and Databases

My guess is you'll quickly become so enamored with Homestead that it will be the default solution for managing all of your Laravel projects. This means you'll need to define multiple projects within the `Homestead.yaml` file. Fortunately, doing so is easier than you think. Check out the following slimmed down version of my own `Homestead.yaml` file, which defines two projects (`hackerpair` and `wjgilmore`):

```
folders:
    - map: ~/code/hackerpair
      to: /home/vagrant/hackerpair
    - map: ~/code/wjgilmore
      to: /home/vagrant/wjgilmore

sites:
    - map: hackerpair.test
      to: /home/vagrant/hackerpair/public
    - map: wjgilmore.test
      to: /home/vagrant/wjgilmore/public

databases:
    - dev_hackerpair
    - dev_wjgilmore
```

Notice how I've also defined two different databases, since each application will logically want its own location to store data.

After saving these changes, you'll want your virtual server to be reconfigured accordingly. If you have never started your virtual server, running `vagrant up` will suffice because the `Homestead.yaml` file had never previously been read. If you've already started the VM then you'll need to force Homestead to *reprovision* the virtual machine. This involves reloading the configuration. To do so, you'll first need to find the identifier used to present the currently running machine:

```
$ vagrant global-status
id       name    provider   state   directory
------------------------------------------------------------------
6f13a59  homestead-7 virtualbox running /Users/wjgilmore/Homestead
```

Copy and paste that `id` value (`6f13a59` in my case), supplying it as an argument to the following command:

```
$ vagrant reload --provision 6f13a59
==> homestead-7: Attempting graceful shutdown of VM...
==> homestead-7: Checking if box 'laravel/homestead' is up to date...
==> homestead-7: Clearing any previously set forwarded ports...
==> homestead-7: Clearing any previously set network interfaces...
==> homestead-7: Preparing network interfaces based on configuration...
...
```

Once this command completes, your latest `Homestead.yaml` changes will be in place!

Introducing Valet

Virtual machines such as Homestead are great, and have become indispensable tools I use on a daily basis. As you've probably gathered from reading the past several pages, Homestead can be overkill for many developers. If you use a Mac and are interested in a no-frills development environment, Laravel offers a streamlined solution called *Valet* which can be configured in mere moments.

Installing Valet

To install Valet you'll need to first install Homebrew (http://brew.sh/), the community-driven package manager for OS X. As you'll see on the home page, Homebrew is very easy to install and should only take a moment to complete. Once done, you'll want to install PHP 7. You can do so by executing the following command:

```
1 $ brew install homebrew/php/php71
```

Next you'll install Valet using Composer. Like Homebrew, Composer (https://getcomposer.org) is a package manager but is specific to PHP development, and is similarly easy to install. With Composer installed, install Valet using the following command:

```
1 $ composer global require laravel/valet
```

Next, add Composer's `bin` directory to your system path. There are a variety of ways to do this but I find the simplest to be editing your home directory's `.bash_profile` file. Open the file in your editor and add the following line to it:

```
1 PATH=$PATH:/~/.composer/vendor/bin
```

Finally, configure Valet by running the following command, which among other things will ensure it always starts automatically whenever you reboot your machine:

```
1 $ valet install
```

Presuming your Laravel applications will use a database, you'll also need to install a database such as MySQL or MariaDB. You can easily install either using Homebrew. For instance, you can install MySQL like so:

```
1 $ brew install mysql
```

After installation completes just follow the instructions displayed in the terminal to ensure MySQL starts automatically upon system boot.

Serving Sites with Valet

With Valet installed and configured, you'll next want to create a directory to host your various Laravel projects. I suggest creating this directory in your home directory; consider calling it something easily recognizable such as `Code` or `Projects`. Enter this directory using your terminal and execute the following command:

```
$ valet park
This directory has been added to Valet's paths.
```

The `park` command tells Valet monitor this directory for Laravel projects, and automatically make a convenient URL available for viewing the project in your browser. For instance, while inside the project directory create a new Laravel project named `hackerpair`:

```
$ laravel new hackerpair
```

Next, if you're using Google Chrome, you'll need to run the following command to change Valet's default use of the `.dev` domain extension to `.test`. I use Chrome for development purposes and so all subsequent URL references will include `.test` however this is just a preference and so you don't need to do this if you're using another browser:

```
$ valet domain test
```

After creating the project, open your browser and navigate to `http://hackerpair.test` and you'll see the project's default splash screen (presented in the following screenshot).

The Laravel splash page

It doesn't get any easier than that!

Perusing the HackerPair Skeleton Code

With the Laravel Installer (and presumably Homestead or Valet) installed and configured, it's time to get our hands dirty! Open a terminal and enter the `hackerpair` project directory. The contents are a combination of files and directories, each of which plays an important role in the functionality of your application so it's important for you to understand their purpose. Let's quickly review the role of each:

- `.env`: Laravel 5 uses the PHP dotenv[15] library to manage your application's configuration variables. You'll use `.env` file as the basis for configuring these settings when working in your development environment. A file named `.env.example` is also included in the project root directory, which should be used as a template from which fellow developers will copy over to `.env` and modify to suit their own needs. I'll talk more about these files and practical approaches for managing your environment settings in the later section, "Configuring Your Laravel Application".
- `.gitattributes`: This file is used by Git[16] to ensure consistent settings across machines, which is useful when multiple developers using a variety of operating systems are working on the same project. You'll find a few default settings in the file; these are pretty standard and you in all likelihood won't have to modify them. Plenty of other attributes are available; Scott Chacon's online book, "Pro Git"[17] includes a section ("Customizing Git - Git Attributes"[18]) with further coverage on this topic.
- `.gitignore`: This file tells Git what files and folders should not be included in the repository. You'll see a few default settings in here, including the `vendor` directory which houses the Laravel source code and other third-party packages, and the `.env` file, which should never be managed in version control since it presumably contains sensitive settings such as database passwords.
- `app`: This directory contains much of the custom code used to power your application, including the models, controllers, and middleware. We'll spend quite a bit of time inside this directory as the book progresses.
- `artisan`: `artisan` is a command-line tool we'll use to rapidly create new parts of your applications such as controllers and models, manage your database's evolution through a great feature known as *migrations*, and interactively debug your application. We'll return to `artisan` repeatedly throughout the book because it is such an integral part of Laravel development.
- `bootstrap`: This directory contains the various files used to initialize a Laravel application, loading the configuration files, various application models and other classes, and define the locations of key directories such as `app` and `public`. Normally you won't have to modify any of the files found in this directory.

[15] https://github.com/vlucas/phpdotenv
[16] http://git-scm.com/
[17] http://git-scm.com/book
[18] http://git-scm.com/book/en/Customizing-Git-Git-Attributes

- `composer.json`: Composer[19] is PHP's de facto package manager, used by thousands of developers around the globe to quickly integrate popular third-party solutions such as Swift Mailer[20] and Doctrine[21] into a PHP application. Laravel heavily depends upon Composer, and you'll use the `composer.json` file to identify the packages you'll like to integrate into your Laravel application. If you're not familiar with Composer by the time you're done reading this book you'll wonder how you ever lived without it. In fact in this introductory chapter alone we'll use it several times to install various useful packages.
- `composer.lock`: This file contains information about the state of your project's installed Composer packages at the time these packages were last installed and/or updated. Like the `bootstrap` directory, you will rarely if ever directly interact with this file.
- `config`: This directory contains several files used to configure various aspects of your Laravel application, such as the database credentials, the cache, e-mail delivery, and session settings.
- `database`: This directory contains the directories used to house your project's database migrations and seed data (migrations and database seeding are both introduced in Chapter 3).
- `package.json`: This file is used to manage locally installed npm (JavaScript) packages. If you look inside the file you'll see references to a number of packages, including jQuery, bootstrap-sass, Laravel Mix, and Vue, among others. Even if you have no JavaScript experience you'll come to find npm packages to be indispensable by the end of this book.
- `phpunit.xml`: Even trivial web applications should be accompanied by an automated test suite. Laravel leaves little room for excuse to avoid this best practice by automatically configuring your application to use the popular PHPUnit[22] test framework. The `phpunit.xml` is PHPUnit's application configuration file, defining characteristics such as the location of the application tests. We'll return to the topic of testing repeatedly throughout the book.
- `resources`: The `resources` directory contains your project's views, localized language files, and raw assets such as Vue components and Sass files.
- `public`: The `public` directory serves as your application's web root directory, housing the `.htaccess`, `robots.txt`, and `favicon.ico` files, in addition to a file named `index.php` that is the *first* file to execute when a user accesses your application. This file is known as the *front controller*, and it is responsible for loading and executing the application. It's because the `index.php` file serves as the front controller that you needed to identify the `public` directory as your application's root directory when configuring `Homestead.yaml` earlier in this chapter.
- `routes`: The `routes` directory is new to 5.3. It replaces the old `app/Http/routes.php` file, and separates your application's routing definitions into four separate files: `api.php`, `channels.php`, `console.php`, and `web.php`. Collectively, these files determine how your application responds to different endpoints. We'll return to these files repeatedly throughout the book, beginning in Chapter 2.
- `server.php`: The `server.php` file can be used to bootstrap your application for the purposes of serving it via PHP's built-in web server. While a nice feature, Homestead and Valet offer a

[19] https://getcomposer.org
[20] http://swiftmailer.org/
[21] http://www.doctrine-project.org/
[22] http://phpunit.de/

far superior development experience and so you can safely ignore this file and feature.
- `storage`: The `storage` directory contains your project's cache, session, and log data.
- `tests`: The `tests` directory contains your project tests. Testing is a recurring theme throughout this book, and thanks to Laravel's incredibly simple test integration features I highly encourage you to follow along closely with the examples provided in these sections.
- `vendor`: The `vendor` directory is where the Laravel framework code itself is stored, in addition to any other third-party code. You won't typically directly interact with anything found in this directory, instead doing so through the Composer interface.
- `webpack.mix.js`: All new Laravel projects include the ability to easily integrate a new feature called *Laravel Mix*. Mix provides a convenient JavaScript-based API for automating various build-related processes associated with your project's CSS, JavaScript, tests, and other assets. I'll introduce Mix in Chapter 2.

Now that you have a rudimentary understanding of the various directories and files comprising a Laravel skeleton application, let's dive a bit deeper into the `config` directory so you have a better understanding of the many different ways in which your application can be tweaked.

Configuring Your Laravel Application

Laravel offers environment-specific configuration, meaning you can define certain behaviors applicable only when you are developing the application, and other behaviors when the application is running in production. For instance you'll certainly want to output errors to the browser during development but ensure errors are only output to the log in production.

Your application's default configuration settings are found in the `config` directory, and are managed in a series of files including:

- `app.php`: The `app.php` file contains settings that have application-wide impact, including whether debug mode is enabled (more on this in a bit), the application URL, timezone, and locale.
- `auth.php`: The `auth.php` file contains settings specific to user authentication, including what model manages your application users, the database table containing the user information, and how password reminders are managed. I'll talk about Laravel's user authentication features in chapter 7.
- `broadcasting.php`: The `broadcasting.php` is used to configure the event broadcasting feature, which is useful when you want to simultaneously notify multiple application users of some event such as the addition of a new blog post. I discuss event broadcasting in chapter 11.
- `cache.php`: Laravel supports several caching drivers, including filesystem, database, memcached, redis, and others. You'll use the `cache.php` configuration file to manage various settings specific to these drivers.

- `database.php`: The `database.php` configuration file defines a variety of database settings, including which of the supported databases the project will use, and the database authorization credentials. You'll learn all about Laravel's database support in chapters 3 and 4.
- `filesystems.php`: The `filesystems.php` configuration file defines the file system your project will use to manage assets such as file uploads. Thanks to Laravel's integration with Flysystem[23], support is available for a wide variety of adapters, among them the local disk, Amazon S3, Azure, Dropbox, FTP, Rackspace, and Redis.
- `mail.php`: As you'll learn in chapter 5 it's pretty easy to send an e-mail from your Laravel application. The `mail.php` configuration file defines various settings used to send those e-mails, including the desired driver (a variety of which are supported, among them Sendmail, SMTP, PHP's `mail()` function, and Mailgun). You can also direct mail to the log file, a technique that is useful for development purposes.
- `queue.php`: Queues can improve application performance by allowing Laravel to offload time- and resource-intensive tasks to a queueing solution such as Beanstalk[24] or Amazon Simple Queue Service[25]. The `queue.php` configuration file defines the desired queue driver and other relevant settings.
- `services.php`: If your application uses a third-party service such as Stripe for payment processing or Mailgun for e-mail delivery you'll use the `services.php` configuration file to define any third-party service-specific settings. We'll return to this file throughout the book as new third-party services are integrated into the application.
- `session.php`: It's entirely likely your application will use sessions to aid in the management of user preferences and other customized content. Laravel supports a number of different session drivers used to facilitate the management of session data, including the file system, cookies, a database, the Alternative PHP Cache, Memcached, and Redis. You'll use the `session.php` configuration file to identify the desired driver, and manage other aspects of Laravel's session management capabilities.
- `view.php`: The `view.php` configuration file defines the default location of your project's view files and the renderer used for pagination.

I suggest spending a few minutes nosing around these files to get a better idea of what configuration options are available to you. There's no need to make any changes at this point, but it's always nice to know what's possible.

Configuring Your Environment

Your application will likely require access to database credentials and other sensitive information such as API keys for accessing third party services. This confidential information should never be shared with others, and therefore you'll want to take care it isn't embedded directly into the

[23] https://github.com/thephpleague/flysystem
[24] http://kr.github.io/beanstalkd/
[25] http://aws.amazon.com/sqs/

code. Instead, you'll want to manage this data within *environment variables*, and then refer to these variables within the application.

Laravel supports a very convenient solution for managing and retrieving these variables thanks to integration with the popular PHP dotenv[26] package. When developing your application you'll define environment variables within the `.env` file found in your project's root directory. The default `.env` file looks like this:

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:7kPp7zGCLzeXbe0CQuWJ1/lsOymtzZhfmkAUryKyHRF=
APP_DEBUG=true
APP_LOG_LEVEL=debug
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

BROADCAST_DRIVER=log
CACHE_DRIVER=file
SESSION_DRIVER=file
SESSION_LIFETIME=120
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
```

[26] https://github.com/vlucas/phpdotenv

These variables can be retrieved anywhere within your application using the `env()` function. For instance, the `config/database.php` is used to define your project's database connection settings (we'll talk more about this file in chapter 3). It retrieves the `DB_HOST`, `DB_DATABASE`, `DB_USERNAME`, `DB_PASSWORD`, and `DB_SOCKET` variables defined within `.env`:

```
'mysql' => [
    'driver' => 'mysql',
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => env('DB_DATABASE', 'forge'),
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
    'unix_socket' => env('DB_SOCKET', ''),
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
    'strict' => true,
    'engine' => null,
],
```

You'll see the `.gitignore` includes `.env` by default. This is because you should *never* manage `.env` in your version control repository! Instead, when it comes time to deploy your application to production, you'll typically define the variables found in `.env` as *server environment variables* which can also be retrieved using PHP's `env()` function. In chapter 9 I'll talk more about managing these variables in other environments.

We'll return to the configuration file throughout the book as new concepts and features are introduced.

Useful Development and Debugging Tools

There are several native Laravel features and third-party tools that can dramatically boost productivity by reducing the amount of time and effort spent identifying and resolving bugs. In this section I'll introduce you to a few of my favorite solutions, and additionally show you how to install and configure the third-party tools.

The debugging and development utilities discussed in this section are specific to Laravel, and do not take into account the many other tools available to PHP in general. Be sure to check out Xdebug[27], and the many tools integrated into PHP IDEs such as Zend Studio[28] and PHPStorm[29].

[27] http://xdebug.org/

[28] http://www.zend.com/en/products/studio

[29] https://www.jetbrains.com/phpstorm/

The dd() Function

Ensuring the `.env` file's `APP_DEBUG` variable is set to `true` is the easiest way to view information about any application errors, because Laravel will dump error- and exception-related information directly to the browser. Sometimes though you'll want to peer into the contents of an object or array even if the data structure isn't causing any particular problem or error. You can do this using Laravel's `dd()`[30] helper function, which will dump a variable's contents to the browser and halt further script execution. For example suppose you defined an array inside a Laravel application and wanted to output its contents to the browser. Here's an example array:

```
$languages = [
    'languages' => [
        'Perl',
        'PHP',
        'Python'
    ]
];
```

You could execute the `dd()` function like so:

```
dd($languages);
```

We haven't yet delved into how to actually add code to your project, so you're probably wondering where this code should go if you want to follow along. Setting up a proper environment for inserting this sort of logic in a natural manner is the subject of another chapter, so for the moment we're going to "cheat" a little and embed the code into our `routes/web.php` routing file. This file is responsible for associating web endpoints (URLs) with corresponding application resources. If you open this file you'll see a single route definition that looks like this:

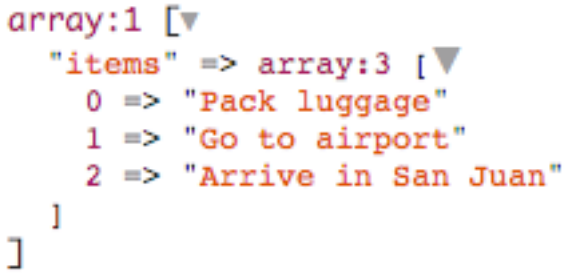
```
Route::get('/', function () {
    return view('welcome');
});
```

This definition ensures that when the application's home page is requested, the template found in `resources/views/welcome.blade.php` is returned. You don't have to understand any of this now because the topic is covered in great detail in the next chapter. For the time being, change this route definition to look like this:

[30] https://laravel.com/docs/master/helpers#method-dd

```
Route::get('/', function () {
    $languages = [
        'languages' => [
            'Perl',
            'PHP',
            'Python'
        ]
    ];
    dd($languages);
    return view('welcome');
});
```

Save the changes and navigate to `http://hackerpair.test` in your browser. Passing `$languages` into `dd()` will cause the array contents to be dumped to the browser window as depicted in the below screenshot.



dd() function output

It is likely at this point you don't know where this code would even be executed. Not to worry! In the chapters to come just keep this and the following solutions in mind so you can easily debug your code once we start building the application.

The Laravel Logger

While the `dd()` helper function is useful for quick evaluation of a variable's contents, taking advantage of Laravel's logging facilities is a more effective approach if you plan on repeatedly monitoring one or several data structures or events without interrupting script execution. Laravel will by default log error-related messages to the application log, located at `storage/logs/laravel.log`. Because Laravel's logging features are managed by Monolog[31], you have a wide array of additional logging options at your disposal, including the ability to write log messages to this log file, set logging levels, send log output to the Chrome console[32] using Chrome Logger[33], or even trigger alerts via e-mail,

[31] https://github.com/Seldaek/monolog
[32] https://developer.chrome.com/devtools/docs/console
[33] http://craig.is/writing/chrome-logger

text messaging, or Slack[34]. Further, if you're using the Laravel Debugbar (introduced later in this chapter) you can easily peruse these messages from the Debugbar's `Messages` tab.

Generating a custom log message is easy, done by embedding one of several available logging methods into the application, passing along the string or variable you'd like to log. Returning to the `$languages` array, suppose you instead wanted to log its contents to Laravel's log:

```
$languages = [
    'languages' => [
        'Perl',
        'PHP',
        'Python'
    ]
];

\Log::debug($languages);
```

After reloading the browser to execute this code, you'll see a log message similar to the following will be appended to `storage/logs/laravel.log`:

```
[2017-11-10 17:59:28] local.DEBUG: array (
  'languages' =>
  array (
    0 => 'Perl',
    1 => 'PHP',
    2 => 'Python',
  ),
)
```

The debug-level message is just one of several at your disposal. Among other levels are info, warning, error and critical, meaning you can use similarly named methods accordingly:

```
\Log::info('Just an informational message.');
\Log::warning('Something may be going wrong.');
\Log::error('Something is definitely going wrong.');
\Log::critical('Danger, Will Robinson! Danger!');
```

[34] https://www.slack.com/

Integrating the Logger and Chrome Logger

When monitoring the log file it's common practice to use the `tail -f` command (available on Linux and OS X; Windows users can use Powershell to achieve a similar behavior) to view any log file changes in real time. You can avoid the additional step of maintaining an additional terminal window for such purposes by instead sending the log messages to the Chrome Logger[35] console, allowing you to see the log messages alongside your application's browser output.

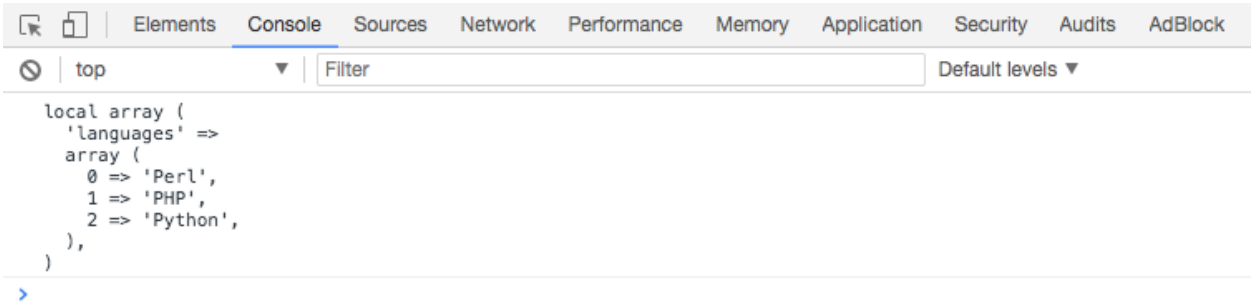
You'll first need to install the Chrome browser and Chrome Logger extension. Once installed, click the Chrome Logger icon once in your browser extension toolbar to enable it for the site. Then add the following anywhere within `bootstrap/app.php`:

```
1  if ($app->environment('local')) {
2      $app->configureMonologUsing(function($monolog) {
3          $monolog->pushHandler(new \Monolog\Handler\ChromePHPHandler());
4      });
5  }
```

You'll want to wrap the configuration logic inside a conditional which ensures your application is running in the local (development) environment, because once deployed you'll want all log messages to be sent to the log file or other third-party logging software. After saving the changes, you can log for instance the `$languages` array just as you did previously:

```
1  \Log::debug($languages);
```

Once executed, the `$languages` array will appear in your browser console as depicted in the below screenshot.



Logging to the Chrome console via the Chrome Logger

[35] http://craig.is/writing/chrome-logger

Using the Tinker Console

You'll often want to test a small PHP snippet or experiment with manipulating a particular data structure, but creating and executing a PHP script for such purposes is kind of tedious. You can eliminate the additional overhead by instead using the tinker console, a command line-based window into your Laravel application. Open tinker by executing the following command from your application's root directory:

```
$ php artisan tinker
Psy Shell v0.8.14 (PHP 7.1.8 â€” cli) by Justin Hileman
>>>
```

Tinker uses PsySH[36], a great interactive PHP console and debugger. PsySH is new to Laravel 5, and is a huge improvement over the previous console. Be sure to take some time perusing the feature list on the PsySH website to learn more about what this great utility can do. In the meantime, let's get used to the interface:

```
>>> $languages = ['Python', 'PHP', 'Perl']
=> [
        "Python",
        "PHP",
        "Perl"
    ]
```

From here you could for instance learn more about how to sort an array using PHP's `sort()` function:

```
>>> sort($languages)
=> true
>>> $languages
=> [
        "Perl",
        "PHP",
        "Python"
    ]
>>>
```

After you're done, type `exit` to exit the PsySH console:

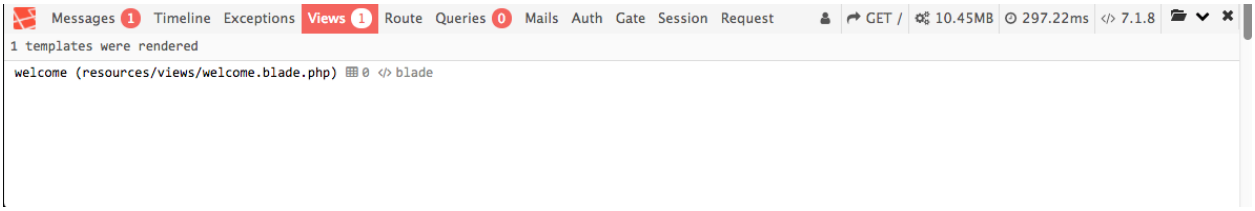
[36] http://psysh.org/

```
>>> exit
Exit:  Goodbye.
$
```

The Tinker console can be incredibly useful for quickly experimenting with PHP snippets, and I'd imagine you'll find yourself repeatedly returning to this indispensable tool. We'll take advantage of Tinker throughout the book to get acquainted with various Laravel features.

Introducing the Laravel Debugbar

It can quickly become difficult to keep tabs on the many different events that are collectively responsible for assembling the application response. You'll regularly want to monitor the status of database requests, routing definitions, view rendering, e-mail transmission and other activities. Fortunately, there exists a great utility called Laravel Debugbar[37] that provides easy access to the status of these events and much more by straddling the bottom of your browser window (see below screenshot).



The Laravel Debugbar

The Debugbar consists of several tabs that when clicked result in context-related information in a panel situated below the menu. These tabs include:

- **Messages**: Use this tab to view log messages directed to the Debugbar. I'll show you how to do this in a moment.
- **Timeline**: Presents a summary of the time required to load the page.
- **Exceptions**: Displays any exceptions thrown while processing the current request.
- **Views**: Provides information about the various views used to render the page, including the layout.
- **Route**: Presents information about the requested route, including the corresponding controller and action.
- **Queries**: Lists the SQL queries executed in the process of serving the request.
- **Mails**: This tab presents information about any e-mails delivered while processing the request.
- **Auth**: Displays information pertaining to user authentication.
- **Gate**: Displays information pertaining to user authorization.

[37] https://github.com/barryvdh/laravel-debugbar

- **Session**: Presents any session-related information made available while processing the request.
- **Request**: Lists information pertinent to the request, including the status code, request headers, response headers, and session attributes.

To install the Laravel Debugbar, execute the following command:

```
$ composer require barryvdh/laravel-debugbar --dev
Using version ^3.1 for barryvdh/laravel-debugbar
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Installing maximebf/debugbar (v1.14.1): Downloading (100%)
  - Installing barryvdh/laravel-debugbar (v3.1.0): Downloading (100%)
maximebf/debugbar suggests installing kriswallsmith/assetic
maximebf/debugbar suggests installing predis/predis (Redis storage)
Writing lock file
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover
Discovered Package: fideloper/proxy
Discovered Package: laravel/tinker
Discovered Package: barryvdh/laravel-debugbar
Package manifest generated successfully.
$
```

Save the changes and install the package configuration to your `config` directory:

```
$ php artisan vendor:publish
Copied File [/vendor/barryvdh/laravel-debugbar/config/debugbar.php]
To [/config/debugbar.php]
Publishing complete.
```

While you don't have to make any changes to this configuration file (found in `config/debugbar.php`), I suggest having a look at it to see what changes are available.

Reload the browser and you should see the Debugbar at the bottom of the page! Keep in mind the Debugbar will only render when used in conjunction with an endpoint that actually renders a view to the browser.

The Laravel Debugbar is tremendously useful as it provides easily accessible insight into several key aspects of your application. Additionally, you can use the Messages panel as a convenient location for viewing log messages. Logging to the Debugbar is incredibly easy, done using the Debugbar facade:

```
\Debugbar::error('Something is definitely going wrong.');
```

Save the changes and reload the home page within the browser. Check the Debugbar's `Messages` panel and you'll see the logged message! Like the Laravel logger, the Laravel Debugbar supports the log levels defined in PSR-3[38], meaning methods for debug, info, notice, warning, error, critical, alert and emergency are available.

To disable the Debugbar, you can add the following line of code to the top of your file:

```
\Debugbar::disable();
```

Testing Your Laravel Application

Automated testing is a critical part of today's web development workflow, and should not be ignored even for the most trivial of projects. Fortunately, the Laravel developers agree with this mindset and include support for both PHPUnit and Dusk with every new Laravel project. PHPUnit is a very popular *unit testing framework* which allows you to create well-organized tests used to confirm all parts of your application are working as expected. Dusk is a Laravel sub-project which allows you to test your code *as it behaves in the web browser*, meaning you can ensure your code runs as desired by actually executing it within a browser such as Chrome. Testing is a major theme throughout this book, a subject we'll return to repeatedly to ensure the HackerPair code is correctly implemented. This section kicks things off by getting you acquainted with Laravel's default test infrastructure and PHPUnit fundamentals.

Introducing Unit Tests

Each new Laravel application even includes two example tests which you can use as a reference for beginning to write your own tests! One of the tests is located inside the `tests/Unit` directory. Tests placed in this directory are intended to ensure the smallest possible units of code are behaving as desired. For instance in later chapters we'll write unit tests to ensure model instance methods are returning correct output in conjunction with a variety of circumstances. The default test found in the `Unit` directory is named `ExampleTest.php` and it looks like this:

[38] http://www.php-fig.org/psr/psr-3/

```php
<?php

namespace Tests\Unit;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $this->assertTrue(true);
    }
}
```

Granted this isn't much to look at, but even so it gives you an idea of the basic test structure. Tests are managed within classes, with each test encapsulated in a class method. Each test begins with the prefix `test`, and will contain one or more *assertions*. Assertions intend to confirm your code's conformance to a requirement. For instance, you might assert that a method response value is `true`, `false`, `New York City`, or null.

Taking these characteristics into consideration, the `ExampleTest` class contains a single test named `testBasicTest`. It uses Laravel's testing API to interact with PHPUnit, confirming that the value `true` does in fact equal true (`assertTrue(true)`). You can run this test by executing the following command:

```
$ vendor/bin/phpunit tests/Unit/ExampleTest.php
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

.                                                   1 / 1 (100%)

Time: 242 ms, Memory: 10.00MB

OK (1 test, 1 assertion)
```

That period is indicative of a passing test. You're also told how many tests and assertions ran (`1 test, 1 assertion`). Let's add another passing test to the class:

```
public function testSomeValueIsFalse()
{
    $this->assertFalse(false);
}
```

Run the test suite again to see the results:

```
$ vendor/bin/phpunit tests/Unit/ExampleTest.php
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

..                                  2 / 2 (100%)

Time: 200 ms, Memory: 10.00MB

OK (2 tests, 2 assertions)
```

Unfortunately, your tests will rarely pass on the first time; that's just part of writing code. To see what a failing test looks like, change the `assertTrue` method in `testBasicTest` to look like this:

```
$this->assertTrue(false);
```

Run the test anew and you'll see an `F` in place of the period, and some feedback regarding why the test failed:

```
$ vendor/bin/phpunit tests/Unit/ExampleTest.php
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

F                                   1 / 1 (100%)

Time: 195 ms, Memory: 10.00MB

There was 1 failure:

1) Tests\Unit\ExampleTest::testBasicTest
Failed asserting that false is true.

/Users/wjgilmore/Code/valet/hackerpair/tests/Unit/ExampleTest.php:17

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

In particular, note the reference to the line number causing the failed assertion (17). This feedback will be very useful in terms of helping you to quickly track down why your tests are failing, and in later chapters I'll introduce even more efficient solutions when the reason isn't so obvious.

`assertTrue` and `assertFalse` are just two of many available assertion methods. For instance you would use `assertEquals` to confirm that a particular return value matches expectations:

```
public function testUserFullNameIsJasonGilmore()
{
    $fullName = "Jason Gilmore";
    $this->assertEquals("Jason Gilmore", $fullName);
}
```

You would use `assertCount` to confirm an array contains the expected number of values:

```
public function testUserHasFavoritedFiveEvents()
{
    $favorites = [45, 12, 676, 88, 15];
    $this->assertCount(5, $favorites);
}
```

Creating Your Own Test

You can easily create a test skeleton using the following command:

```
$ php artisan make:test TicketsTest --unit
```

This will create a new test inside the `tests/Unit` directory named `TicketsTest.php`. Open it up and you'll find the following contents:

```
<?php

namespace Tests\Unit;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class TicketsTest extends TestCase
{
    public function testExample()
    {
        $this->assertTrue(true);
    }
}
```

Once generated you can go about modifying (or deleting) the example test. We'll return to unit tests repeatedly throughout the book, introducing other assertion methods along the way. In the meantime, browse the PHPUnit documentation[39] for a preview of what's available.

Introducing Feature Tests

Another example test is found in the directory `tests/Feature` and also named `ExampleTest.php`. Feature tests are intended to confirm the behavior of multiple code units working together and often involve performing HTTP requests. In fact, the example test issues an HTTP request to retrieve the project home page, and determines whether a 200 response status code is returned (a 200 status code is indicative of a successful request):

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

To run the test, execute the `phpunit` command from within your project's root directory:

[39] https://phpunit.de/manual/current/en/appendixes.assertions.html

```
$ vendor/bin/phpunit tests/Feature/ExampleTest.php
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

.                                                   1 / 1 (100%)

Time: 130 ms, Memory: 12.00MB

OK (1 test, 1 assertion)
```

As you learned earlier in the chapter, the `web.php` routes file only contains a single route definition pointing to `/`. Let's confirm the `/contact` URI doesn't exist by attempting to retrieve it and confirming a 404 status code is returned:

```
public function testNonexistentEndpointReturns404()
{
    $response = $this->get('/contact');

    $response->assertStatus(404);
}
```

You can test much more than mere status codes; for instance you can use the `assertSeeText` method to determine whether the string `Laravel` is found in the response:

```
public function testHomepageContainsProjectName()
{
    $response = $this->get('/');

    $response->assertSeeText('Laravel');
}
```

Like unit tests, we'll return to feature tests throughout the book, and later we'll more heavily rely upon a relatively recent Laravel testing solution known as Dusk whenever the test involves interaction with web page elements. Despite Laravel Dusk's rise to prominence (Dusk is introduced in chapter 2), feature tests certainly continue to play an important role in ensuring application quality, and in later chapters we'll return to them whenever appropriate.

In the meantime, have a look at the Laravel documentation[40] for a list of assertions which can be used in conjunction with the `get`, `post`, `put`, `delete`, and `json` methods (I'll introduce these other test helper methods in later chapters).

[40] https://laravel.com/docs/5.5/http-tests

Bear in mind that while the `get` method is in fact retrieving your project's home page via an HTTP request, this does not involve a web browser. Therefore any JavaScript which may execute on a given endpoint is not going to execute, possibly resulting in unexpected results. You can test JavaScript within automated tests using Laravel Dusk, a Laravel feature we'll explore in chapter 2.

Additional Testing Resources

Automated testing is such an important part of building modern web applications that you owe it to yourself, your employer, and your clients to incorporate it into all of your projects. In doing so, you'll save untold amounts of time, pain, and money, not to mention allow you to focus on the entertaining aspects of web development rather than dreary manual testing and bug hunting. That said, I encourage you to keep the following resources in mind as you continue reading this book:

- Almost every chapter in this book concludes with a section explaining how to test the chapter's subject matter.
- Chapter 2 introduces Laravel Dusk, an amazing integration testing solution which allows you to confirm how your web application runs inside an actual browser. This capability is particularly crucial for applications which include JavaScript, since JavaScript is otherwise not capable of being tested using the other automated approaches discussed in this book.
- Laracasts (https://laracasts.com/) includes a free video series called "Testing Laravel" which covers Laravel testing fundamentals.
- Adam Wathan's Test-Driven Laravel[41] is undoubtedly the reference resource for learning how to test all facets of Laravel applications.

Conclusion

It's only the end of the first chapter and we've already covered a tremendous amount of ground! With your project generated and development environment configured, it's time to begin building the application. Onwards!

[41] http://www.testdrivenlaravel.com