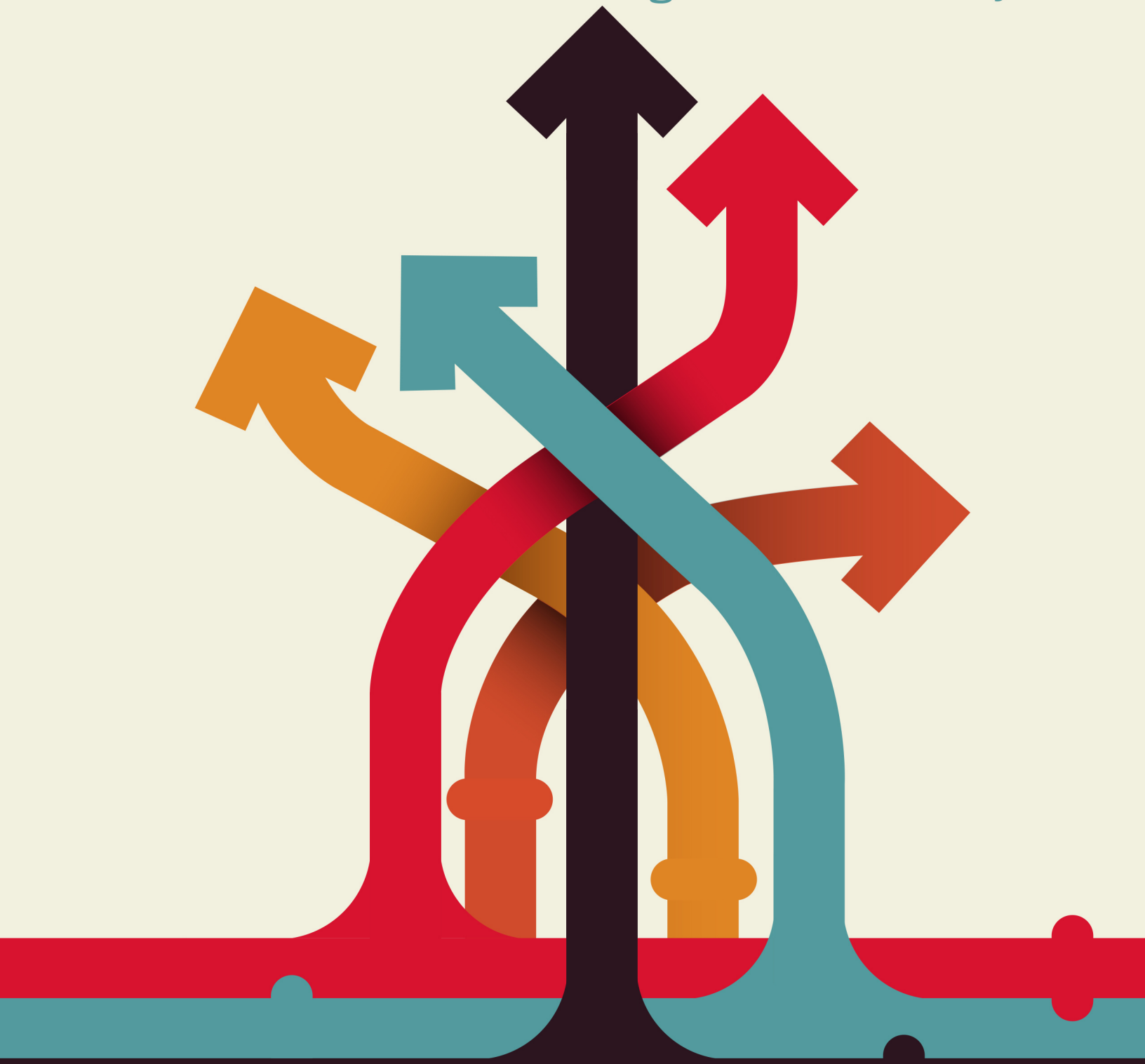
EASY LARAVEL 5

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 2015-04-01

Also By W. Jason Gilmore

Easy Active Record for Rails Developers

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

Contents

Introduction

I've spent the vast majority of the past 15 years immersed in the PHP language. During this time I've written seven PHP-related books, including a bestseller that has been in print for more than ten years. 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 instructed hundreds of developers in the United States and Europe. So you might be surprised to learn that a few years ago I became rather disenchanted with PHP. It felt like there were more exciting developments taking place within other programming communities, and wanting to be part of that buzz, I wandered off. In recent years, I spent the majority of my time working on a variety of projects including among others several ambitious Ruby on Rails applications and even a pretty amazing Linux-powered robotic device.

Of course, even during this time in the wilderness I kept tabs on the PHP community, watching with great interest as numerous talented developers worked tirelessly to inject that missing enthusiasm back into the language. Nils Adermann and Jordi Boggiano released the Composer[1] dependency manager. The Framework Interoperability Group[2] was formed. And in 2012 the incredibly talented Taylor Otwell[3] created the Laravel framework[4] which out of nowhere became the most popular PHP project on GitHub, quickly surpassing projects and frameworks that had been actively developed for years.

At some point I spent some time with Laravel and after a scant 30 minutes knew it was the real deal. Despite being the latest in a string of high profile PHP frameworks, Laravel is incredibly polished, offering a shallow learning curve, convenient PHPUnit integration, a great object-relational mapping solution called Eloquent, and a wide variety of other great features. The reasoning behind this pragmatic approach is laid bare in the project documentation[5], in which the Laravel development team describes their project goals:

> Laravel aims to make the development process a pleasing one for the developer without sacrificing application functionality. Happy developers make the best code. To this end, we've attempted to combine the very best of what we have seen in other web frameworks, including frameworks implemented in other languages, such as Ruby on Rails, ASP.NET MVC, and Sinatra.

Now that's something to get excited about! In the pages to follow I promise to add you to the ranks of fervent Laravel users by providing a wide-ranging and practical introduction to its many features.

[1] https://getcomposer.org/
[2] http://www.php-fig.org/
[3] http://taylorotwell.com/
[4] http://laravel.com/
[5] http://laravel.com/docs/master

What's New in Laravel 5?

Laravel 5 is an ambitious step forward for the popular framework, offering quite a few new features. In addition to providing newcomers with a comprehensive overview of Laravel's fundamental capabilities, I'll devote special coverage to several of these new features, including:

- New Project Structure: Laravel 5 projects boast a revamped project structure. In Chapter 1 I'll review every file and directory comprising the new structure so you know exactly where to find and place project files and other assets..
- Improved Environment Configuration: Laravel 5 adopts the PHP dotenv[6] package for environment configuration management. I think Laravel 4 users will really find the new approach to be quite convenient and refreshing. I'll introduce you to this new approach in Chapter 1.
- Route Annotations: The `routes.php` file remains in place for Laravel 5, however users now have the choice of alternatively using route annotations for route definitions. I'll show you how to use route annotations in Chapter 2.
- Elixir: Elixir[7] offers Laravel users a convenient way to automate various development tasks using Gulp[8], among them CSS and JavaScript compilation, JavaScript linting, image compression, and test execution. I'll introduce you to Elixir in Chapter 2.
- Flysystem: Laravel 5 integrates Flysystem[9], which allows you to easily integrate your application with remote file systems such as Dropbox, S3 and Rackspace.
- Form Requests: Laravel 5's new form requests feature greatly reduces the amount of code you'd otherwise have to include in your controller actions when validating and processing form data. In Chapter 5 I'll introduce you to this great new feature.
- Middleware: Laravel 5 introduces easy middleware integration. Middleware is useful when you want to interact with your application's request and response process in a way that doesn't pollute your application-specific logic. Chapter 7 is devoted entirely to this topic.
- Easy User Authentication: User account integration is the norm these days, however integrating user registration, login, logout, and password recovery into an application is often tedious and time-consuming. Laravel 5 all but removes this hassle by offering these features as a turnkey solution. I'll introduce you to these exciting capabilities in Chapter 6.

About this Book

This book is broken into eight chapters, each of which is briefly described below.

[6] https://github.com/vlucas/phpdotenv
[7] https://github.com/laravel/elixir
[8] http://gulpjs.com/
[9] https://github.com/thephpleague/flysystem

Chapter 1. Introducing Laravel

In this opening chapter you'll learn how to create and configure your Laravel project both using your existing PHP development environment and Laravel Homestead. I'll also show you how to properly configure your environment for effective Laravel debugging, and how to expand Laravel's capabilities by installing several third-party Laravel packages that promise to supercharge your development productivity. We'll conclude the chapter with an introduction to PHPUnit, showing you how to create and execute your first Laravel unit 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 using Laravel 5's new route annotations feature. 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 managing Gulp[10] tasks, and show you how to integrate the popular Bootstrap front-end framework and jQuery JavaScript library. We'll conclude the chapter with several examples demonstrating how to test your controllers and views using PHPUnit.

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, create and manage models, and interact with the database through your project models. You'll also learn how to deftly configure and traverse model relations, allowing you to greatly reduce the amount of SQL you'd otherwise have to write to integrate a normalized database into your application.

Chapter 4. Model Relations, Scopes, and Other Advanced Features

Building and navigating table relations is an 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. You'll also learn about a great feature known as scopes which encapsulate the logic used for more advanced queries, thereby hiding it from your controllers.

[10] http://gulpjs.com/

Chapter 5. Forms Integration

Your application will almost certainly contain at least a few web forms, which will likely interact with the models, meaning you'll require a solid grasp on Laravel's form generation and processing capabilities. While creating simple forms is fairly straightforward, things can complicated fast when implementing more ambitious solutions such as forms involving multiple models. In this chapter I'll go into extensive detail regarding how you can integrate forms into your Laravel applications, introducing Laravel 5's new form requests feature, covering both Laravel's native form generation solutions as well as several approaches offered by popular packages. You'll also learn how to test your forms in an automated fashion, meaning you'll never have to repetitively complete and submit forms again!

Chapter 6. Integrating Middleware

Laravel 5 introduces middleware integration. In this chapter I'll introduce you to the concept of middleware and the various middleware solutions bundled into Laravel 5. You'll also learn how to create your own middleware solution!

Chapter 7. 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.

Chapter 8. Deploying, Optimizing and Maintaining Your Application

"Deploy early and deploy often" is an oft-quoted mantra of successful software teams. To do so you'll need to integrate a painless and repeatable deployment process, and formally define and schedule various maintenance-related processes in order to ensure your application is running in top form. In this chapter I'll introduce the Laravel 5 Command Scheduler, which you can use to easily schedule rigorously repeating tasks. I'll also talk about optimization, demonstrating how to create a faster class router and how to cache your application routes. Finally, I'll demonstrate just how easy it can be to deploy your Laravel application to the popular hosting service Heroku, and introduce Laravel Forge.

Introducing the TODOParrot Project

Learning about a new technology is much more fun and practical when introduced in conjunction with real-world examples. Throughout this book I'll introduce Laravel concepts and syntax using code found in TODOParrot[11], a web-based task list application built atop Laravel.

[11] http://todoparrot.com

The TODOParrot code is available on GitHub at https://github.com/wjgilmore/todoparrot[12]. It's released under the MIT license, so feel free to download the project and use it as an additional learning reference or in any other manner adherent to the licensing terms.

About the Author

W. Jason Gilmore[13] is a software developer, consultant, and bestselling author. He has spent much of the past 15 years helping companies of all sizes build amazing solutions. Recent projects include a Rails-driven e-commerce analytics application for a globally recognized publisher, a Linux-powered autonomous environmental monitoring buoy, and a 10,000+ product online store.

Jason is the author of seven books, including the bestselling "Beginning PHP and MySQL, Fourth Edition", "Easy Active Record for Rails Developers", and "Easy PHP Websites with the Zend Framework, Second Edition".

Over the years Jason has 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. Jason is cofounder of the wildly popular CodeMash Conference[14], the largest multi-day developer event in the Midwest.

Away from the keyboard, you'll often find Jason playing with his kids, hunched over a chess board, and having fun with DIY electronics.

Jason loves talking to readers and invites you to e-mail him 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.

[12] https://github.com/wjgilmore/todoparrot
[13] http://www.wjgilmore.com
[14] http://www.codemash.org

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-driven development. If this is your first acquaintance with framework-driven development, you're in for quite a treat! Frameworks are so popular precisely because they dramatically decrease the amount of work you'd otherwise have to do by making many of the mundane decisions for you, a concept known as convention over configuration[15].

In this chapter you'll learn how to install Laravel and create your first Laravel project. We'll use this project as the basis for introducing new concepts throughout the remainder of the book, and to keep things interesting I'll base many of the examples around the TODOParrot application introduced in this book's introduction. I'll also introduce you to several powerful debugging and development tools that I consider crucial to Laravel development, showing you how to integrate them into your development environment. Finally, I'll show you how to configure Laravel's testing environment in order to create powerful automated tests capable of ensuring your Laravel application is operating precisely as expected.

After several months of at times incredibly tedious work I published this book on February 4, 2015, the very same day Laravel 5 officially released. So you may occasionally find conflicting material given the cutting-edge nature of this release. I do my very best to update the book immediately upon receiving errata (see http://easylaravelbook.com/changelog/), so please be patient, have fun reading, and if you think something is wrong, e-mail me at wj@wjgilmore.com.

Installing Laravel

Laravel is a PHP-based framework that you'll typically use in conjunction with a database such as MySQL or PostgreSQL. Therefore, before you can begin building a Laravel-driven web application you'll need to first install PHP 5.4 or newer and one of Laravel's supported databases (MySQL, PostgreSQL, SQLite, and Microsoft SQL Server). Therefore if you're already developing PHP-driven web sites and are running PHP 5.4 then installing Laravel will be a breeze, and you can jump ahead to the section "Creating the TODOParrot Application". If this is your first encounter with PHP then please take some time to install a PHP development environment now. How this is accomplished depends upon your operating system and is out of the scope of this book, however there are plenty

[15] http://en.wikipedia.org/wiki/Convention_over_configuration

of available online resources. If you have problems finding a tutorial suitable to your needs, please e-mail me and I'll help you find one.

Alternatively, if you'd rather go without installing a PHP development environment at this time, you have a fantastic alternative at your disposal called *Homestead*.

Laravel currently supports several databases, including MySQL, PostgreSQL, SQLite, and Microsoft SQL Server.

Introducing Homestead

PHP is only one of several technologies you'll need to have access to in order to begin building Laravel-driven web sites. Additionally you'll need to install a web server such as Apache[16] or nginx[17], a database server such as MySQL[18] or PostgreSQL[19], and often a variety of supplemental technologies such as Redis[20] and Grunt[21]. As you might imagine, it can be quite a challenge to install and configure all of these components, particularly when you'd prefer to be writing code instead of grappling with configuration issues.

In recent years the bar was dramatically lowered with the advent of the *virtual machine*. 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 an incredibly useful bit of technology, because you can use a virtual machine to for instance run Ubuntu Linux inside Windows 7, 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[22], a Vagrant[23]-based virtual machine which bundles everything you need to get started building Laravel-driven websites.

Homestead is currently based on Ubuntu 14.04, and includes everything you need to get started building Laravel applications, including PHP 5.6, Nginx, MySQL, PostgreSQL and a variety of other useful utilities. It runs flawlessly on OS X, Linux and Windows, and Vagrant configuration is pretty straightforward, meaning in most cases you'll have everything you need to begin working with Laravel in less than 30 minutes.

[16] http://httpd.apache.org/
[17] http://nginx.org/
[18] http://www.mysql.com/
[19] http://www.postgresql.org/
[20] http://redis.io/
[21] http://gruntjs.com/
[22] http://laravel.com/docs/homestead
[23] http://www.vagrantup.com/

Installing Homestead

Homestead requires Vagrant[24] and VirtualBox[25]. 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 window and execute the following command:

```
$ vagrant box add laravel/homestead
==> box: Loading metadata for box 'laravel/homestead'
    box: URL: https://vagrantcloud.com/laravel/homestead
==> box: Adding box 'laravel/homestead' (v0.2.2) for provider: virtualbox
    box: Downloading: https://vagrantcloud.com/laravel/boxes/homestead/
    versions/0.2.2/providers/virtualbox.box
==> box: Successfully added box 'laravel/homestead' (v0.2.2) for 'virtualbox'!
```

Throughout the book I'll use the $ to symbolize 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 a variety of boxes useful for different applications, so check out this list of popular boxes[26] for an idea of what else is available.

Once the box has been added, you'll next want to install the Homestead CLI tool. To do so, you'll use Composer:

```
$ composer global require "laravel/homestead=~2.0"
Changed current directory to /Users/wjgilmore/.composer
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing symfony/process (v2.6.3)
    Downloading: 100%

  - Installing laravel/homestead (v2.0.8)
    Downloading: 100%

Writing lock file
Generating autoload files
```

[24] http://www.vagrantup.com/

[25] https://www.virtualbox.org/wiki/Downloads

[26] https://vagrantcloud.com/discover/popular

After this command has completed, make sure your ~/.composer/vendor/bin directory is available within your system PATH. This is because the laravel/homestead package includes a command-line utility (named homestead) which you'll use to create your Homestead configuration directory:

```
$ homestead init
Creating Homestead.yaml file... ok
Homestead.yaml file created at: /Users/wjgilmore/.homestead/Homestead.yaml
```

Next you'll want to configure the project directory that you'll share with the virtual machine. Doing so requires you to identify the location of your public SSH key, because key-based encryption is used to securely share this directory. If you don't already have an SSH key and are running Windows, this SiteGround tutorial[27] offers a succinct set of steps. If you're running Linux or OS X, nixCraft[28] offers a solid tutorial.

You'll need to identify the location of your public SSH key in the .homestead directory's Homestead.yaml file. Open this file and locate the following line:

```
authorize: ~/.ssh/id_rsa.pub
```

If you're running Linux or OS X, then you probably don't have to make any changes to this line because SSH keys are conventionally stored in a directory named .ssh found in your home directory. If you're running Windows then you'll need to update this line to conform to Windows' path syntax:

```
authorize: c:/Users/wjgilmore/.ssh/id_rsa.pub
```

If you're running Linux or OS X and aren't using the conventional SSH key location, or are running Windows you'll also need to modify keys accordingly. For instance Windows users would have to update this section to look something like this:

```
keys:
    - c:/Users/wjgilmore/.ssh/id_rsa
```

Next you'll need to modify the Homestead.yaml file's folders list to identify the location of your Laravel project (which we'll create a bit later in this chapter). The two relevant Homestead.yaml settings are folders and sites, which by default look like this:

[27] http://kb.siteground.com/how_to_generate_an_ssh_key_on_windows_using_putty/
[28] http://www.cyberciti.biz/faq/how-to-set-up-ssh-keys-on-linux-unix/

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

sites:
    - map: homestead.app
      to: /home/vagrant/Code/Laravel/public
```

It's this particular step that tends to confuse most Homestead beginners, so pay close attention to the following description. The `folders` object's `map` attribute identifies the location in which your Laravel project will be located. The default value is `~/Code`, meaning Homestead expects your project to reside in a directory named `Code` found in your home directory. You're free to change this to any location you please, keeping in mind for the purposes of this introduction the directory *must* be your Laravel project's root directory (why this is important will become apparent in a moment). The `folders` object's `to` attribute identifies the location *on the virtual machine* that will mirror the contents of the directory defined by the `map` key, thereby making the contents of your local directory available to the virtual machine.

The `sites` object's `map` attribute defines the domain name used to access the Laravel application via the browser. Leave this untouched for now. Finally, the `sites` object's `to` attribute defines the Laravel project's root *web directory*, which is `/public` by default. This isn't just some contrived setting; not only is `/public` the directory you would need to configure when setting up a web server to serve a Laravel application, but `/home/vagrant/Code/Laravel/public` is also the directory that Homestead's nginx web server has been configured to use! This means that the path defined by the `folders` `map` attribute *must* contain a directory named `Laravel`, and inside that a directory named `public`. If you do not do this you'll receive the dreaded `404` error when attempting to access the application via your browser.

If this explanation is clear as mud, let's clarify with an example. Begin by setting the `folders` object's `map` attribute to any path you please, likely somewhere within the directory where you tend to manage your various software projects. For instance, mine is currently set like this:

```
folders:
    - map: /Users/wjgilmore/Software/dev.todoparrot.com
    - to: /home/vagrant/Code
```

Next, create a directory named `Laravel` inside the directory identified by the `map` attribute, and inside it create a directory named `public`. Create a file named `index.php` inside the `public` directory, adding the following contents to it:

```
<?php echo "Hello from Homestead!"; ?>
```

Save these changes, and then run the following command:

```
$ homestead up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'laravel/homestead'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'laravel/homestead' is up to date...
...
==> default: Forwarding ports...
default: 80 => 8000 (adapter 1)
default: 443 => 44300 (adapter 1)
default: 3306 => 33060 (adapter 1)
default: 5432 => 54320 (adapter 1)
default: 22 => 2222 (adapter 1)
$
```

Your Homestead virtual machine is up and running! Open a browser and navigate to the URL `http://localhost:8000` and you should see `Hello from Homestead!`. Note the use of the `8000` port in the URL. This is because the Homestead virtual machine forwards several key ports to non-standard port numbers, allowing you to continue using the standard ports locally. I've included the list of forwarded ports in the debug output that followed the `vagrant up` command. As you can see, port 80 (for HTTP) forwards to 8000, port 3306 (for MySQL) forwards to 33060, port 5432 (for PostgreSQL) forwards to 54321, and port 22 (for SSH) forwards to 2222.

Next you'll want to update your development machine's `hosts` file so you can easily access the server via a hostname rather than the IP address found in the `Homestead.yaml` 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  homestead.app
```

After saving the changes, open a browser and navigate to `http://homestead.app`. If the virtual machine did not start, or if you do not see `Hello from Homestead!` when accessing `http://homestead.app`, then double-check your `Homestead.yaml` file, ensuring all of the paths are properly set.

Remember, we just created the `Laravel/public` directory to confirm Homestead is properly configured and able to serve files found in our local development directory. You should subsequently delete this directory as it will be created automatically when we generate the book theme project in the section, "Creating the TODOParrot Application".

Incidentally, if you'd like to shut down the virtual machine you can do so using the following command:

```
$ homestead halt
==> default: Attempting graceful shutdown of VM...
$
```

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

```
$ homestead destroy
```

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. You can SSH into the virtual machine using the `ssh` command if you're running Linux or OS X, or using a variety of SSH clients if you're running Windows (My favorite Windows SSH client is PuTTY[29].:

```
$ ssh vagrant@127.0.0.1 -p 2222
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-30-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

  System information as of Thu Jan  8 00:57:20 UTC 2015

  System load:  0.96              Processes:           104
  Usage of /:   5.0% of 39.34GB   Users logged in:     0
  Memory usage: 28%               IP address for eth0: 10.0.2.15
  Swap usage:   0%                IP address for eth1: 192.168.33.10

  Graph this data and manage this system at:
    https://landscape.canonical.com/

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

Last login: Fri Dec 19 15:01:15 2014 from 10.0.2.2
```

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:

[29] 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!

Creating the TODOParrot Application

With Laravel (and optionally Homestead) installed and configured, it's time to get our hands dirty! We're going to start by creating the TODOParrot application, as it will serve as the basis for much of the instructional material presented throughout this book. There are a couple of different ways in which you can do this, but one of the easiest involves invoking Composer's[30] `create-project` command (Composer is PHP's de facto package management solution):

```
$ composer create-project laravel/laravel dev.todoparrot.com --prefer-dist
Installing laravel/laravel (v5.0.1)
-     Installing laravel/laravel (v5.0.1)
...
Application key [9UCBk7IDjvAGrkLOUBXw43yYKlymlqE3Y]
set successfully.
```

If you're using Homestead remember that the Laravel application must reside *inside* the directory identified by the `homestead.yaml folders` object's `map` attribute. Otherwise, if you're working with a local PHP environment, you can execute it wherever you'd like the project to be managed:

Obviously you'll need to install Composer to use Laravel in this fashion, however you'll need it anyway to perform other tasks such as package installation. See the Composer[31] website for more information regarding installation.

Additionally, Laravel will create a random application key used when encrypting sensitive data. This key is stored in `config/app.php` (more about this file in a moment). You'll typically not have to do anything with this key however I wanted to at least briefly introduce now in case you're wondering why the installer thought it important to reference the key's generation.

This command tells Composer to create a new project using the `laravel/laravel` package, placing the project contents in the directory `dev.todoparrot.com`. These 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:

[30] https://getcomposer.org

[31] https://getcomposer.org

- `.env`: Laravel 5 uses the PHP dotenv[32] to conveniently manage environment-specific settings. You'll use `.env` file as the basis for configuring these settings. A file named `.env.example` is also included in the project root directory. This file should be used as the setting template, which fellow developers will subsequently copy over to `.env` and change to suit their own needs. I'll talk about this file and Laravel 5's new approach to managing environment settings in the later section, "Configuring Your Laravel Application".
- `.gitattributes`: This file is used by Git[33] to ensure consistent settings across machines, which is particularly useful when multiple developers using a variety of operating systems are working on the same project. The lone setting found in your project's `.gitattributes` file (`text=auto`) ensures file line endings are normalized to `LF` whenever the files are checked into the repository. Plenty of other attributes are however available; Scott Chacon's book, "Pro Git"[34] includes a section ("Customizing Git - Git Attributes"[35]) 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 the usual suspects in here, including the annoying OS X `.DS_Store` file, Windows' equally annoying `Thumbs.db` file, and the `vendor` directory, which includes the Laravel source code and various other third-party packages.
- `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 application development progresses.
- `artisan`: `artisan` is a command-line interface we'll use to rapidly develop new parts of your applications such as controllers, manage your database's evolution through a great feature known as *migrations*, and clear the application cache. You'll also regularly use `artisan` to interactively debug your application, and even easily view your application within the browser using the native PHP development server. We'll return to `artisan` repeatedly throughout the book as 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 the `bootstrap` directory, although I encourage you to have a look as each is heavily commented.
- `composer.json`: Composer[36] is the name of PHP's popular package manager, used by thousands of developers around the globe to quickly integrate popular third-party solutions such as Swift Mailer[37] and Doctrine[38] into a PHP application. Laravel supports 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 you'll quickly come to wonder how you

[32] https://github.com/vlucas/phpdotenv
[33] http://git-scm.com/
[34] http://git-scm.com/book
[35] http://git-scm.com/book/en/Customizing-Git-Git-Attributes
[36] https://getcomposer.org
[37] http://swiftmailer.org/
[38] http://www.doctrine-project.org/

ever lived without it. In fact in this introductory chapter alone we'll use it several times to install several useful packages.

- `composer.lock`: This file contains information about the state of the 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 more than a dozen files used to configure various aspects of your Laravel application, such as the database credentials, and 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).
- `gulpfile.js`: Laravel 5 introduces a new feature called *Laravel Elixir*. `Gulpfile.js` is used by Elixir to define various Gulp.js[39] tasks used by Elixir to automate various build-related processes associated with your project's CSS, JavaScript, tests, and other assets. I'll introduce Elixir in Chapter 2.
- `package.json`: This file is used by the aforementioned Elixir to install Elixir and its various dependencies. I'll talk about this file in Chapter 2.
- `phpspec.yml`: This file is used to configure the behavior driven development tool phpspec[40]. In this book I'll discuss Laravel testing solely in the context of PHPUnit but hope to include coverage of phpspec in a forthcoming update.
- `phpunit.xml`: Even relatively trivial web applications should be accompanied by an automated test suite. Laravel leaves little room for excuse to shirk this best practice by configuring your application to use the popular PHPUnit[41] 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 this topic repeatedly throughout the book, so stay tuned.
- `public`: The `public` directory serves as your application's 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.
- `readme.md`: The `readme.md` file contains some boilerplate information about Laravel of the sort that you'll typically find in an open source project. Feel free to replace this text with information about your specific project. See the TODOParrot[42] README file for an example.
- `resources`: The `resources` directory contains your project's views and localized language files. You'll also store your project's raw assets (CoffeeScript, SCSS, etc.).
- `storage`: The `storage` directory contains your project's cache, session, and log data.
- `tests`: The `tests` directory contains your project's PHPUnit tests. Testing is a recurring theme throughout this book, complete with numerous examples.

[39] http://gulpjs.com/
[40] http://www.phpspec.net/
[41] http://phpunit.de/
[42] http://github.com/wjgilmore/todoparrot

- `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 `artisan` utility and Composer interface.

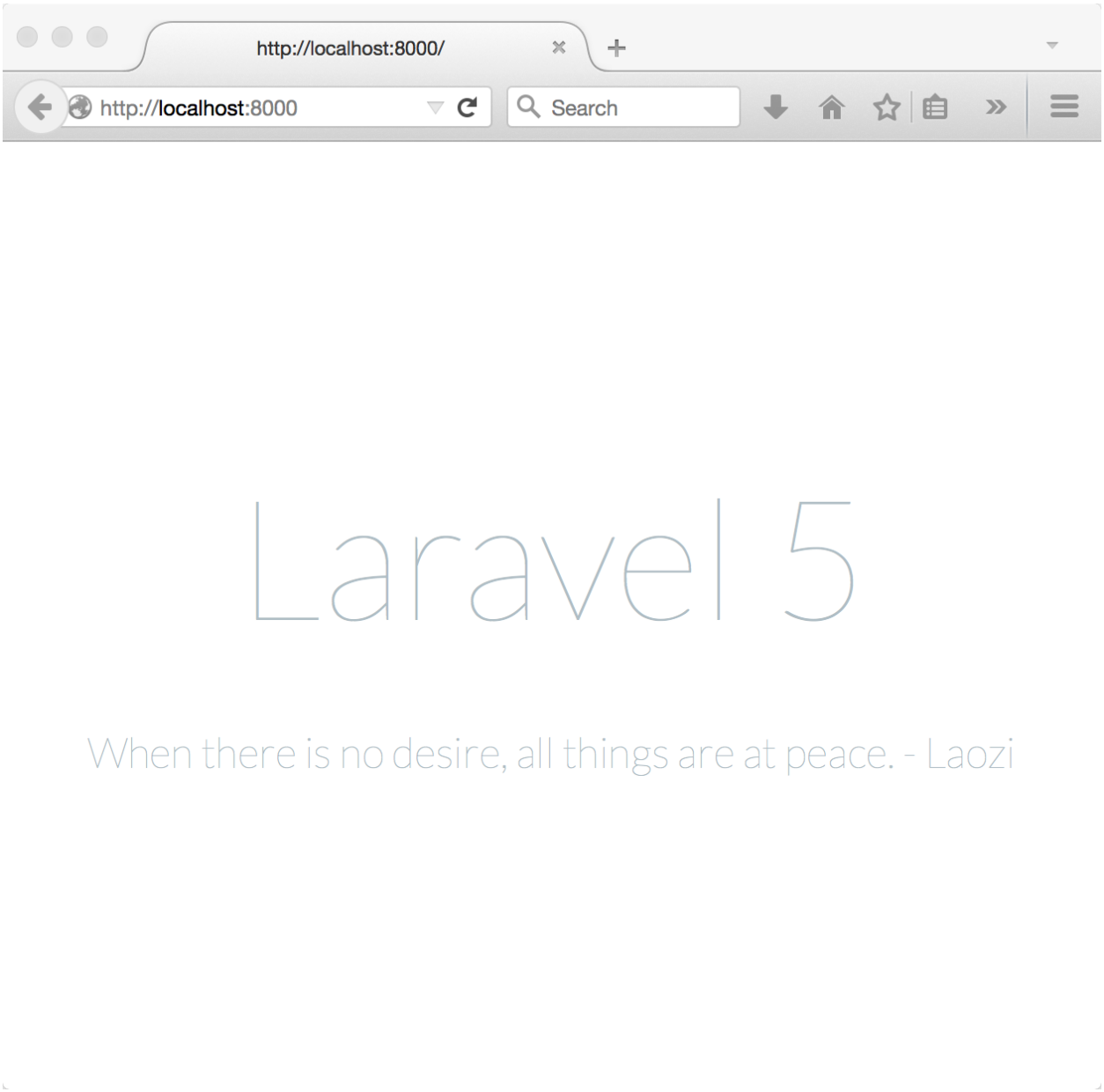
Now that you have a rudimentary understanding of the various directories and files comprising a Laravel skeleton application let's see what happens when we load the default application into a browser. If you're using Homestead then navigate to `http://homestead.app`, otherwise if you plan on using PHP's built-in development server, start the server by executing the following command:

```
$ php artisan serve
Laravel development server started on http://localhost:8000
```

Alternatively you could use the built-in PHP server:

```
$ php -S localhost:8000 -t public /
PHP 5.5.15 Development Server started at Wed Jan  7 20:30:49 2015
Listening on http://localhost:8000
Document root is /Users/wjgilmore/Code/Laravel/public
Press Ctrl-C to quit.
```

Once the server is running, open your browser and navigate to the URL `http://localhost:8000`. Load this URL to your browser and you'll see the page presented in the below figure.



The Laravel splash page

As you can see, the Laravel logo is presented in the default page. So where is this page and logo located? It's found in a *view*, and in the next chapter I'll introduce Laravel views in great detail.

Setting the Application Namespace

Laravel 5 uses the PSR-4 autoloading standard[43], meaning your project controllers, models, and other key resources are namespaced. The default namespace is set to `app`, which is pretty generic. You'll

[43] http://www.php-fig.org/psr/psr-4/

likely want to update your project's namespace to something reasonably unique, such as `todoparrot`. You can do so using the `artisan` CLI's `app:name` command:

```
$ php artisan app:name todoparrot
Application namespace set!
```

This command will not only update the default namespace setting (by modifying `composer.json`'s `autoload/psr-4` setting), but will additionally updating any namespace declarations found in your controllers, models, and other relevant files.

Configuring Your Laravel Application

Most web frameworks, Laravel included, offer 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 moment), the application URL, timezone, locale, and autoloaded service providers.
- `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.
- `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.
- `compile.php`: Laravel can improve application performance by generating a series of files that allow for faster package autoloading. The `compile.php` configuration file allows you to define additional class files that should be included in the optimization step.
- `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.
- `filesystems.php`: The `filesystems.php` configuration file defines the file system your project will use to manage assets such as file uploads. Currently the local disk, Amazon S3, and Rackspace are supported.

- `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 (SMTP, Sendmail, PHP's `mail()` function, Mailgun, and the Mandrill API are supported). You can also direct mails 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[44] or Amazon Simple Queue Service[45]. 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 Mandrill for e-mail delivery you'll use the `services.php` configuration file to define any third-party service-specific settings.
- `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[46], 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.

Programming Terminology Alert

The terms *service provider* and *facade* regularly make and appearance within Laravel documentation, tutorials and discussions. This is because Laravel was conceived with interoperability in mind, providing the utmost flexibility in terms of being able to swap out for instance one logging or authentication implementation for another, extend Laravel with a new approach to database integration, or enhance Laravel's form generation capabilities with new features. Each of these distinct features are incorporated into Laravel via a *service provider*, which is responsible for configuring the feature for use within a Laravel application. A list of service providers integrated into your newly created project can be found in the `config/app.php` file's `providers` array. Laravel users will then typically use *facades* to access the functionality made available by the classes integrated via the service providers. A facade just facilitates interaction with these classes, and nothing more. For now that's pretty much all you need to know about these two topics, but I thought at least a cursory definition of each was in order since I'll unavoidably use both terms in this chapter and beyond.

[44] http://kr.github.io/beanstalkd/

[45] http://aws.amazon.com/sqs/

[46] http://php.net/manual/en/book.apc.php

Configuring Your Environment

Laravel presumes your application is running in a production environment, meaning the options found in the various `config` files are optimized for production use. Logically you'll want to override at least a few of these options when the application is running in your development (which Laravel refers to as `local`) environment. Laravel 5 completely overhauls the approach used to detect the environment and override environment-specific settings. It now relies upon the popular PHP dotenv[47] package. You'll set the environment simply by updating the `.env` file found in your project's root directory to reflect the desired environment settings. The `.env` file looks like this:

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=SomeRandomString

DB_HOST=localhost
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

CACHE_DRIVER=file
SESSION_DRIVER=file
```

Laravel will look to this file to determine which environment is being used (as defined by the `APP_ENV` variable). These variables can then be used within the configuration files via the `env` function, as demonstrated within the `config/database.php` file, which retrieves the `DB_DATABASE`, `DB_USERNAME`, and `DB_PASSWORD` variables:

```
'mysql' => [
        'driver'    => 'mysql',
        'host'      => env('DB_HOST', 'localhost'),
        'database'  => env('DB_DATABASE', 'forge'),
        'username'  => env('DB_USERNAME', 'forge'),
        'password'  => env('DB_PASSWORD', ''),
        'charset'   => 'utf8',
        'collation' => 'utf8_unicode_ci',
        'prefix'    => '',
        'strict'    => false,
],
```

We'll add to the configuration file as new concepts and features are introduced throughout the remainder of this book.

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

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 my favorite such 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[48], FirePHP[49], and the many tools integrated into PHP IDEs such as Zend Studio[50] and PHPStorm[51].

The dd Function

Ensuring the `debug` option is enabled is the easiest way to proactively view information about any application errors however it isn't a panacea for all debugging tasks. For instance, sometimes 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()`[52] helper function, which will dump a variable's contents to the browser and halt further script execution. To demonstrate the `dd()` function open the file `app/Http/Controllers/WelcomeController.php` and you'll find single class method that looks like this:

```
1  public function index()
2  {
3      return view('welcome');
4  }
```

Controllers and these class methods (actions) will be formally introduced in Chapter 4; for the moment just keep in mind that the `Welcome` controller's `index` action executes when a user requests your web application's home page. Modify this method to look like this:

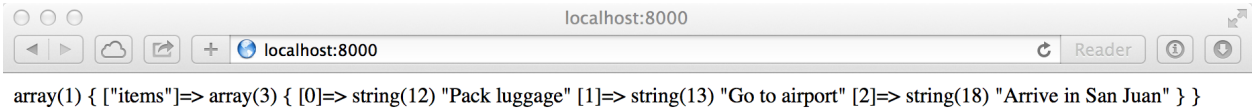
[48] http://xdebug.org/
[49] http://www.firephp.org/
[50] http://www.zend.com/en/products/studio
[51] http://www.jetbrains.com/phpstorm/
[52] http://laravel.com/docs/helpers#miscellaneous

```
public function index()
{

    $items = array('items' => ['Pack luggage', 'Go to airport',
    'Arrive in San Juan']);
    dd($items);

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

Reload the home page in your browser and you should see the `$items` array contents dumped to the browser window as depicted in the below screenshot.



dd() function output

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 necessarily 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[53], 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 Firebug console[54] via FirePHP[55], to the Chrome console[56] using Chrome Logger[57], or even trigger alerts via e-mail, HipChat[58] or Slack[59]. Further, if you're using the Laravel 4 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. Open the `app/Http/Controllers/WelcomeController.php` file and modify the `index` method to look like this:

```
1  public function index()
2  {
3
4      $items = ['Pack luggage', 'Go to airport', 'Arrive in San Juan'];
5      \Log::debug($items);
6
7  }
```

Save the changes, reload `http://localhost:8000`, and a log message similar to the following will be appended to `storage/logs/laravel.log`:

[2015-01-08 01:51:56] local.DEBUG: array (0 ⇒ 'Pack luggage', 1 ⇒ 'Go to airport', 2 ⇒ 'Arrive in San Juan',)

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:

[53] https://github.com/Seldaek/monolog
[54] https://getfirebug.com/
[55] http://www.firephp.org/
[56] https://developer.chrome.com/devtools/docs/console
[57] http://craig.is/writing/chrome-logger
[58] http://hipchat.com/
[59] https://www.slack.com/

```
\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!');
```

Integrating the Logger and FirePHP

When monitoring the log file it's common practice to use the `tail -f` command (available on Linux and OS X) to view any log file changes in real time. You can however avoid the additional step of maintaining an additional terminal window for such purposes by instead sending the log messages to the Firebug[60] console, allowing you to see the log messages alongside your application's browser output. You'll do this by integrating FirePHP[61].

You'll first need to install the Firebug and FirePHP[62] extensions, both of which are available via Mozilla's official add-ons site. After restarting your browser, you can begin sending log messages directly to the Firebug console like so:

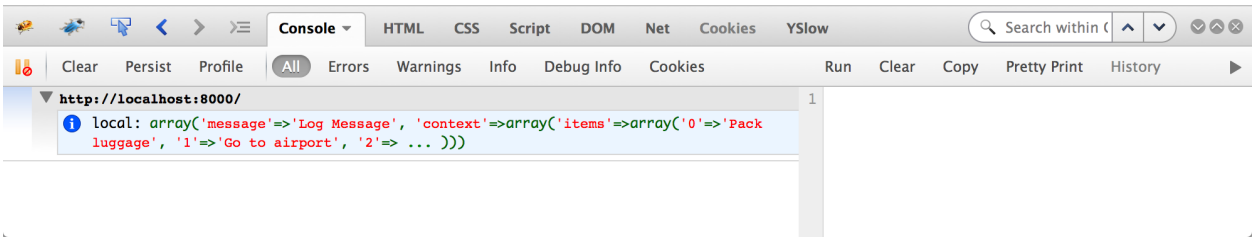
```
$monolog = \Log::getMonolog();

$items = ['Pack luggage', 'Go to airport', 'Arrive in San Juan'];

$monolog->pushHandler(new \Monolog\Handler\FirePHPHandler());

$monolog->addInfo('Log Message', array('items' => $items));
```

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



Logging to Firebug via FirePHP

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

[60] https://getfirebug.com/
[61] http://www.firephp.org/
[62] https://addons.mozilla.org/en-US/firefox/addon/firephp/

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 --env=local
Psy Shell v0.3.3 (PHP 5.5.18 â€” cli) by Justin Hileman
>>>
```

Notice tinker uses PsySH[63], 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[64] to learn more about what this great utility can do. In the meantime, let's get used to the interface:

```
>>> $items = ['Pack luggage', 'Go to airport', 'Arrive in San Juan'];
=> [
       "Pack luggage",
       "Go to airport",
       "Arrive in San Juan"
   ]
```

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

```
>>> var_dump($items);
array(3) {
  [0]=>
  string(12) "Pack luggage"
  [1]=>
  string(13) "Go to airport"
  [2]=>
  string(18) "Arrive in San Juan"
}
=> null
>>> sort($items);
=> true
>>> $items;
=> [
       "Arrive in San Juan",
       "Go to airport",
       "Pack luggage"
   ]
>>>
```

[63] http://psysh.org/
[64] http://psysh.org/

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

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

PsySH 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 it 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[65] 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

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

The Debugbar is visually similar to Firebug[66], consisting of multiple 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: This tab presents a summary of the time required to load the page.
- Exceptions: This tab displays any exceptions thrown while processing the current request.
- Views: This tab provides information about the various views used to render the page, including the layout.
- Route: This tab presents information about the requested route, including the corresponding controller and action.
- Queries: This tab 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.
- Request: This tab 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
Using version ~2.0 for barryvdh/laravel-debugbar
./composer.json has been updated
...
Writing lock file
Generating autoload files
$
```

Next, add the following lines to the `providers` and `aliases` arrays to your `config/app.php` file, respectively:

```
'providers' => [
    ...
    'Barryvdh\Debugbar\ServiceProvider'
],

...

'aliases' => [
    ...
    'Debugbar' => 'Barryvdh\Debugbar\Facade'
]
```

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

[66] http://getfirebug.com

```
$ php artisan vendor:publish
```

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!

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. Add the following line to the `Welcome` controller's `index` action (`app/Http/Controllers/WelcomeController.php`):

```
\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[67], meaning methods for debug, info, notice, warning, error, critical, alert and emergency are available.

Testing Your Laravel Application with PHPUnit

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 automatically include reference the PHPUnit package within every new Laravel project's `composer.json` file:

```
"require-dev": {
  "phpunit/phpunit": "~4.0"
},
```

Laravel 5 includes support for a second testing framework called phpspec[68]. This book doesn't currently include phpspec coverage (pun not intended, I swear!), however stay tuned as a forthcoming release will include an introduction to the topic in the context of Laravel.

Running Your First Test

PHPUnit is a command-line tool that when installed via your project's `composer.json` file is found in `vendor/bin`. Therefore to run PHPUnit you'll execute it like this:

[67] http://www.php-fig.org/psr/psr-3/
[68] http://www.phpspec.net/

```
$ vendor/bin/phpunit --version
PHPUnit 4.6-dev by Sebastian Bergmann and contributors.
```

If you find typing `vendor/bin/` to be annoying, consider making PHPUnit globally available, done using Composer's `global` modifier. Rob Allen has written up a concise tutorial[69] showing you how this is accomplished.

Inside the `tests` directory you'll find a file named `ExampleTest.php` that includes a simple unit test. This test accesses the project home page, and determines whether a 200 status code is returned:

```
<?php

class ExampleTest extends TestCase {

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->call('GET', '/');

        $this->assertEquals(200, $response->getStatusCode());
    }

}
```

In PHPUnit lingo, the test is *asserting* that `$response->getResponse()->isOk()` will return `200`. This approach of confirming assertions is the typical approach when writing PHPUnit tests; each test will define a particular application state and then you'll make an assertion regarding a particular condition. To run the test, just execute the `phpunit` command:

[69] http://akrabat.com/php/global-installation-of-php-tools-with-composer/

```
$ vendor/bin/phpunit
PHPUnit 4.6-dev by Sebastian Bergmann.

Configuration read from /Users/wjgilmore/Software/dev.todoparrot.com/phpunit.xml

.

Time: 94 ms, Memory: 10.25Mb

OK (1 test, 1 assertion)
```

See that single period residing on the line by itself? That represents a passed test, in this case the test defined by the `testBasicExample` method. If the test failed, you would instead see an `F` for error. To see what a failed test looks like, open up `app/Http/routes.php` and comment out the following line:

```
$router->get('/', 'WelcomeController@index');
```

I'll introduce the `app/Http/routes.php` file in the next chapter, so don't worry if you don't understand what a route definition is; just understand that by commenting out this line you will prevent Laravel from being able to serve the home page. Save the changes and execute `phpunit` anew:

```
$ vendor/bin/phpunit
PHPUnit 4.6-dev by Sebastian Bergmann and contributors.

Configuration read from /Users/wjgilmore/Code/Laravel/phpunit.xml

F

Time: 234 ms, Memory: 10.25Mb

There was 1 failure:

1) ExampleTest::testBasicExample
Failed asserting that 404 matches expected 200.

/Users/wjgilmore/Code/Laravel/tests/ExampleTest.php:14

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

This time the `F` is displayed, because the assertion defined in `testBasicExample` failed. Additionally, information pertaining to why the test failed is displayed. In the chapters to come we will explore other facets of PHPUnit and write plenty of additional tests.

Consider spending some time exploring the PHPUnit[70] and Laravel[71] documentation to learn more about the syntax available to you. In any case, be sure to uncomment that route definition before moving on!

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 TODOParrot application. Onwards!

[70] https://phpunit.de/documentation.html
[71] http://laravel.com/docs/master/testing

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

The typical dynamic web page consists of various components which are assembled at runtime to produce what the user sees in the browser. These components include the *view*, which consists of the design elements and content specific to the page, the *layout*, which consists of the page header, footer, and other design elements that tend to more or less globally appear throughout the site, and other assets such as the images, JavaScript and CSS. Web frameworks such as Laravel create and return these pages in response to a route request, processing these requests through a controller and action. This chapter offers a wide-ranging introduction to all of these topics, complete with introductions to new Laravel 5 features including Elixir and the route annotations add-on. We'll conclude the chapter with several examples demonstrating how to test your views and controllers using PHPUnit.

Creating Your First View

In the previous chapter we created the TODOParrot project and viewed the default landing page within the browser. The page was pretty sparse, consisting of the text "Laravel 5" and a random quotation. If you view the page's source from within the browser, you'll see a few CSS styles are defined, a Google font reference, and some simple HTML. You'll find this view in the file `welcome.blade.php`, found in the directory `resources/views`. Open this file in your PHP editor, and update it to look like this:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Welcome to TODOParrot</title>
    <style>
      @import url(//fonts.googleapis.com/css?family=Lato:700);

      body {
        margin:0;
        font-family:'Lato', sans-serif;
        text-align:center;
```

```
        color: #999;
      }

    </style>
  </head>
  <body>
  <h1>Welcome to TODOParrot</h1>
  </body>
</html>
```

Reload the application's home page within your browser (don't forget to restart the PHP development server if you shut it down after completing the last chapter), and you should see "Welcome to TODOParrot" centered on the page. Congratulations! You've just created your first Laravel view.

So why is this particular view returned when you navigate to the application home page? The `welcome.blade.php` view is served by the `Welcome` controller's `index` action, which is configured to respond to a Laravel application's home page by default. Let's have a look at the `WelcomeController.php` file (comments removed):

```
<?php namespace todoparrot\Http\Controllers;

class WelcomeController extends Controller {

  public function __construct()
  {
    $this->middleware('guest');
  }

  public function index()
  {
    return view('welcome');
  }

}
```

As you can see, a Laravel controller is just a PHP class that inherits from Laravel's `Controller` class. For the moment don't worry about the middleware reference in the class constructor, as this is something we'll discuss in detail in Chapter 6. The class consists of one or more public methods known as *actions*. The `Welcome` controller contains just a single action named `index`, which looks like this:

```
public function index()
{
  return view('welcome');
}
```

The `index` action is currently responsible for a single task: serving the `welcome` view, accomplished using the `view` method:

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

Because it's understood that views use the `.php` extension (more about the meaning of `blade` in a moment), Laravel saves you the hassle of referencing the extension. Of course, you're free to name the view whatever you please; try renaming `welcome.blade.php` as `hola.blade.php`, and then update the `view` method to look like this:

```
return view('hola');
```

Additionally, you're free to manage views in separate directories for organizational purposes, and can use a convenient dot-notation syntax for representing the directory hierarchy. For instance you could organize views according to controller by creating a series of aptly-named directories in `resources/views`. As an example, create a directory named `Homepage` in `resources/views`, and move the `welcome.blade.php` view into the newly created directory. Then update the `view` method to look like this:

```
return view('Homepage.welcome');
```

So you now know a bit about views, controllers and actions, but I still haven't explained how Laravel knew to execute *this* particular action and serve the `welcome.blade.php` view when the home page was requested. Without further ado let's answer this question.

Managing Your Application Routes

Every controller action is associated with a URL endpoint so Laravel knows how to respond to a particular request. For instance the `Welcome` controller's `index` action is associated by default with the root, or home URL. You're free to change this route to anything you please, and additionally create new routes as new controllers and actions are created. Beginning with version 5 Laravel offers two distinct approaches to route management; you're free to choose either approach, just be sure to stick with one or the other so it's always apparent how to locate and manage your application's routing definitions.

Introducing the routes.php File

Route definitions are by default managed in the `app/Http/routes.php` file. The default file looks like this (comments removed):

```
<?php

Route::get('/', 'WelcomeController@index');

Route::get('home', 'HomeController@index');

Route::controllers([
  'auth' => 'Auth\AuthController',
  'password' => 'Auth\PasswordController',
]);
```

Three routes are defined in this example. The first tells Laravel to respond to `GET` requests made to the project's root URL (`http://localhost:8000` or `http://homestead.app` in the development environment) by executing the `Welcome` controller's `index` action. The second definition referring to the `HomeController` defines the location where *authenticated* users will be routed to by default. We'll talk more about the purpose of this route in Chapter 6.

The third definition is a tad more complicated; it uses the `Route::controllers` method to define a series of URIs and associated controllers. Reflection is used parse and register all of the controller's routable methods. I'd be jumping the gun at this point to try and discuss exactly how this is accomplished in regards to the `AuthController.php` and `PasswordController.php` controllers, because both are examples of somewhat more complicated controllers used to manage various aspects of user authentication and accounts (both of these controllers are introduced in detail in Chapter 6). However, the examples found below should help you to understand how `Route::controllers` and other routing-related features behave.

Simultaneously Defining Multiple Routes

As mentioned above, you can define multiple URIs and their associated controllers by passing them into the `Route::controllers` method. Therefore if you created two controllers named `ListsController.php` and `TasksController.php` you might register all of the actions found in these controllers by adding the following definition to your `routes.php` file:

```
Route::controllers([
  'lists' => 'ListsController',
  'tasks' => 'TasksController',
]);
```

The `ListsController.php` file might look like this:

```
<?php namespace todoparrot\Http\Controllers;

class ListsController extends Controller {

  public function getIndex()
  {
    return view('lists.index');
  }

  public function getCreate()
  {
    return view('lists.create');
  }

  public function postStore()
  {
    return view('lists.store')
  }

}
```

Notice how I've prefixed the `ListsController.php` actions with `get` and `post`. These prefixes inform Laravel as to the HTTP method which should be used in conjunction with the URI. Therefore after creating a corresponding view for the `getCreate` action (storing it in `resources/views/lists/create.blade.php`) you should be able to navigate to `http://homestead.app/tasks/create` and see the view contents. If you were to create a form and *post* the form contents to `http://homestead.app/tasks/store` you should see the contents of the `resources/view/lists/store.blade.php` view.

Incidentally, if you prefer to define each set of controller routes separately you can do so using the `Route::controller` method:

```
Route::controller('tasks', 'TasksController');
```

If you're familiar with RESTful routing then the above approach certainly seems rather tedious. Not to worry! Laravel supports RESTful routing, and in the next chapter I'll show you how to take advantage of it to eliminate the need to prefix your action names when working within a pure REST-based environment. In the meantime let's have a look at several other useful routing examples.

Defining Custom Routes

Laravel supports RESTful controllers (introduced in the next chapter), meaning for many standard applications you won't necessarily have to be bothered with explicitly defining custom routes and

managing route parameters. However, should you need to define a non-RESTful route Laravel offers an easy way to define and parse parameters passed along via the URL. Suppose you wanted to build a custom blog detailing the latest TODOParrot features, and wanted to create pages highlighting posts on a per-category basis. For instance, the `PHP` category page might use a URL such as `http://todoparrot.com/blog/category/php`. You might first create a `Blog` controller (`BlogController.php`), and then point to a specific action in that controller intended to retrieve and display category-specific posts:

```
Route::get('blog/category/{category}', 'BlogController@category');
```

Once defined, when a user accesses a URI such as `blog/category/mysql`, Laravel would execute the `Blog` controller's `category` action, making `mysql` available to the `category` action by expressly defining a method input argument as demonstrated here:

```
public function category($category)
{
  return view('blog.category')->with('category', $category);
}
```

In this example the `$category` variable is subsequently being passed into the view. Admittedly I'm getting ahead of things here because views and view variables haven't yet been introduced, so if this doesn't make any sense don't worry as these concepts are introduced later in the chapter.

If you need to pass along multiple parameters just specify them within the route definition as before:

```
Route::get('blog/category/{category}/{subcategory}', 'BlogController@category');
```

Then in the corresponding action be sure to define the input arguments in the same order as the parameters are specified in the route definition:

```
public function category($category, $subcategory)
{
  return view('blog.category')
    ->with('category', $category)
    ->with('subcategory', $subcategory);
}
```

Creating Route Aliases

Route aliases (also referred to as *named routes*) are useful because you can use the route name when creating links within your views, allowing you to later change the route path in the annotation without worrying about breaking the associated links. For instance the following definition associates a route with an alias named `blog.category`:

```
Route::get('blog/category/{category}',
  ['as' => 'blog.category', 'uses' => 'BlogController@category']);
```

Once defined, you can reference routes by their alias using the `URL::route` method:

```
<a href="{{ URL::route('blog.category', ['category' => 'mysql']) }}">MySQL</a>
```

Listing Routes

As your application grows it can be easy to forget details about the various routes. You can view a list of all available routes using Artisan's `route:list` command:

```
$ php artisan route:list
+--------+----------------------+------+---------------------------------------+-\
-----------+
| Domain | URI                  | Name | Action                                | \
Middleware |
+--------+----------------------+------+---------------------------------------+-\
-----------+
|        | GET|HEAD /          |      | App\..\HomeController@index           | \
           |
|        | POST auth/register   |      | App\..\Auth\AuthController@register   | \
guest      |
|        | POST auth/login      |      | App\..\Auth\AuthController@login      | \
guest      |
|        | GET|HEAD auth/logout |      | App\..\Auth\AuthController@logout     | \
           |
|        | ...                  |      | ...                                   | \
guest      |
+--------+----------------------+------+---------------------------------------+-\
-----------+
```

I'll talk more about the various aspects of this output as the book progresses.

Caching Routes

Laravel 5 introduces route caching, an optimization step that serializes your route definitions and places the results in a single file (`storage/framework/routes.php`). Once serialized, Laravel no longer has to parse the route definitions with each request in order to initiate the associated response. To cache your routes you'll use the `route:cache` command:

```
$ php artisan route:cache
Route cache cleared!
Routes cached successfully!
```

The cached route definitions are stored in the file `storage/framework/routes.php`. If you subsequently add, edit or delete a route definition you'll need to clear and rebuild the cache. To clear the route cache, execute the `route:clear` command:

```
$ php artisan route:clear
```

This command will delete `storage/framework/routes.php`, causing Laravel to return to parsing the route definitions until you again decide to cache the routes.

Introducing Route Annotations

Early in Laravel 5's development cycle a new feature known as *route annotations* was introduced. Route annotations offered developers the opportunity to define routes by annotating the action associated with the route within a comment located directly above the action. For instance you could use annotations to tell Laravel you'd like the `Welcome` controller's `index` action to map to the application root URL like so:

```
/**
 * @Get("/")
 */
public function index()
{
  return view('welcome');
}
```

This feature resulted in quite a bit of controversy, and the Laravel developers eventually decided to remove the feature from the core distribution and instead make it available through a third-party package. If you're familiar with route annotations from use within other frameworks and would like to use it in conjunction with Laravel 5, head on over to the Laravel Annotations package's GitHub page[72] and carefully review the installation and configuration instructions found in the `README`.

While route annotations are an interesting new feature, I prefer to use the `routes.php` file as it offers a centralized location for easily examining all defined routes. That said while in this section I'll introduce a few key route annotation features, for the remainder of the book I'll rely on the `routes.php` file for defining TODOParrot routes.

Defining URL Parameters Using Annotations

Defining URL parameter placeholders within route annotations is easy; just delimit the parameter with curly brackets:

[72] https://github.com/LaravelCollective/annotations

```
/**
 * @Get("/blog/category/{category}")
 */
```

You'll access the `category` parameter via a method argument as explained in the earlier section, "Defining Custom Routes".

Defining Route Aliases Using Annotations

You can define route aliases like so:

```
/**
 * @Get("/blog/category/{category}", as="blog.category")
 */
```

You can then use the alias name within your views as described in the earlier section, "Creating Route Aliases".

Creating Your First Controller

The `Welcome` controller generated along with every new Laravel project nicely serves the purpose of managing the application home page, but you'll of course want to create other controllers to manage other areas of the application. For instance no self-respecting web application would omit an "About" page, so let's create a controller for managing this content. We can easily generate the controller using Artisan's `make:controller` command:

```
$ php artisan make:controller --plain AboutController
Controller created successfully.
```

When generating controllers with `make:controller`, Laravel will by default stub out the various actions comprising a RESTful resource (more about this in the next chapter). You can override this behavior and instead create an empty controller by passing along the `--plain` option as demonstrated above. With the controller generated, let's create the `index` action and corresponding view. Begin by opening the newly created controller (`app/Http/Controllers/AboutController.php`) and modifying the `index` action (created by default with every new controller) to look like this:

```
function index()
{
  return view('about.index');
}
```

Next, create a new directory named `about` in your `resources/views` directory, and inside it create a file named `index.blade.php`, adding the following contents to it:

```
<h1>About TODOParrot</h1>

<p>
TODOParrot is the first tropically-themed TODO List
application to hit the market.
</p>
```

Save the changes, and next open up the `routes.php` file. If you only plan on managing a single route in this controller, you could define a specific route like this:

```
Route::get('/about', 'AboutController@index');
```

Save the changes to `routes.php` and navigate to `http://homestead.app/about` to see the newly created view!

Introducing the Blade Template Engine

One of the primary goals of MVC frameworks is *separation of concerns*. We don't want to pollute views with database queries and other logic, and don't want the controllers and models to make any presumptions regarding how data should be formatted. Because the views are intended to be largely devoid of any programming language syntax, they can be easily maintained by a designer who might not otherwise have any programming experience. But certainly *some* logic must be found in the view, otherwise we would be pretty constrained in terms of what could be done with the data intended to be presented within the page. Most frameworks attempt to achieve a happy medium in this regards, providing a simplified facility for embedding logic into a view. Such facilities are known as *template engines*. Laravel's template engine is called *Blade*. Blade offers all of the features one would expect of a template engine, including inheritance, output filtering, if conditionals, and looping.

In order for Laravel to recognize a Blade-augmented view, you'll need to use the `.blade.php` extension. You've already worked with several Blade-enabled views (`welcome.blade.php` for instance), however we have yet to actually take advantage of any Blade-specific features! Let's work through a number of different examples involving Blade syntax and the `welcome.blade.php` view.

Displaying Variables

Your views will often include dynamic, typically created within a view's corresponding controller action. For instance, suppose you wanted to pass the name of a list into a view. Because we haven't yet discussed how to create new controllers and actions, let's continue experimenting with the existing TODOParrot `WelcomeController` (`app/Http/Controllers/WelcomeController.php`) and corresponding view (`resources/views/welcome.blade.php`). Open up `WelcomeController.php` and modify the `index` action to look like this:

```
public function index()
{
  return view('welcome')->with('name', 'San Juan Vacation');
}
```

Save these changes and then open `welcome.blade.php` (`resources/views`), and add the following line anywhere within the file:

```
{{-- Output the $name variable. --}}
<p>{{ $name }}</p>
```

Reload the home page and you should see "San Juan Vacation" embedded into the view! As an added bonus, I included an example of a Blade comment (Blade comments are enclosed within the `{{--` and `--}}` tags).

You can also use a cool shortcut known as a *magic method* to identify the variable name:

```
public function index()
{
    $name = 'San Juan Vacation';
    return view('welcome')->withName($name);
}
```

This variable is then made available to the view just as before:

```
<p>{{ $name }}</p>
```

Escaping Dangerous Input

Because web applications commonly display contributed user data (product reviews, blog comments, etc.), you must take great care to ensure malicious data isn't inserted into the database. You'll typically do this by employing a multi-layered filter, starting by properly validating data (discussed in Chapter 3) and additionally escaping potentially dangerous data (such as JavaScript code) prior to embedding it into a view. In earlier versions of Laravel this was automatically done using the double brace syntax presented in the previous example, meaning if a malicious user attempted to inject JavaScript into the view, the HTML tags would be escaped. Here's an example:

```
{{ 'My list <script>alert("spam spam spam!")</script>' }}
```

Rather than actually executing the JavaScript `alert` function when the string was rendered to the browser, Laravel would instead rendered the string as text:

```
My list &lt;script&gt;alert("spam spam spam!")&lt;/script&gt;
```

In Laravel 4 if you wanted to output raw data and therefore *allow* in this case the JavaScript code to execute, you would use triple brace syntax:

```
{{{ 'My list <script>alert("spam spam spam!")</script>' }}}
```

Perhaps because at a glance it was too easy to confuse `{{{...}}}` and `{{...}}`, this syntax has changed in Laravel 5. In Laravel 5 you'll use the `{!!` and `!!}` delimiters to output raw data:

```
{!! 'My list <script>alert("spam spam spam!")</script>' !!}
```

Of course, you should only output raw data when you're absolutely certain it does not originate from a potentially dangerous source.

Displaying Multiple Variables

You'll certainly want to pass multiple variables into a view; you can do so exactly as demonstrated in the earlier example using the `with` method:

```
public function index()
{

    $data = array('name' => 'San Juan',
                  'date' => date('Y-m-d'));

    return view('welcome')->with($data);

}
```

To view both the `$name` and `$date` variables within the view, update your view to include the following:

```
You last visited {{ $name }} on {{ $date }}.
```

You could also use multiple `with` methods, like so:

```
return view('welcome')->with('name', 'San Juan Vacation')->with('date', date('Y-\
m-d'));
```

Logically this latter approach could get rather unwieldy if you needed to pass along more than two variables. You can save some typing by using PHP's `compact()` function:

```
$name = 'San Juan Vacation';
$date = date('Y-m-d');
return view('welcome', compact('name', 'date'));
```

The `$name` and `$date` variables defined in your action will then automatically be made available to the view. If this is confusing see the PHP manual's `compact()` function documentation at http://php.net/compact[73].

Determining Variable Existence

There are plenty of occasions when a particular variable might not be set at all, and if not you want to output a default value. You can use the following shortcut to do so:

```
Welcome, {{ $name or 'California' }}
```

Looping Over an Array

TODOParrot users spend a lot of time working with lists, such as the list of tasks comprising a list, or a list of their respective TODO lists. These various list items are stored as records in the database (we'll talk about database integration in Chapter 3), retrieved within a controller action, and then subsequently iterated over within the view. Blade supports several constructs for looping over arrays, including `@foreach` which I'll demonstrate in this section (be sure to consult the Laravel documentation for a complete breakdown of Blade's looping capabilities). Let's demonstrate each by iterating over an array into the `index` view. Begin by modifying the `WelcomeController.php` `index` method to look like this:

[73] http://php.net/compact

```
public function index()
{
  $lists = array('Vacation Planning', 'Grocery Shopping', 'Camping Trip');
  return view('welcome')->with('lists', $lists);
}
```

Next, update the `index` view to include the following code:

```
<ul>
  @foreach ($lists as $list)
    <li>{{ $list }}</li>
  @endforeach
</ul>
```

When rendered to the browser, you should see a bulleted list consisting of the three items defined in the `$lists` array.

Because the array could be empty, consider using the `@forelse` construct instead:

```
<ul>
  @forelse ($lists as $list)
    <li>{{ $list }}</li>
  @empty
    <li>You don't have any lists saved.</li>
  @endforelse
</ul>
```

This variation will iterate over the `$lists` array just as before, however if the array happens to be empty the block of code defined in the `@empty` directive will instead be executed.

If Conditional

In the previous example I introduced the `@forelse` directive. While useful, for readability reasons I'm not personally a fan of this syntax and instead use the `@if` directive to determine whether an array contains data:

```
<ul>
  @if (count($lists) > 0)
    @foreach ($lists as $list)
      <li>{{ $list }}</li>
    @endforeach
  @else
    <li>You don't have any lists saved.</li>
  @endif
</ul>
```

Blade also supports the if-elseif-else construct:

```
@if (count($lists) > 1)
  <ul>
    @foreach ($lists as $list)
      <li>{{ $list }}</li>
    @endforeach
  </ul>
@elseif (count($lists) == 1)
  <p>
    You have one list: {{ $lists[0] }}.
  </p>
@else
  <p>You don't have any lists saved.</p>
@endif
</ul>
```

Managing Your Application Layout

The typical web application consists of a design elements such as a header and footer, and these elements are generally found on every page. Because eliminating redundancy is one of Laravel's central tenets, clearly you won't want to repeatedly embed elements such as the site logo and navigation bar within every view. Instead, you'll use Blade syntax to create a *master layout* that can then be inherited by the various page-specific views. To create a layout, first create a directory within `resources/views` called `layouts`, and inside it create a file named `master.blade.php`. Add the following contents to this newly created file:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Welcome to TODOParrot</title>
</head>
<body>

  @yield('content')

</body>
</html>
```

The `@yield` directive identifies the name of the *section* that should be embedded into the template. This is best illustrated with an example. After saving the changes to `master.blade.php`, open the `welcome.blade.php` file and modify its contents to look like this:

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

@section('content')

<h1>Welcome to TODOParrot</h1>

<p>
  TODOParrot is the ultimate productivity application for
  tropically-minded users.
</p>

@endsection
```

The `@extends` directive tells Laravel which layout should be used. Note how dot notation is used to represent the path, so for instance `layouts.master` translates to `layouts/master`. You specify the layout because it's possible your application will employ multiple layouts, for instance one sporting a sidebar and another without.

After saving the changes reload the TODOParrot home page and you'll see that the `index.blade.php` view is wrapped in the HTML defined in `master.blade.php`, with the HTML found in the `@section` directive being inserted into `master.blade.php` where the `@yield('content')` directive is defined.

Defining Multiple Layout Sections

A layout can identify multiple sections. For instance many web applications employ a main content area and a sidebar. In addition to the usual header and footer the layout might include some globally

available sidebar elements, but you probably want the flexibility of appending view-specific sidebar content. This can be done using multiple `@section` directives in conjunction with `@show` and `@parent`. For reasons of space I'll just include the example layout's `<body>`:

```
<body>

  <div class="container">

    <div class="col-md-9">
      @yield('content')
    </div>

    <div class="col-md-3">
      @section('advertisement')
      <p>
        Jamz and Sun Lotion Special $29!
      </p>
      @show
    </div>
  </div>
</body>
```

You can think of the `@show` directive as a shortcut for closing the section and then immediately yielding it:

```
@endsection
@yield('advertisement')
```

The view can then also reference `@section('advertisement')`, additionally referencing the `@parent` directive which will cause anything found in the view's sidebar section to be *appended* to anything found in the layout's sidebar section:

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

@section('content')
  <h1>Welcome to TODOParrot!</h1>
@endsection

@section('advertisement')
  @parent
    <p>
```

```
    Buy the TODOParrot Productivity guide for $10!
    </p>
@endsection
```

Once this view is rendered, the `advertisement` section would look like this:

```
<p>
  Jamz and Sun Lotion Special $29!
</p>
<p>
  Buy the TODOParrot Productivity guide for $10!
</p>
```

If you would rather replace (rather than append to) the parent section, just eliminate the `@parent` directive reference.

Taking Advantage of View Partials

Suppose you wanted to include a recurring widget within several different areas of the application. This bit of markup is fairly complicated, such as a detailed table row, and you assume it will be subject to considerable evolution in the coming weeks. Rather than redundantly embed this widget within multiple locations, you can manage this widget within a separate file (known as a *view partial*) and then include it within the views as desired. For instance, if you wanted to manage a complex table row as a view partial, create a file named for instance `row.blade.php`, placing this file within your `resources/views` directory (I prefer to manage mine within a directory named `partials` found in `resources/views`). Add the table row markup to the file:

```
<tr style="padding-bottom: 5px;">
  <td>
  {{ $link->name }}
  </td>
</tr>
```

Notice how I'm using a view variable (`$link`) in the partial. When importing the partial into your view you an optionally pass a variable into the view as demonstrated here:

```
<table class="table borderless">
@foreach ($links as $link)

  @include('partials.linkrow', array('link' => $link))

@endforeach
</table>
```

Integrating Images, CSS and JavaScript

Your project images, CSS and JavaScript should be placed in the project's `public` directory. While you could throw everything into `public`, for organizational reasons I prefer to create `images`, `css`, and `javascript` directories. Regardless of where in the `public` directory you places these files, you're free to reference them using standard HTML or can optionally take advantage of a few helpers via the Laravel HTML package[74]. For instance, the following two statements are identical:

```
<img src="/images/logo.png" alt="TODOParrot logo" />

{!! HTML::image('images/logo.png', 'TODOParrot logo') !!}
```

Similar HTML component helpers are available for CSS and JavaScript. Again, you're free to use standard HTML tags or can use the facade. The following two sets of statements are identical:

```
<link rel="stylesheet" href="/css/app.min.css">
<script src="/javascript/jquery-1.10.1.min.js"></script>
<script src="/javascript/bootstrap.min.js"></script>

{!! HTML::style('css/app.min.css') !!}
{!! HTML::script('javascript/jquery-1.10.1.min.js') !!}
{!! HTML::script('javascript/bootstrap.min.js') !!}
```

If you want to take advantage of the HTML helpers, you'll need to install the `LaravelCollective/HTML` package. This was previously part of the native Laravel distribution, but has been moved into a separate package as of version 5. Fortunately, installing the package is easy. First, add the `LaravelCollective/HTML` package to your `composer.json` file:

[74] https://github.com/LaravelCollective/html

```
"require": {
  ...
  "laravelcollective/html": "~5.0"
},
```

Save the changes and run `composer install` to install the package. Next, add the following line to the `providers` array found in your `config/app.php` file:

```
'Collective\Html\HtmlServiceProvider',
```

Next, add the following line to the `config/app.php` `aliases` array:

```
'HTML' => 'Collective\Html\HtmlFacade'
```

With these changes in place, you can begin using the `LaravelCollective/HTML` package. Be sure to check out the GitHub README[75] of the Laravel documentation for a list of available helpers.

Integrating the Bootstrap Framework

Bootstrap[76] is a blessing to design-challenged web developers like yours truly, offering an impressively comprehensive and eye-appealing set of widgets and functionality. Being a particularly design-challenged developer I use Bootstrap as the starting point for all of my personal projects, often customizing it with a Bootswatch theme[77]. TODOParrot is no exception, taking advantage of both Bootstrap and the Bootswatch Flatly[78] theme.

The Bootstrap CSS source files (in Less format) are now automatically included with every new Laravel project. You'll find all of the various Less files in `resources/assets/less/bootstrap`. Note how the `app.less` file found in `resources/assets/less` automatically imports all of these files, meaning if you include `app.less` in your `master.blade.php` header, all of Bootstrap's files will additionally be incorporated into the compiled `app.css` file (see the later section "Introducing Elixir" for more information about Less compilation).

At the time of this writing Laravel did *not* include Bootstrap's JavaScript files, so you'll need to download and integrate those separately if you wish to take advantage of those features.

If you'd like to use the Bootstrap CSS included by default with each Laravel project, then incorporating the framework into your Laravel application is as easy as compiling the `app.less` file and incorporating it into your layout header. However, you could also alternatively (and likely preferably) use Bootstrap's recommended CDNs (Content Delivery Network) to add Bootstrap and jQuery (jQuery is required to take advantage of the Bootstrap JavaScript plugins:

[75] https://github.com/LaravelCollective/html
[76] http://getbootstrap.com/
[77] http://bootswatch.com/
[78] http://bootswatch.com/flatly/

```
<head>
    ...
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/b\
ootstrap.min.css">
    <script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>
    <script src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js">\
</script>
</head>
```

Once added you're free to begin taking advantage of the various CSS widgets and JavaScript plugins found in the Bootstrap documentation[79]. For instance try adding a stylized hyperlink to the `welcome.blade.php` view just to confirm everything is working as expected:

```
<a href="http://www.wjgilmore.com" class="btn btn-success">W.J. Gilmore, LLC</a>
```

Integrating the Bootstrapper Package

At the time of this writing, the Bootstrapper package is no longer compatible with Laravel 5, however I'm expect this matter to be resolved in the very near future and will update the installation instructions found in this section as soon as I have more information. In the meantime consider the installation instructions found in this section to be outdated.

Using Bootstrap as described above is perfectly fine, and in fact I do exactly that in TODOParrot. However, for those of you who prefer to use view helpers whenever possible, check out Bootstrapper[80], a great package created by Patrick Tallmadge[81]. Once installed you can use a variety of helpers to integrate Bootstrap widgets into your views. For instance the following helper will create a hyperlinked button:

```
{!! Button::success('Success') !!}
```

To install Bootstrapper, open your project's `composer.json` file and add the following line to the `require` section:

[79] http://getbootstrap.com/
[80] http://bootstrapper.aws.af.cm/
[81] https://github.com/patricktalmadge

```
"require": {
  ...
  "patricktalmadge/bootstrapper": "dev-laravel-5"
},
```

Note the use of a Bootstrapper development branch; this branch should at the time of this writing be used with Laravel 5, although presumably these changes will shortly be merged into the master branch.

Save the changes and then open up `config/app.php` and locate the `providers` array, adding the following line to the end of the array:

```
'providers' => array(
  ...
  'Bootstrapper\BootstrapperServiceProvider'
),
```

By registering the Bootstrapper service provider, Laravel will known to initialize Bootstrapper alongside any other registered service providers, making its functionality available to the application. Next, search for the `aliases` array also located in the `app/config/app.php` file. Paste the following rather lengthy bit of text into the bottom of the array:

```
'Accordion' => 'Bootstrapper\Facades\Accordion',
'Alert' => 'Bootstrapper\Facades\Alert',
'Badge' => 'Bootstrapper\Facades\Badge',
'Breadcrumb' => 'Bootstrapper\Facades\Breadcrumb',
'Button' => 'Bootstrapper\Facades\Button',
'ButtonGroup' => 'Bootstrapper\Facades\ButtonGroup',
'Carousel' => 'Bootstrapper\Facades\Carousel',
'ControlGroup' => 'Bootstrapper\Facades\ControlGroup',
'DropdownButton' => 'Bootstrapper\Facades\DropdownButton',
'Form' => 'Bootstrapper\Facades\Form',
'Helpers' => 'Bootstrapper\Facades\Helpers',
'Icon' => 'Bootstrapper\Facades\Icon',
'InputGroup' => 'Bootstrapper\Facades\InputGroup',
'Image' => 'Bootstrapper\Facades\Image',
'Label' => 'Bootstrapper\Facades\Label',
'MediaObject' => 'Bootstrapper\Facades\MediaObject',
'Modal' => 'Bootstrapper\Facades\Modal',
'Navbar' => 'Bootstrapper\Facades\Navbar',
'Navigation' => 'Bootstrapper\Facades\Navigation',
'Panel' => 'Bootstrapper\Facades\Panel',
```

```
'ProgressBar' => 'Bootstrapper\Facades\ProgressBar',
'Tabbable' => 'Bootstrapper\Facades\Tabbable',
'Table' => 'Bootstrapper\Facades\Table',
'Thumbnail' => 'Bootstrapper\Facades\Thumbnail',
```

Adding these aliases will save you the hassle of having to type the entire namespace when referencing one of the Bootstrap components. Save these changes and then run `composer update` from your project's root directory to install Bootstrapper. With Bootstrapper installed, all that remains to begin using Bootstrap is to add Bootstrap and jQuery to your project layout. Open the `master.blade.php` file we created in the earlier section and add the following lines to the layout `<head>`:

```
<link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/boots\
trap.min.css">
<script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>
<script src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js"></sc\
ript>
```

Save the changes, and you're ready to begin taking advantage of Bootstrapper's CSS styling and jQuery plugins!

Introducing Elixir

Writing code is but one of many tasks the modern developer has to juggle when working on even a simple web application. You'll want to compress images, minify CSS and JavaScript files, remove debugging statements, run unit tests, and perform countless other mundane duties. Keeping track of these responsibilities let alone ensuring you remember to carry them all out is a pretty tall order, particularly because you're presumably devoting the majority of your attention to creating and maintaining great application features.

The Laravel 5 developers hope to reduce some of the time and hassle associated with these sort of tasks by providing a new API called Laravel Elixir[82]. The Elixir API integrates with Gulp, providing an easy solution for compiling your Laravel project's Less[83], Sass[84] and CoffeeScript[85], and perform any other such administrative task. In this section I'll show you how to create and execute several Elixir tasks in conjunction with your Laravel application. But first because many readers are likely not familiar with Gulp I'd like to offer a quick introduction, including instructions for installing Gulp and it's dependencies.

[82] https://github.com/laravel/elixir
[83] http://lesscss.org/
[84] http://sass-lang.com/
[85] http://coffeescript.org/

Introducing Gulp

Gulp[86] is a powerful open source build system you can use to automate all of the aforementioned tasks and many more. You'll automate away these headaches by writing *Gulp tasks*, and can save a great deal of time when doing so by integrating one or more of the hundreds of available Gulp plugins[87]. In this section I'll show you how to install and configure Gulp for subsequent use within Elixir.

Installing Gulp

Because Gulp is built atop Node.js[88], you'll first need to install Node. No matter your operating system this is easily done by downloading one of the installers via the Node.js website[89]. If you'd prefer to build Node from source you can download the source code via this link. If like me you're a Mac user, you can install Node via Homebrew. Linux users additionally likely have access to Node via their distribution's package manager.

Once installed you can confirm Node is accessible via the command-line by retrieving the Node version number:

```
1 $ node -v
2 v0.10.36
```

Node users have access to a great number of third-party libraries known as Node Packaged Modules (NPM). You can install these modules via the aptly-named `npm` utility. We'll use `npm` to install Gulp:

```
1 $ npm install -g gulp
```

Once installed you should be able to execute Gulp from the command-line:

```
1 $ gulp -v
2 [14:12:51] CLI version 3.8.10
```

With Gulp installed it's time to install Elixir!

Installing Elixir

Laravel 5 applications automatically include a file named `package.json` which resides in the project's root directory. This file looks like this:

[86] http://gulpjs.com/
[87] http://gulpjs.com/plugins/
[88] http://nodejs.org
[89] http://nodejs.org/download/

```
{
  "devDependencies": {
    "gulp": "^3.8.8",
    "laravel-elixir": "*"
  }
}
```

Node's `npm` package manager uses `package.json` to learn about and install a project's Node module dependencies. As you can see, a default Laravel project requires two Node packages: `gulp` and `laravel-elixir`. You can install these packages locally using the package manager like so:

```
$ npm install
```

Once complete, you'll find a new directory named `node_modules` has been created within your project's root directory, and within in it you'll find the `gulp` and `laravel-elixir` packages.

Creating Your First Elixir Task

Your Laravel project includes a default `gulpfile.js` which defines your Elixir-flavored Gulp tasks. Inside this file you'll find an example Gulp task:

```
elixir(function(mix) {
    mix.less('app.less');
});
```

The `mix.less` task compiles a Less[90] file, in this case the file named `app.less`. This file resides in `resources/assets/less`, and looks like this:

```
@import "bootstrap/bootstrap";

@btn-font-weight: 300;
@font-family-sans-serif: "Roboto", Helvetica, Arial, sans-serif;

body, label, .checkbox label {
  font-weight: 300;
}
```

You're free to add other tasks to this function, meaning you can easily carry out multiple annoying and repetitive tasks in just a few keystrokes. You can execute these tasks by running `gulp` from within your project root directory:

[90] http://lesscss.org/

```
$ gulp
[13:16:18] Using gulpfile ~/Software/dev.todoparrot.com/gulpfile.js
[13:16:18] Starting 'default'...
[13:16:18] Starting 'less'...
[13:16:19] Finished 'default' after 480 ms
[13:16:20] gulp-notify: [Laravel Elixir]
[13:16:20] Finished 'less' after 1.52 s
```

By executing `gulp` we've compiled `app.less`, saving the output to a file named `app.css` which resides in your project's `public/css` directory. Of course, in order to actually use the styles defined in the `app.css` file you'll need to reference it within your layout:

```
<link rel="stylesheet" href="/css/app.css">
```

Keep in mind that Elixir does not minify compiled CSS by default. You can however minify it by passing the `--production` option to `gulp`:

```
$ gulp --production
```

Additionally, the compiled `app.css` file will additionally contain *all* of the compiled Bootstrap CSS definitions. In some cases you'll only want to use a subset of these definitions, so be sure to do some research regarding how you can selectively determine which files are compiled.

Compiling Your JavaScript Assets

You'll likely also want to manage your JavaScript assets. For instance if you use CoffeeScript[91], you'll place your CoffeeScript files in `resources/assets/coffee` (you'll need to create this directory). Here's a simple CoffeeScript statement which will display one of those annoying alert boxes in the browser:

alert "Hi I am annoying"

Save this statement to `resources/assets/coffee/test.coffee`. Next, modify your `gulpfile.js` file to look like this:

```
elixir(function(mix) {
    mix.less('app.less');
    mix.coffee();
});
```

Incidentally, you could also chain the commands together like so:

[91] http://coffeescript.org/

```
elixir(function(mix) {
    mix.less('app.less').coffee();
});
```

Save the changes and run `gulp` again:

```
$ gulp
[14:40:25] Using gulpfile ~/Software/dev.todoparrot.com/gulpfile.js
[14:40:25] Starting 'default'...
[14:40:25] Starting 'less'...
[14:40:26] Finished 'default' after 478 ms
[14:40:27] gulp-notify: [Laravel Elixir]
[14:40:27] Finished 'less' after 1.88 s
[14:40:27] Starting 'coffee'...
[14:40:27] gulp-notify: [Laravel Elixir]
[14:40:27] Finished 'coffee' after 236 ms
```

You'll see that a directory named `js` has been created inside `public`. Inside this directory you'll find the file `test.js` which contains the following JavaScript code:

```
(function() {
  alert("Hi I am annoying");

}).call(this);
```

Watching for Changes

Because you'll presumably be making regular tweaks to your CSS and JavaScript and will want to see the results in your development browser, consider using Elixir's `watch` command to automatically execute `gulpfile.js` anytime your assets change:

```
$ gulp watch
```

Other Elixir Tasks

Less and CoffeeScript compilation are but two of several Elixir features you can begin taking advantage of right now. Be sure to check out the Elixir README[92] for an extended list of capabilities.

While Elixir is still very much a work in progress and documentation remains slim, this tool already holds bunches of potential. Stay tuned as I'll be sure to expand this section significantly in future revisions to include additional examples.

[92] https://github.com/laravel/elixir

Testing Your Views

Earlier versions of Laravel automatically included the BrowserKit[93] and DomCrawler[94] packages, both of which are very useful for functionally testing various facets of your application in conjunction with PHPUnit. Among their many capabilities you can write and execute tests that interact with your Laravel application in the very same way a user would. For instance you might wish to confirm a page is rendering and displaying a particular bit of content, or ensure that a particular link is taking users to a specific destination.

Fortunately, you can easily add these capabilities to your application via Composer via the powerful Goutte[95] package. Goutte is a web crawling library that can mimic a user's behavior in many important ways, and we an use it in conjunction with PHPUnit to functionally test the application. Open your project's `composer.json` file and add the following line:

```
"require-dev": {
  ...
  "fabpot/goutte": "2.*"
},
```

Save the changes and run `composer update` to install Goutte.

For organizational purposes it makes sense to create a new test file for each controller/view pair you plan on testing (and breaking it down even further if necessary), so let's create a new file named `WelcomeTest.php`, placing it in the `tests` directory:

```
<?php

use Goutte\Client;

class WelcomeTest extends TestCase {

}
```

Inside this class create the following test, which will access the home page and confirm the user sees the message `Welcome to TODOParrot`, which is placed inside an `h1` tag:

[93] http://symfony.com/components/BrowserKit
[94] http://symfony.com/doc/current/components/dom_crawler.html
[95] https://github.com/FriendsOfPHP/Goutte

```
public function testUserSeesWelcomeMessage()
{

$client = new Client();
$crawler = $client->request('GET', 'http://homestead.app/');

$this->assertEquals(200, $client->getResponse()->getStatus());

$this->assertCount(1,
  $crawler->filter('h1:contains("Welcome to TODOParrot")'));

}
```

In this test we're actually executing two assertions: the first (`assertEquals()`) confirms that the response returned a 200 status code (successful request). The second uses the `assertCount` method to confirm there is only one instance of an `h1` tag that contains the welcome message. It's not that we expect there to be multiple instances, but we're particularly concerned about there being one instance and so the `assertCount` is a useful method for such purposes. Of course, when writing these sorts of test you'll need to keep in mind that the user interface is often in a state of constant change, therefore you'll want to reserve the use of such tests to confirming particularly important bits of content.

You can also create tests that interact with the page. For instance, when the user isn't logged into TODOParrot a link to the registration page is provided. The link looks like this:

```
<a href="/about/contact" class="btn btn-primary">Contact Us</a>
```

When the user clicks on this link we clearly want him to be taken to the contact form. Let's confirm that when this link is clicked the user is taken to the `About` controller's `contact` action, and the corresponding view contains the `h1` header "Contact Us":

```
public function testUserClicksContactLinkAndIsTakenToContactPage()
{

  $client = new Client();
  $crawler = $client->request('GET', 'http://homestead.app/');

  $link = $crawler->selectLink('Contact Us')->link();

  $this->assertEquals('http://homestead.app/about/contact',
    $link->getUri());

```

```
12    $crawler = $client->click($link);
13
14    $this->assertCount(1,
15      $crawler->filter('h1:contains("Contact Us")'));
16
17  }
```

In the upcoming chapters we'll expand upon these simple examples, writing automated tests to test forms and user authentication are all implemented to the project specifications.

Conclusion

If you created a TODOParrot list identifying the remaining unread chapters, it's time to mark Chapter 2 off as completed! In the next chapter we'll really dive deep into how your Laravel project's data is created, managed and retrieved. Onwards!