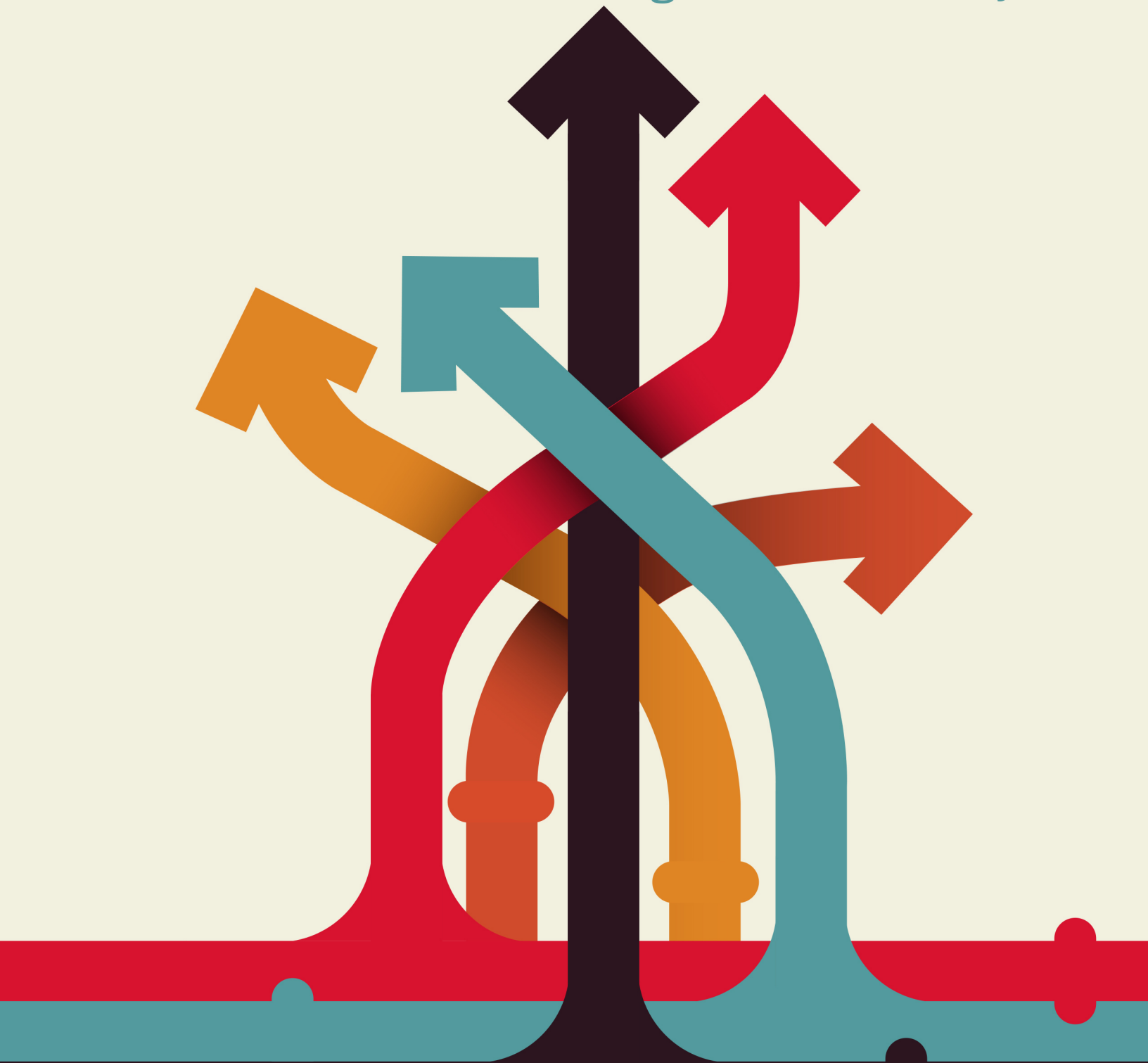# EASY LARAVEL 5

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 2015-08-11

Also By W. Jason Gilmore

Easy Active Record for Rails Developers

Easy E-Commerce Using Laravel and Stripe

Easy React

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

Contents

Introduction

I've spent the vast majority of 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 eleven chapters and an appendix, 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. Integrating Web Forms

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 upload files using a web form and Laravel's fantastic file upload capabilities.

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.

Chapter 9. Creating a Restricted Administration Console

This chapter shows you how to identify certain users as administrators and then grant them access to a restricted web-based administrative console using a prefixed route grouping and custom middleware.

Chapter 10. Introducing the Lumen Microframework

This chapter introduces the new Laravel Lumen microframework. You'll learn all about Lumen fundamentals while building a companion microservice for the TODOParrot companion application!

Chapter 11. Introducing Events

This chapter introduces Laravel Events, showing you how to create event handlers, event listeners, and integrate events into your application logic. You'll also learn all about Laravel 5.1's fascinating event broadcasting capabilities, accompanied by a real-world example.

Appendix A. Deploying Your Laravel Application to DreamHost

In this appendix you'll learn how to configure Capistrano and GitHub to effortlessly deploy a Laravel project to the popular hosting provider DreamHost. Even if you're using another shared hosting provider, much of what is discussed in this appendix will apply to your own specific circumstances.

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.

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

---

[11]http://todoparrot.com

[12]https://github.com/wjgilmore/todoparrot

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

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

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.

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.

> ⚠ I published this book on February 4, 2015, the very same day Laravel 5 officially released. Since then I've made more than one hundred improvements and expansions, and often fix any reported errata within a few days following notification (see http://easylaravelbook.com/changelog/). More recently this includes a *major* book revision to reflect Laravel 5.1 changes. If you find an issue please 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:

```
1  $ vagrant box add laravel/homestead
2  ==> box: Loading metadata for box 'laravel/homestead'
3      box: URL: https://vagrantcloud.com/laravel/homestead
4  ==> box: Adding box 'laravel/homestead' (v0.2.2) for provider: virtualbox
5      box: Downloading: https://vagrantcloud.com/laravel/boxes/homestead/
6      versions/0.2.2/providers/virtualbox.box
7  ==> 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:

```
1  $ composer global require "laravel/homestead=~2.0"
2  Changed current directory to /Users/wjgilmore/.composer
3  ./composer.json has been updated
4  Loading composer repositories with package information
5  Updating dependencies (including require-dev)
6    - Installing symfony/process (v2.6.3)
7      Downloading: 100%
8
9    - Installing laravel/homestead (v2.0.8)
10     Downloading: 100%
11
12 Writing lock file
13 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:

```
1  $ homestead init
2  Creating Homestead.yaml file... ok
3  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:

```
1  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:

```
1  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:

```
1  keys:
2      - 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/

```
1   folders:
2       - map: ~/Code
3         to: /home/vagrant/Code
4
5   sites:
6       - map: homestead.app
7         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:

```
1   folders:
2       - map: /Users/wjgilmore/Software/dev.todoparrot.com
3       - 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:

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

Save these changes, and then run the following command:

```
 1  $ homestead up
 2  Bringing machine 'default' up with 'virtualbox' provider...
 3  ==> default: Importing base box 'laravel/homestead'...
 4  ==> default: Matching MAC address for NAT networking...
 5  ==> default: Checking if box 'laravel/homestead' is up to date...
 6  ...
 7  ==> default: Forwarding ports...
 8  default: 80 => 8000 (adapter 1)
 9  default: 443 => 44300 (adapter 1)
10  default: 3306 => 33060 (adapter 1)
11  default: 5432 => 54320 (adapter 1)
12  default: 22 => 2222 (adapter 1)
13  $
```

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:

```
 1  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:

```
1  $ homestead halt
2  ==> default: Attempting graceful shutdown of VM...
3  $
```

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

```
1  $ 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].:

```
1  $ ssh vagrant@127.0.0.1 -p 2222
2  Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-30-generic x86_64)
3
4   * Documentation:  https://help.ubuntu.com/
5
6    System information as of Thu Jan  8 00:57:20 UTC 2015
7
8    System load:  0.96                Processes:           104
9    Usage of /:   5.0% of 39.34GB   Users logged in:     0
10   Memory usage: 28%                 IP address for eth0: 10.0.2.15
11   Swap usage:   0%                  IP address for eth1: 192.168.33.10
12
13   Graph this data and manage this system at:
14     https://landscape.canonical.com/
15
16   Get cloud support with Ubuntu Advantage Cloud Guest:
17     http://www.ubuntu.com/business/services/cloud
18
19 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/

```
1  vagrant@homestead:~$ ls
2  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 installing the Laravel installer using Composer[30]:

```
1  $ composer global require "laravel/installer=~1.1"
```

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.

After installation you'll be able to create new Laravel project skeletons using the `laravel` utility's `new` command:

```
1  $ laravel new dev.todoparrot.com
2  Crafting application...
3  Application ready! Build something amazing.
```

Next you'll need to create an environment-specific configuration file. This file is used as a central repository for managing your various database connection variables, API keys, and other crucial information. Laravel provides a configuration file template, so all you need to do is create a copy:

```
1  $ cd dev.todoparrot.com
2  $ cp .env.example .env
```

Finally, you'll need to generate an application key. This key is used for managing sensitive data such as cookie encryption:

---

[30]https://getcomposer.org
[31]https://getcomposer.org

```
1  $ php artisan key:generate
```

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:

This command creates a new Laravel skeleton project 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:

- `.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 solution for 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.

---

[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

- `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 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*

---

[36]https://getcomposer.org

[37]http://swiftmailer.org/

[38]http://www.doctrine-project.org/

[39]http://gulpjs.com/

[40]http://www.phpspec.net/

[41]http://phpunit.de/

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.
- `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:

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

Alternatively you could use the built-in PHP server:

```
1  $ php -S localhost:8000 -t public /
2  PHP 5.5.15 Development Server started at Wed Jan  7 20:30:49 2015
3  Listening on http://localhost:8000
4  Document root is /Users/wjgilmore/Code/Laravel/public
5  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.

---

[42] http://github.com/wjgilmore/todoparrot
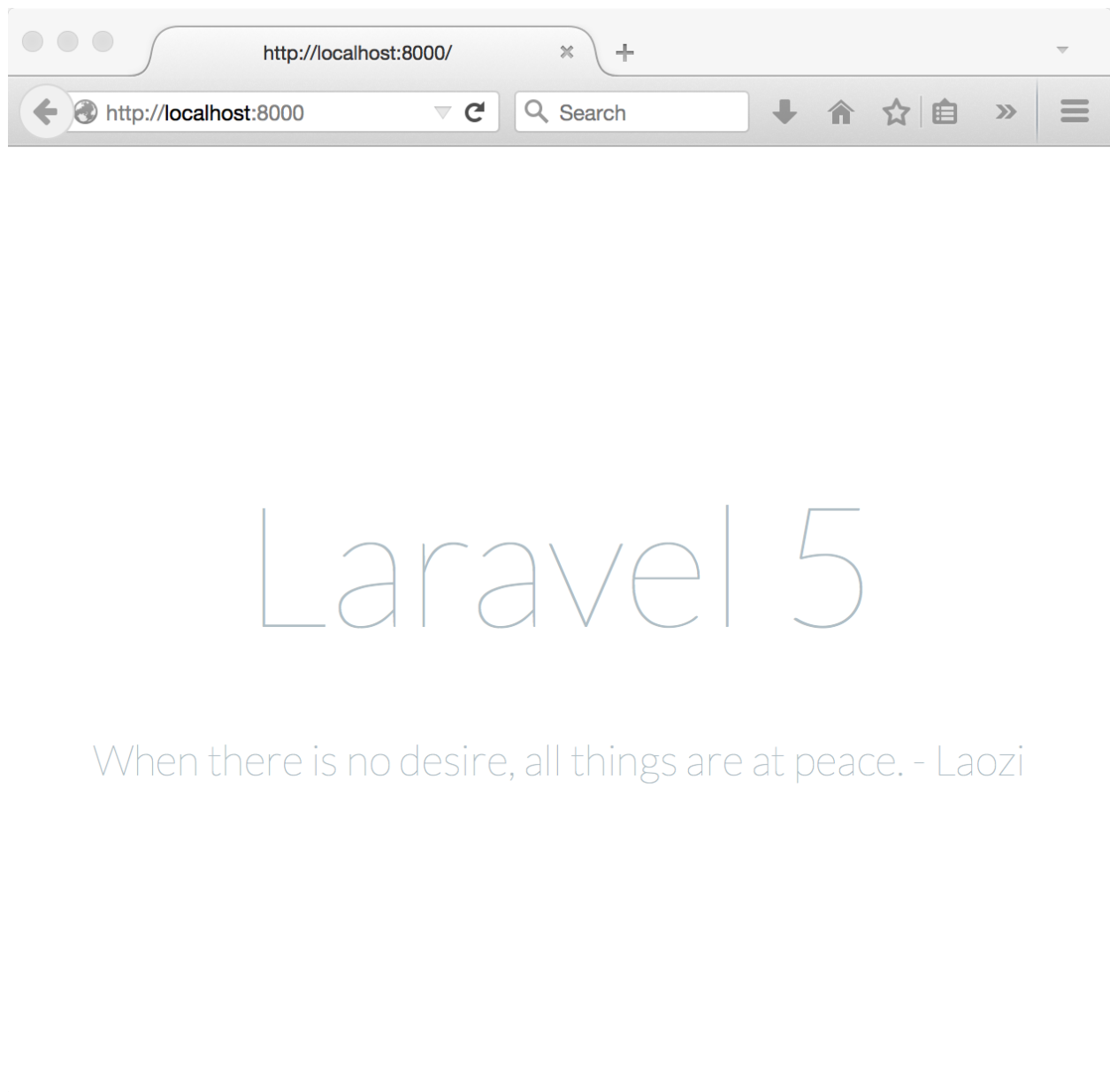
The Laravel splash page

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:

```
1  $ php artisan app:name todoparrot
2  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.
- broadcasting.php: The broadcasting.php is used to configure Laravel 5.1's new event broadcasting feature. I discuss event broadcasting in Chapter 11.
- cache.php: Laravel supports several caching drivers, including filesystem, database, memcached, redis, and others. You'll use the cache.php configuration file to manage various settings specific to these drivers.
- 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 default `.env` file looks like this:

```
1   APP_ENV=local
2   APP_DEBUG=true
3   APP_KEY=SomeRandomString
4
5   DB_HOST=localhost
6   DB_DATABASE=homestead
7   DB_USERNAME=homestead
8   DB_PASSWORD=secret
9
10  CACHE_DRIVER=file
11  SESSION_DRIVER=file
12  QUEUE_DRIVER=sync
13
14  MAIL_DRIVER=smtp
15  MAIL_HOST=mailtrap.io
16  MAIL_PORT=2525
17  MAIL_USERNAME=null
18  MAIL_PASSWORD=null
19  MAIL_ENCRYPTION=null
```

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:

---

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

```
1  'mysql' => [
2      'driver'    => 'mysql',
3      'host'      => env('DB_HOST', 'localhost'),
4      'database'  => env('DB_DATABASE', 'forge'),
5      'username'  => env('DB_USERNAME', 'forge'),
6      'password'  => env('DB_PASSWORD', ''),
7      'charset'   => 'utf8',
8      'collation' => 'utf8_unicode_ci',
9      'prefix'    => '',
10     'strict'    => false,
11 ],
```

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

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. If you'd like to see the dd() function in action we'll need to create a proper controller/action/view configuration for

---

[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

the home page as Laravel 5.1 includes some unfortunate changes to the default home page behavior. If you're new to concepts such as controllers, actions and views I suggest just skipping this section and returning to it after you've read chapter 2.

Begin by creating a file named `WelcomeController.php`, and save it to `app/Http/Controllers`. Add the following contents to it (I'm assuming you've changed your project namespace to `Todoparrot` otherwise you'll need to adjust that first line accordingly):

```php
1   <?php namespace Todoparrot\Http\Controllers;
2
3   class WelcomeController extends Controller {
4
5       public function index()
6       {
7         $items = array(
8           'items' => [
9             'Pack luggage',
10            'Go to airport',
11            'Arrive in San Juan'
12          ]
13        );
14        dd($items);
15        return view('welcome');
16      }
17
18  }
```

Don't get too caught up on what's going on here if you're not familiar with the concept of a controller. Controllers and these class methods (actions) will be formally introduced in Chapter 2. Next, open up app/Http/routes.php . It currently looks like this:

```php
1   Route::get('/', function () {
2       return view('welcome');
3   });
```

Delete that code and replace it with:

```php
1   Route::get('/', ['as' => 'home',
2     'uses' => 'WelcomeController@index']);
```

Finally, create a file named `welcome.blade.php`, placing it inside `resources/views`. Add the following contents to it:

```
1    <h1>This is your new home page</h1>
```

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. Keep in mind that even though we created a display page (`welcome.blade.php`), it won't get displayed because the `dd()` function will immediately force a halt to subsequent execution and just display whatever debugging information provided to it.

○ ○ ○   localhost:8000

◄ ► ☁ ⬈ + 🌐 localhost:8000   ↻ Reader ⓘ ⊙

array(1) { ["items"]=> array(3) { [0]=> string(12) "Pack luggage" [1]=> string(13) "Go to airport" [2]=> string(18) "Arrive in San Juan" } }

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:

```php
public function index()
{

    $items = [
      'Pack luggage',
      'Go to airport',
      'Arrive in San Juan'
    ];
    \Log::debug($items);
    return view('welcome');

}
```

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/

```
1  \Log::info('Just an informational message.');
2  \Log::warning('Something may be going wrong.');
3  \Log::error('Something is definitely going wrong.');
4  \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:

```
1  $monolog = \Log::getMonolog();
2
3  $items = ['Pack luggage', 'Go to airport', 'Arrive in San Juan'];
4
5  $monolog->pushHandler(new \Monolog\Handler\FirePHPHandler());
6
7  $monolog->addInfo('Log Message', array('items' => $items));
```

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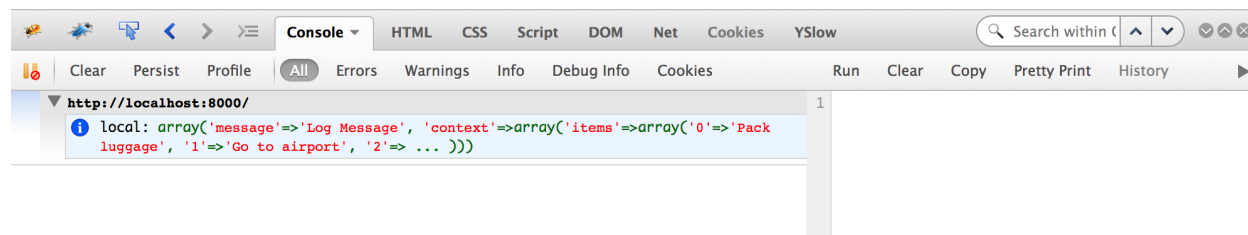
Logging to Firebug via FirePHP

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:

```
1  $ php artisan tinker --env=local
2  Psy Shell v0.5.2 (PHP 5.5.21 â€" cli) by Justin Hileman
3  >>>
```

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:

```
1  >>> $items = ['Pack luggage', 'Go to airport', 'Arrive in San Juan'];
2  => [
3         "Pack luggage",
4         "Go to airport",
5         "Arrive in San Juan"
6     ]
```

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

```
1  >>> var_dump($items);
2  array(3) {
3    [0]=>
4    string(12) "Pack luggage"
5    [1]=>
6    string(13) "Go to airport"
7    [2]=>
8    string(18) "Arrive in San Juan"
9  }
10 => null
11 >>> sort($items);
12 => true
13 >>> $items;
14 => [
15        "Arrive in San Juan",
16        "Go to airport",
17        "Pack luggage"
18    ]
19 >>>
```

---

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

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

```
1  >>> exit
2  Exit:  Goodbye.
3  $
```

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 PsySH 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
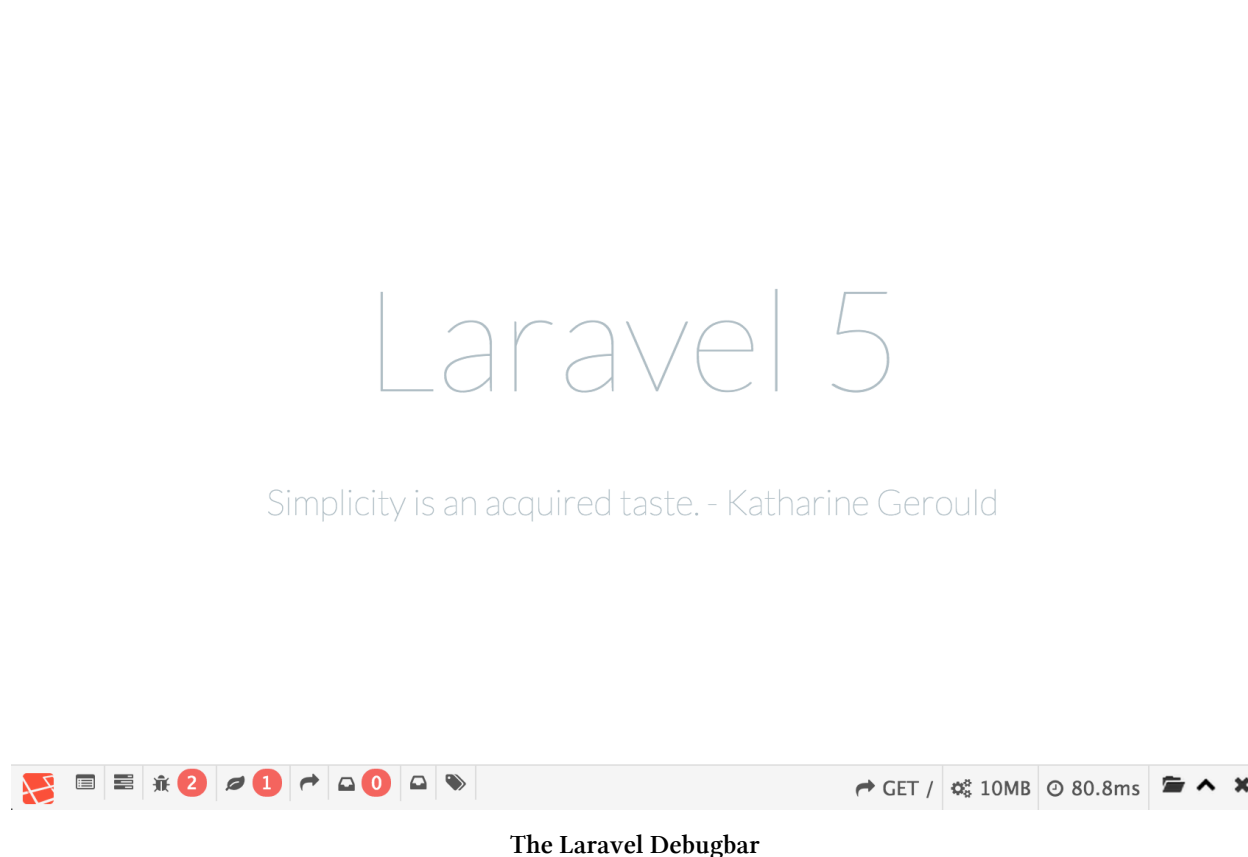
The Debugbar is visually similar to Firebug[66], consisting of multiple tabs that when clicked result in context-related information in a panel situated below the menu. These tabs include:

- **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:

```
1  $ composer require barryvdh/laravel-debugbar
2  Using version ~2.0 for barryvdh/laravel-debugbar
3  ./composer.json has been updated
4  ...
5  Writing lock file
6  Generating autoload files
7  $
```

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

```
1  'providers' => [
2      ...
3      'Barryvdh\Debugbar\ServiceProvider'
4  ],
5
6  ...
7
8  'aliases' => [
9      ...
10     'Debugbar' => 'Barryvdh\Debugbar\Facade'
11 ]
```

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

---

[66]http://getfirebug.com

```
1  $ 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! Keep in mind the Debugbar will only render when used in conjunction with an endpoint that actually renders a view to the browser.

The Laravel Debugbar is tremendously useful as it provides easily accessible insight into several key aspects of your application. Additionally, you can use the Messages panel as a convenient location for viewing log messages. Logging to the Debugbar is incredibly easy, done using the Debugbar facade. Add the following line to the `Welcome` controller's `index` action (`app/Http/Controllers/WelcomeController.php`):

```
1  \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:

```
1  "require-dev": {
2    "phpunit/phpunit": "~4.0"
3  },
```

> **ℹ** 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/

```
1  $ vendor/bin/phpunit --version
2  PHPUnit 4.7.2 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
1  <?php
2
3  use Illuminate\Foundation\Testing\WithoutMiddleware;
4  use Illuminate\Foundation\Testing\DatabaseMigrations;
5  use Illuminate\Foundation\Testing\DatabaseTransactions;
6
7  class ExampleTest extends TestCase
8  {
9      /**
10      * A basic functional test example.
11      *
12      * @return void
13      */
14      public function testBasicExample()
15      {
16          $this->visit('/')
17                  ->see('Laravel 5');
18      }
19  }
```

This slick testing API is available as of Laravel 5.1. As you can see, the syntax is very readable and understandable. This example test accesses the project home page and confirms that the text `Laravel 5` is found somewhere on the page. To run the test, just execute the `phpunit` command:

---

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

```
1  $ vendor/bin/phpunit
2  PHPUnit 4.7.2 by Sebastian Bergmann and contributors.
3
4  .
5
6  Time: 615 ms, Memory: 12.00Mb
7
8  OK (1 test, 2 assertions)
```

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 lines:

```
1  $router::get('/', function() {
2    return view('welcome');
3  });
```

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:

```
1  $ vendor/bin/phpunit
2  PHPUnit 4.7.2 by Sebastian Bergmann and contributors.
3
4  F
5
6  Time: 403 ms, Memory: 11.75Mb
7
8  There was 1 failure:
9
10 1) ExampleTest::testBasicExample
11 A request to [http://localhost] failed. Received status code [404].
```

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 Laravel[70] documentation to learn more about the syntax available to you. In any case, be sure to uncomment that route definition before moving on!

---

[70]http://laravel.com/docs/master/testing

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!

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:

```
 1  <!doctype html>
 2  <html lang="en">
 3    <head>
 4      <meta charset="UTF-8">
 5      <title>Welcome to TODOParrot</title>
 6    </head>
 7    <body>
 8    <h1>Welcome to TODOParrot</h1>
 9    </body>
10  </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 the header "Welcome to TODOParrot". 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 a simple *route definition* found in the `app/Http/routes.php` file:

```
1  Route::get('/', function () {
2      return view('welcome');
3  });
```

This route tells Laravel to serve the `welcome.blade.php` file when a GET request is made to the application's homepage, represented by the forward slash (`/`). Although the majority of your views will be served in conjunction with a controller (more about this in a bit), if your application contains a bit of static content (such as an "About Us" page) then the above approach is a convenient solution for directly serving the view.

Because it's understood that views use the `.php` extension (you'll find more about the meaning of `blade` later in this chapter), 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:

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

Incidentally, for organizational purposes you can manage views in separate directories. To do so you 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 `home` in `resources/views`, and move the `welcome.blade.php` view into the newly created directory. Then update the route to look like this:

```
1  Route::get('/', function () {
2      return view('home.welcome');
3  });
```

You're certainly not required to manage views in this fashion, however I find the approach indispensable given that a typical Laravel application can quickly grow to include dozens of views.

Creating Your First Controller

The lone default route serves the important purpose of giving you *something* to see when accessing the home page of a newly generated Laravel application, however in practice you'll rarely serve views directly through the `routes.php` file. This is because the majority of your views will contain some degree of dynamic data, and this dynamic data is typically retrieved and passed into a view by way of a *controller*.

Although we're not yet ready to begin passing dynamic data into a view (I'll introduce this topic later in the chapter), it seems a fine time to learn how to create a controller capable of serving the welcome view. You can easily generate controllers using Artisan's `make:controller` command:

```
1  $ php artisan make:controller --plain WelcomeController
2  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. Doing so will create the following empty controller class, placing the contents inside `app/Http/Controllers/WelcomeController.php`:

```php
1  <?php
2
3  namespace todoparrot\Http\Controllers;
4
5  use Illuminate\Http\Request;
6
7  use todoparrot\Http\Requests;
8  use todoparrot\Http\Controllers\Controller;
9
10 class WelcomeController extends Controller
11 {
12     //
13 }
```

With the controller generated, let's create the `index` action and corresponding view. Open the newly created controller (`app/Http/Controllers/WelcomeController.php`) and add the following `index` method to the class:

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

> Controller class methods intended to respond to an application endpoint request are generally referred to as *actions*.

In this example I'm presuming you followed along with the earlier discussion pertaining to organizing views within subdirectories. If not, just replace `home.welcome` with `welcome`. In either case, after saving these changes open the `routes.php` file (`app/Http/routes.php`). Replace the lone defined route with the following route:

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

This route tells Laravel to respond with GET requests to the application home page (/) by executing the `WelcomeController`'s `index` action. Save these changes, return to the browser, and reload the home page. You should see the same view as earlier, but this time it's being served from a controller!

Managing Your Application Routes

As you learned earlier in this chapter, the `app/Http/routes.php` file defines your application's URL endpoints so Laravel knows how to respond to a particular request. To illustrate this capability we had a look at the default home page route, and subsequently update that route to instead serve the `welcome.blade.php` view using a controller. But this really only scratches the surface in terms of the many different ways in which you can define and manage routes. In this section I'll touch upon several other key routing capabilities.

Defining Resource (RESTful) Controller Routes

These days it's commonplace to dedicate each application controller to managing an associated resource. For instance a controller named `ListsController` might be responsible for retrieving all lists, retrieving a list detail view, inserting a new list, modifying an existing list, and deleting a list. These types of controllers are often created in such a way so as to conform to REST[71] conventions. Laravel supports RESTful controllers (often referred to as *resource controllers* within the Laravel community), and although I'll formally introduce the concept in Chapter 3 I nonetheless thought this a suitable spot to at least show you how to define a RESTful controller's routes within the `routes.php` file. For instance, to define the seven routes associated with a RESTful controller, all you need to do is identify the controller in your routes file like so:

---

[71]http://en.wikipedia.org/wiki/Representational_state_transfer

```
1  Route::resource('lists', 'ListsController');
```

Doing so will automatically make the following routes available to your application:

HTTP Method	Path	Controller	Description
GET	/lists	lists#index	Display all TODO lists
GET	/lists/new	lists#create	Display an HTML form for creating a new TODO list
POST	/lists	lists#store	Create a new TODO list
GET	/lists/:id	lists#show	Display a specific TODO list
GET	/lists/:id/edit	lists#edit	Display an HTML form for editing an existing TODO list
PUT	/lists/:id	lists#update	Update an existing TODO list
DELETE	/lists/:id	lists#destroy	Delete an existing TODO list

Obviously you'll need to additionally define the seven controller actions identified in this table (`index`, `create`, `store`, `show`, `edit`, `update`, and `destroy`)! I'll talk at length about this topic in the next chapter.

Defining Implicit Routes

In addition to defining routes using methods such as `Route::get`, `Route::post`, and `Route::Resource`, you can alternatively identify the route request methods associated with each controller action directly within the action name itself. For instance, suppose a particular controller isn't intended to be RESTful, and instead contains just a handful of GET- and POST-oriented actions. You can identify the controller in `routes.php` using the `Route::controllers` method:

```
1  Route::controllers([
2    'lists' => 'ListsController'
3  ]);
```

The `ListsController.php` file would then contain one or more actions, with the request method prefixed to the action name:

```php
1   <?php
2
3   namespace todoparrot\Http\Controllers;
4
5   use Illuminate\Http\Request;
6
7   use App\Http\Requests;
8   use App\Http\Controllers\Controller;
9
10  class ListsController extends Controller {
11
12    public function getIndex()
13    {
14      return view('lists.index');
15    }
16
17    public function getCreate()
18    {
19      return view('lists.create');
20    }
21
22    public function postStore()
23    {
24      return view('lists.store')
25    }
26
27  }
```

Notice how I've prefixed each action with `get` or `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 `create.blade.php`) you should be able to navigate to `/lists/create` and see the view contents. If you were to create a form and *post* the form contents to `/tasks/store` you should see the contents of the `resources/view/lists/store.blade.php` view.

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 Route Parameters

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

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:

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

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

```
1  public function category($category)
2  {
3    return view('blog.category')->with('category', $category);
4  }
```

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:

```
1  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:

```
1  public function category($category, $subcategory)
2  {
3    return view('blog.category')
4      ->with('category', $category)
5      ->with('subcategory', $subcategory);
6  }
```

Keep in mind these parameters are *required*. Neglecting to include them in the URL will result in an error. Sometimes however you might prefer to define these parameters as *optional*, and display a default view should the optional parameter(s) not be provided. To define an optional parameter you'll append a question mark onto the parameter name, like this:

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

You can then identify the parameter is optional in the associated action:

```
1  public function category($category = '')
2  {
3    $category = $category == '' ? 'php' : $category;
4    return view('blog.category')->with('category', $category);
5  }
```

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`:

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

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

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

```
1  $ php artisan route:list
2  +-----+--------------------...+------------------------------+-----------+
3  | ... | URI                ...| Action                       | Middleware |
4  +-----+--------------------...+------------------------------+-----------+
5  |     | GET|HEAD /         ...| App\..\HomeController@index   |           |
6  |     | POST auth/register ...| App\..\Auth\Auth...@register  | guest     |
7  |     | POST auth/login    ...| App\..\Auth\Auth...@login     | guest     |
8  |     | ...                ...| ...                           | guest     |
9  +-----+--------------------...+------------------------------+-----------+
```

Incidentally, this command will fail if you haven't yet configured your database, a task which isn't completed yet if you're not using Homestead. Therefore if you see an `Access denied` message in conjunction with executing this command, don't worry too much about it as it will resolve itself after you've completed the next chapter.

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

Caching Routes

Laravel 5 introduces route caching, an optimization that serializes your route definitions and places the results in a single file (`bootstrap/cache/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:

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

If you subsequently add, edit or delete a route definition you'll need to clear and rebuild the cache. You can do so by running the `route:cache` command again. To clear (without rebuilding) the route cache, execute the `route:clear` command:

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

This command will delete the cached routes file, causing Laravel to return to parsing the route definitions using `app/Http/routes.php` 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:

```
1  /**
2   * @Get("/")
3   */
4  public function index()
5  {
6    return view('welcome');
7  }
```

This feature generated 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`.

---

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

Defining URL Parameters Using Annotations

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

```
1    /**
2     * @Get("/blog/category/{category}")
3     */
```

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:

```
1    /**
2     * @Get("/blog/category/{category}", as="blog.category")
3     */
```

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

While route annotations are an interesting concept, 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.

Introducing the Blade Template Engine

One of the primary goals of an MVC framework such as Laravel 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 lack 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. Most frameworks attempt to achieve a happy medium by providing a simplified syntax 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. In this section we'll work through a number of different examples involving Blade syntax and the `welcome.blade.php` view.

Displaying Variables

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

```
1  public function index()
2  {
3      return view('welcome')->with('name', 'San Juan Vacation');
4  }
```

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

```
1  {{-- Output the $name variable. --}}
2  <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:

```
1  public function index()
2  {
3      $name = 'San Juan Vacation';
4      return view('welcome')->withName($name);
5  }
```

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

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

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:

```
1  public function index()
2  {
3
4      $data = array('name' => 'San Juan',
5                    'date' => date('Y-m-d'));
6
7      return view('welcome')->with($data);
8
9  }
```

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

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

You could also use multiple with methods, like so:

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

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

```
1  $name = 'San Juan Vacation';
2  $date = date('Y-m-d');
3  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:

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

---

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

Escaping Dangerous Input

Because web applications commonly display user-contributed 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:

```
1  {{ '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:

```
1  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:

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

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

```
1  {!! '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.

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:

```
1  public function index()
2  {
3    $lists = array('Vacation Planning', 'Grocery Shopping', 'Camping Trip');
4    return view('welcome')->with('lists', $lists);
5  }
```

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

```
1  <ul>
2    @foreach ($lists as $list)
3      <li>{{ $list }}</li>
4    @endforeach
5  </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:

```
1  <ul>
2    @forelse ($lists as $list)
3      <li>{{ $list }}</li>
4    @empty
5      <li>You don't have any lists saved.</li>
6    @endforelse
7  </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:

```
1   <ul>
2     @if (count($lists) > 0)
3       @foreach ($lists as $list)
4         <li>{{ $list }}</li>
5       @endforeach
6     @else
7       <li>You don't have any lists saved.</li>
8     @endif
9   </ul>
```

Blade also supports the if-elseif-else construct:

```
1   @if (count($lists) > 1)
2     <ul>
3       @foreach ($lists as $list)
4         <li>{{ $list }}</li>
5       @endforeach
6     </ul>
7   @elseif (count($lists) == 1)
8     <p>
9       You have one list: {{ $lists[0] }}.
10    </p>
11  @else
12    <p>You don't have any lists saved.</p>
13  @endif
14  </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:

```
1   <!doctype html>
2   <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Welcome to TODOParrot</title>
6   </head>
7   <body>
8
9     @yield('content')
10
11  </body>
12  </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 `welcome.blade.php` and modify its contents to look like this:

```
1   @extends('layouts.master')
2
3   @section('content')
4
5   <h1>Welcome to TODOParrot</h1>
6
7   <p>
8     TODOParrot is the ultimate productivity application for
9     tropically-minded users.
10  </p>
11
12  @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 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>:

```
1   <body>
2
3     <div class="container">
4
5       <div class="col-md-9">
6         @yield('content')
7       </div>
8
9       <div class="col-md-3">
10        @section('advertisement')
11        <p>
12          Jamz and Sun Lotion Special $29!
13        </p>
14        @show
15      </div>
16    </div>
17  </body>
```

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

```
1   @endsection
2   @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:

```
1   @extends('layouts.master')
2
3   @section('content')
4     <h1>Welcome to TODOParrot!</h1>
5   @endsection
6
7   @section('advertisement')
8     @parent
9       <p>
```

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

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

```
1  <p>
2    Jamz and Sun Lotion Special $29!
3  </p>
4  <p>
5    Buy the TODOParrot Productivity guide for $10!
6  </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:

```
1  <tr style="padding-bottom: 5px;">
2    <td>
3    {{ $link->name }}
4    </td>
5  </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 like so:

```
1  <table class="table borderless">
2  @foreach ($links as $link)
3
4    @include('partials.row', array('link' => $link))
5
6  @endforeach
7  </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 place these files, you're free to reference them using standard HTML or can optionally take advantage of a few helpers available via the Laravel HTML package[74]. For instance, the following two statements are identical:

```
1  <img src="/images/logo.png" alt="TODOParrot logo" />
2
3  {!! 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:

```
1  <link rel="stylesheet" href="/css/app.min.css">
2  <script src="/javascript/jquery-1.10.1.min.js"></script>
3  <script src="/javascript/bootstrap.min.js"></script>
4
5  {!! HTML::style('css/app.min.css') !!}
6  {!! HTML::script('javascript/jquery-1.10.1.min.js') !!}
7  {!! 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

```
1  "require": {
2    ...
3    "laravelcollective/html": "~5.1.*"
4  },
```

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:

```
1  Collective\Html\HtmlServiceProvider::class,
```

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

```
1  'HTML' => Collective\Html\HtmlFacade::class
```

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.

> For reasons unbeknownst to the author, the Bootstrap CSS source files have once again been removed from new Laravel projects as of version 5.1. However, because this occurred with an earlier version 5.0.X release, only to see the files later again return, I am for the time being going to leave this section untouched and presume they will very shortly again be included. If it turns out they are indeed going to be removed permanently, I'll adjust the following paragraph accordingly.

The Bootstrap CSS source files (in Less format) were initially included with every new Laravel 5 project, later removed in a subsequent release, and returned in an even later release. I'll presume you're using the very latest Laravel release and therefore already have the Bootstrap files available in your project. If not, just head over to http://getbootstrap.com/[79] and download the *source files*, placing the unzipped contents into `resources/assets/less/bootstrap`. After doing so, place the following line at the top of the `app.less` file (found in `resources/assets/less`):

---

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

[76]http://getbootstrap.com/

[77]http://bootswatch.com/

[78]http://bootswatch.com/flatly/

[79]http://getbootstrap.com/

```
1  @import "bootstrap/bootstrap";
```

If your project already includes the files (located inside `resources/assets/less/bootstrap`), then the above line will already exist in your `app.less` file.

This `import` statement will automatically compile all of the Bootstrap files along with any Less statements found in `app.less`, meaning if you include the compiled `app.css` in your `master.blade.php` header, all of Bootstrap's files will additionally be made available (see the later section "Introducing Elixir" for more information about Less compilation).

> Laravel does not include Bootstrap's JavaScript files, so you'll need to download and integrate those separately if you wish to take advantage of Bootstrap's JavaScript plugins.

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:

```
1   <head>
2     ...
3     <link rel="stylesheet"
4       href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css">
5     <script
6       src="http://code.jquery.com/jquery-1.11.3.min.js"></script>
7     <script
8       src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.min.js">
9     </script>
10  </head>
```

Regardless of the approach you choose, once added you're free to begin taking advantage of Bootstrap's various CSS widgets and JavaScript plugins. For instance try adding a stylized hyperlink to the `welcome.blade.php` view just to confirm everything is working as expected:

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

Integrating the Bootstrapper Package

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:

---

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

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

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

```
1  "require": {
2    ...
3    "patricktalmadge/bootstrapper": "~5"
4  },
```

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:

```
1  'providers' => array(
2    ...
3    'Bootstrapper\BootstrapperL5ServiceProvider'
4  ),
```

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 `app/config/app.php`. Paste the following rather lengthy bit of text into the bottom of the array:

```
1   'Accordion' => 'Bootstrapper\Facades\Accordion',
2   'Alert' => 'Bootstrapper\Facades\Alert',
3   'Badge' => 'Bootstrapper\Facades\Badge',
4   'Breadcrumb' => 'Bootstrapper\Facades\Breadcrumb',
5   'Button' => 'Bootstrapper\Facades\Button',
6   'ButtonGroup' => 'Bootstrapper\Facades\ButtonGroup',
7   'Carousel' => 'Bootstrapper\Facades\Carousel',
8   'ControlGroup' => 'Bootstrapper\Facades\ControlGroup',
9   'DropdownButton' => 'Bootstrapper\Facades\DropdownButton',
10  'Form' => 'Bootstrapper\Facades\Form',
11  'Helpers' => 'Bootstrapper\Facades\Helpers',
12  'Icon' => 'Bootstrapper\Facades\Icon',
13  'InputGroup' => 'Bootstrapper\Facades\InputGroup',
14  'Image' => 'Bootstrapper\Facades\Image',
15  'Label' => 'Bootstrapper\Facades\Label',
16  'MediaObject' => 'Bootstrapper\Facades\MediaObject',
17  'Modal' => 'Bootstrapper\Facades\Modal',
18  'Navbar' => 'Bootstrapper\Facades\Navbar',
```

```
19  'Navigation' => 'Bootstrapper\Facades\Navigation',
20  'Panel' => 'Bootstrapper\Facades\Panel',
21  'ProgressBar' => 'Bootstrapper\Facades\ProgressBar',
22  'Tabbable' => 'Bootstrapper\Facades\Tabbable',
23  'Table' => 'Bootstrapper\Facades\Table',
24  'Thumbnail' => 'Bootstrapper\Facades\Thumbnail',
```

Adding these aliases will save you the hassle of having to include 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>`:

```
1  {!! HTML::style('path/to/bootstrap.css') !!}
2  {!! HTML::script('http://code.jquery.com/jquery-1.11.3.min.js') !!}
3  {!! HTML::script('path/to/bootstrap.js') !!}
```

Notice I'm using the jQuery CDN in the above example. jQuery is required to take advantage of Bootstrap's JavaScript components. You're not required to use them, and in fact if you don't plan on using them I suggest removing the last two lines in the above example. Also, I'm using the `HTML` package's `script` helper to reference the CSS and JS files. You can alternatively use the HTML `link` and `script` elements to accomplish the same goal. In either case, after saving the changes you're ready to begin taking advantage of Bootstrapper's CSS styling and (optionally) 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, hide debugging statements from the production environment, 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 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

---

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

[83]http://lesscss.org/

[84]http://sass-lang.com/

[85]http://coffeescript.org/

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.

ⓘ Laravel 5.1 Update Alert

As of Laravel 5.1 users have the luxury of writing ECMAScript 6 and taking advantage of Browserify when integrating JavaScript into a Laravel application!

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

Gulp is built atop Node.js[88], meaning you'll need to install Node.js. 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. Mac users 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  v0.12.2
```

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:

---

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

```
1  $ gulp -v
2  [14:12:51] CLI version 3.9.0
```

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:

```
1  {
2    "devDependencies": {
3      "gulp": "^3.8.8",
4      "laravel-elixir": "*"
5    }
6  }
```

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:

```
1  $ 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. This directory is used by Node.js to house the various packages installed per the `package.json` specifications, so you definitely do not want to delete nor modify it.

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:

```
1  elixir(function(mix) {
2      mix.less('app.less');
3  });
```

The `mix.less` task compiles a Less[90] file named `app.less` which resides in `resources/assets/less`. In Laravel 5.0.X this file imported Bootstrap and used Less syntax to override a few Bootstrap styles:

---

[90]http://lesscss.org/

```
1  @import "bootstrap/bootstrap";
2
3  @btn-font-weight: 300;
4  @font-family-sans-serif: "Roboto", Helvetica, Arial, sans-serif;
5
6  body, label, .checkbox label {
7     font-weight: 300;
8  }
```

As of Laravel 5.1 the `app.less` file has been wiped clean, so I suggest adding the following content back into the file in order to follow along. Keep in mind you'll need to download the Bootstrap Less source files per the earlier instructions in order for the `@import` statement and variable overrides to work. After adding a few styles , you can execute the Elixir task by running `gulp` within your project root directory:

```
1  $ gulp
2  [14:44:43] Using gulpfile ~/Software/dev.todoparrot.com/gulpfile.js
3  [14:44:43] Starting 'default'...
4  [14:44:43] Starting 'less'...
5  [14:44:43] Running Less: resources/assets/less/app.less
6  [14:44:44] Finished 'default' after 753 ms
7  [14:44:44] gulp-notify: [Laravel Elixir] Less Compiled!
8  [14:44:44] Finished 'less' after 875 ms
```

Once executed, you'll find a compiled CSS file named `app.css` inside 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:

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

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:

---

[91]http://coffeescript.org/

```
1  alert "Hi I am annoying"
```

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

```
1  elixir(function(mix) {
2      mix.less('app.less');
3      mix.coffee();
4  });
```

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

```
1  elixir(function(mix) {
2      mix.less('app.less').coffee();
3  });
```

Save the changes and run gulp again:

```
1   $ gulp
2   [14:47:06] Using gulpfile ~/Software/dev.todoparrot.com/gulpfile.js
3   [14:47:06] Starting 'default'...
4   [14:47:06] Starting 'less'...
5   [14:47:06] Running Less: resources/assets/less/app.less
6   [14:47:07] Finished 'default' after 743 ms
7   [14:47:07] gulp-notify: [Laravel Elixir] Less Compiled!
8   [14:47:07] Finished 'less' after 857 ms
9   [14:47:07] Starting 'coffee'...
10  [14:47:07] Running CoffeeScript: resources/assets/coffee//**/*.coffee
11  [14:47:07] gulp-notify: [Laravel Elixir] CoffeeScript Compiled!
12  [14:47:07] Finished 'coffee' after 190 ms
```

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

```
1  (function() {
2    alert("Hi I am annoying");
3
4  }).call(this);
```

Like the compiled CSS, you'll need to properly reference the app.js file within your project layout or appropriate view.

Watching for Changes

Because you'll presumably be making regular tweaks to your CSS and JavaScript, consider using Elixir's watch command to automatically execute gulpfile.js anytime your assets change:

```
1  $ gulp watch
2  [14:49:29] Starting 'watch'...
3  [14:49:29] Starting 'less'...
4  [14:49:29] Running Less: resources/assets/less/app.less
5  [14:49:30] Finished 'watch' after 718 ms
6  [14:49:30] gulp-notify: [Laravel Elixir] Less Compiled!
7  [14:49:30] Finished 'less' after 851 ms
8  [14:49:30] Starting 'coffee'...
9  [14:49:30] Running CoffeeScript: resources/assets/coffee//**/*.coffee
10 [14:49:30] gulp-notify: [Laravel Elixir] CoffeeScript Compiled!
11 [14:49:30] Finished 'coffee' after 178 ms
12 [14:49:30] Starting 'watch-assets'...
```

This process will continue to run, so you'll want to execute it in a dedicated tab or in the background. Once running, each time the target files associated with the Elixir tasks are changed, Gulp will automatically run the tasks and update the compiled files accordingly.

> 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.

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:

---

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

```
1   "require-dev": {
2     ...
3     "fabpot/goutte": "2.*"
4   },
```

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:

```
1   <?php
2
3   use Goutte\Client;
4
5   class WelcomeTest extends TestCase {
6
7   }
```

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:

```
1    public function testUserSeesWelcomeMessage()
2    {
3
4      $client = new Client();
5      $crawler = $client->request('GET', 'http://localhost:8000/');
6
7      $this->assertEquals(200, $client->getResponse()->getStatus());
8
9      $this->assertCount(1,
10        $crawler->filter('h1:contains("Welcome to TODOParrot")'));
11
12    }
```

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, suppose you wanted to confirm a link to the Contact Us page is included in a given page. The link looks like this:

```
1  <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:

```
1  public function testUserClicksContactLinkAndIsTakenToContactPage()
2  {
3
4    $client = new Client();
5    $crawler = $client->request('GET', 'http://localhost:8000/');
6
7    $link = $crawler->selectLink('Contact Us')->link();
8
9    $this->assertEquals('http://localhost:8000/about/contact',
10     $link->getUri());
11
12   $crawler = $client->click($link);
13
14   $this->assertCount(1,
15     $crawler->filter('h1:contains("Contact Us")'));
16
17 }
```

Incidentally, as of Laravel 5.1 you can use the new HTTP request API and other interactive testing capabilities to implement a similar test to the above using more user-friendly syntax:

```
1  public function testUserClicksContactLinkAndIsTakenToContactPage()
2  {
3
4    $this->visit('/')
5          ->click('Contact Us')
6          ->seePageIs('/about/contact')
7          ->see('<h1>Contact Us</h1>');
8
9  }
```

In forthcoming chapters we'll continue to expand upon test-related matters, 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!

Chapter 3. Introducing Laravel Models

A well-designed web application will be *model-centric*, meaning the models that form the crux of the application (for instance, users, lists and tasks) will be the primary driver for implementing an application's business logic. Fortunately, Laravel provides developers with a powerful set of tools useful for building and managing models and their respective underlying database tables.

This chapter is the first of two devoted entirely to the data-related aspects of your Laravel application. This chapter focuses largely on model fundamentals. You'll learn how to generate models, and extend their capabilities with accessors, mutators and custom methods. You'll also learn how to use *migrations* to manage your project's underlying database schema, and how to *seed* your database with useful test and helper data. We'll spend the majority of the remaining chapter reviewing dozens of examples demonstrating how Laravel's *Eloquent* ORM can be used to retrieve, insert, update and delete data, before wrapping up with a brief look at Laravel's *Query Builder* interface (including a demonstration of how to execute raw SQL) and a section on testing your models.

What you learn here sets the stage for the advanced model-related material found in the next chapter, so be sure you reasonably understand the concepts presented here before moving on!

Configuring Your Project Database

In Chapter 1 I briefly touched upon Laravel's database support. To quickly recap, Laravel currently supports four databases, including MySQL, PostgreSQL, SQLite, and Microsoft SQL Server. The `config/database.php` file tells Laravel which database you'd like your project to use. In this file you'll also define the database connection credentials along with a few other settings. I've pasted in the contents of `config/database.php` below (with comments removed for reasons of space). Following the snippet I'll summarize the purpose of each setting:

```php
1   <?php
2
3   return [
4
5       'fetch' => PDO::FETCH_CLASS,
6
7       'default' => 'mysql',
8
9       'connections' => [
10
11          'sqlite' => [
12              'driver'   => 'sqlite',
13              'database' => storage_path().'/database.sqlite',
14              'prefix'   => '',
15          ],
16
17          'mysql' => [
18              'driver'    => 'mysql',
19              'host'      => env('DB_HOST', 'localhost'),
20              'database'  => env('DB_DATABASE', 'forge'),
21              'username'  => env('DB_USERNAME', 'forge'),
22              'password'  => env('DB_PASSWORD', ''),
23              'charset'   => 'utf8',
24              'collation' => 'utf8_unicode_ci',
25              'prefix'    => '',
26              'strict'    => false,
27          ],
28
29          'pgsql' => [
30              'driver'   => 'pgsql',
31              'host'     => env('DB_HOST', 'localhost'),
32              'database' => env('DB_DATABASE', 'forge'),
33              'username' => env('DB_USERNAME', 'forge'),
34              'password' => env('DB_PASSWORD', ''),
35              'charset'  => 'utf8',
36              'prefix'   => '',
37              'schema'   => 'public',
38          ],
39
40          'sqlsrv' => [
41              'driver'   => 'sqlsrv',
42              'host'     => env('DB_HOST', 'localhost'),
```

```
43        'database' => env('DB_DATABASE', 'forge'),
44        'username' => env('DB_USERNAME', 'forge'),
45        'password' => env('DB_PASSWORD', ''),
46        'prefix'   => '',
47      ],
48
49    ],
50
51    'migrations' => 'migrations',
52
53    'redis' => [
54
55      'cluster' => false,
56
57      'default' => [
58        'host'     => '127.0.0.1',
59        'port'     => 6379,
60        'database' => 0,
61      ],
62
63    ],
64
65  ];
```

Let's review each setting:

- `fetch`: The Eloquent ORM will by default return database results as instances of PHP's `stdClass` object. You could optionally instead return results in array format by changing this setting to `PDO:FETCH_ASSOC`. If you're not particularly familiar with object orientation then this alternative might seem attractive, however I suggest leaving the default in place unless special circumstances warrant the change.
- `default`: The `default` setting identifies the type of database used by your project. You can set this to `mysql` (the default), `pgsql` (PostgreSQL), `sqlite`, or `sqlsrv` (Microsoft SQL Server). Keep in mind none of these databases come packaged with Laravel. You'll need to separately install and configure the database, or obtain credentials to access the desired database.
- `connections`: This array defines the database authentication credentials for each supported database. Of course, Laravel will only consider the setting associated with the database defined by the `default` setting, therefore you can leave the unused database options untouched (or entirely remove them). The `env()` function used to retrieve various connection parameters will look to the `.env` file (introduced in Chapter 1) and retrieve a configuration variable equal to the name of the function's first parameter (for instance, `DB_HOST`). The `env()` function's second parameter identifies a default value should the desired configuration variable not be

found. If you plan on using Homestead's MySQL database you'll need to change the `.env` file's `DB_HOST` variable to `localhost:33060`.

- `migrations`: This setting defines the name of the table used for managing your project's migration status. This is by default set to `migrations`, however if by the wildest of circumstances you needed to change the table name to something else, you can do so here.

- `redis`: You can optionally use Redis[96] to manage cache and session data (among other things). If you'd like to use Redis in your Laravel application, you'll define the connection information using this setting.

After identifying the desired database and defining the authorization credentials, don't forget to create the database because Laravel will not do it for you. If you're using Homestead then a database named `homestead` has already been created for you, negating the need to go through these additional steps. Of course, if you're working on multiple Laravel 5 projects then regardless you'll need to create separate databases. With the database defined, it's time to begin interacting with it!

Introducing the Eloquent ORM

Object-relational mapping (ORM) is without question the feature that led me to embrace web frameworks several years ago. Even if you've never heard of an ORM, anybody who has created a database-backed web site has undoubtedly encountered the problem this programming technique so gracefully resolves: *impedence mismatch.* Borrowed from the electrical engineering field, impedence mismatch[97] is the term used to describe the challenges associated with using an object-oriented language such as PHP or Ruby in conjunction with a relational database, because the programmer is faced with the task of somehow mapping the application objects to database tables. An ORM solves this problem by providing a convenient interface for converting application objects to database table records, and vice versa. Additionally, most ORMs offer a vast array of convenient features useful for querying, inserting, updating, and deleting records, managing table relationships, and dealing with other aspects of the data life cycle.

Creating Your First Model

Creating a model using Artisan is easy. Let's kick things off by creating the `Todolist` model, which the TODOParrot application uses to manage the user's various lists. Beginning with Laravel 5 you can now generate models using Artisan:

---

[96]http://redis.io/
[97]http://en.wikipedia.org/wiki/Impedance_matching

```
1  $ php artisan make:model Todolist -m
2  Model created successfully.
3  Created Migration: 2015_01_14_215705_create_todolists_table
```

Notice I also supplied the -m option to the make:model command. This tells Laravel to additionally create a companion migration. This feature was enabled by default at some point during the 5.0.X release series however was suddenly disabled with the 5.1 release. At any rate, supplying the -m option resolves the issue. If you don't know what a migration is, we'll talk about them in just a moment.

You'll find the new model in app/Todolist.php. It looks like this:

```php
1  <?php namespace todoparrot;
2
3      use Illuminate\Database\Eloquent\Model;
4
5      class Todolist extends Model {
6
7        //
8
9      }
```

> Like most web frameworks, Laravel expects the model name to be singular form (Todolist), and the underlying table names to be plural form (todolists).

As you can see, a Laravel model is just a PHP class that extends Laravel's Model class, thereby endowing the class with the features necessary to act as the bridge between your Laravel application and the underlying database table. Currently the class is empty however we'll begin expanding its contents soon enough.

A model is only useful when it's associated with an underlying database table. You might have noticed from the above output that when you created the model Laravel also created something called a *migration*. This file contains the blueprint for creating the the model's associated table. Let's talk about the power of migrations next.

Introducing Migrations

With the model created, you'll typically create the corresponding database table, done through a fantastic Laravel feature known as *migrations*. Migrations offer a file-based approach to changing the structure of your database, allowing you to create and drop tables, add, update and delete columns, and add indexes, among other tasks. Further, you can easily revert, or *roll back*, any changes

if a mistake has been made or you otherwise reconsider the decision. Finally, because each migration is stored in a text file, you can manage them within your project repository.

To demonstrate the power of migrations, let's have a look at the migration file that was created along with the `Todolist` model (presuming you supplied the `-m` option). This migration file is named `2015_01_14_215705_create_todolists_table.php`, and it was placed in the `database/migrations` directory. Open up this file and you'll see the following contents:

```php
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateTodolistsTable extends Migration {

  /**
   * Run the migrations.
   *
   * @return void
   */
  public function up()
  {
    Schema::create('todolists', function(Blueprint $table)
    {
      $table->increments('id');
      $table->timestamps();
    });
  }

  /**
   * Reverse the migrations.
   *
   * @return void
   */
  public function down()
  {
    Schema::drop('todolists');
  }

}
```

Like a model, a Laravel migration is just a standard PHP class, except in this case the class extends the `Migration` class. Take note of the two class methods, `up()` and `down()`. These have

special significance in regards to migrations, with the up() method defining what occurs when the migration is executed, and down() defining what occurs when the migration is reverted. Therefore the down() method should define what happens when you'd like to *undo* the changes occurring as a result of executing the up() method. Let's first discuss this example's up() method:

```
1  public function up()
2  {
3    Schema::create('todolists', function(Blueprint $table)
4    {
5      $table->increments('id');
6      $table->timestamps();
7    });
8  }
```

You'll regularly see the Schema class appear in migration files, because it is Laravel's solution for manipulating database tables in all manner of fashions. This example uses the Schema::create method to create a table named todolists. An anonymous function (closure) passed along as the Schema::create method's second parameter defines the table columns:

- $table->increments('id'): The increments method indicates we want to create an automatically incrementing integer column that will additionally serve as the table's primary key.
- $table->timestamps(): The timestamps method informs Laravel to include created_at and updated_at timestamp columns, which will be automatically updated to reflect the current timestamp when the record is created and updated, respectively.

There are plenty of other methods useful for creating different column data types, setting data type attributes, and more. Be sure to check out the Schema[98] documentation for a complete list, otherwise stay tuned as we'll cover various other methods in the examples to come.

We want each todolists record to manage more than just a primary key and timestamps, of course. Notably each list should be assigned a name and description, so let's modify the Schema::create body to additionally add these fields:

---

[98]http://laravel.com/docs/schema

```
1    Schema::create('todolists', function(Blueprint $table)
2    {
3      $table->increments('id');
4      $table->string('name');
5      $table->text('description');
6      $table->timestamps();
7    });
```

If you're not familiar with these column types I'll describe them next:

- $table->string('name'): The string method indicates we want to create a variable character column (commonly referred to as a VARCHAR by databases such as MySQL and PostgreSQL) named name. Remember, the Schema class is database-agnostic, and therefore leaves it to whatever supported Laravel database you happen to be using to determine the maximum string length, unless you use other means to constrain the limit.
- $table->text('description'): The text method indicates we want to create a text column (commonly referred to as TEXT by databases such as MySQL and PostgreSQL).

The example's down() method is much easier to introduce because it consists of a single instruction: Schema::drop('todolists'). When run it will remove, or *drop* the todolists table.

Now that you understand what comprises this migration file, let's execute it and create the todolists table:

```
1    $ php artisan migrate
2    Migration table created successfully.
3    Migrated: 2015_01_14_215705_create_todolists_table
```

> ℹ️ If this is your very first migration, then Laravel will additionally create the password_-resets and users table. These tables are used to manage user-related data, and we'll talk about both in detail in Chapter 7.

Typically only the second output line will be displayed, however because this is the very first migration, Laravel also creates a table called migrations. This table is used by Laravel to keep track of the current migration version. This version number correlates with the name of the migration file. For instance after running the 2015_01_14_215705_create_todolists_table migration the migrations table looks like this:

```
1  mysql> select * from migrations;
2  +------------------------------------------+-------+
3  | migration                                | batch |
4  +------------------------------------------+-------+
5  | 2015_01_14_215705_create_todolists_table |     1 |
6  +------------------------------------------+-------+
```

Every time a migration is run, a record will be added to the `migrations` table identifying the migration file name, and the group, or *batch* in which the migration belongs. In other words, if you create for instance three migrations and then run `php artisan migrate`, those three migrations will be placed in the same batch. If you later wanted to undo any of the changes found in any of those migrations, those three migrations would be treated as a group and rolled back together.

> When using the mysql client to view tables consisting of a relatively large number of columns, ending the query with \G instead of a semicolon will result in the output being displayed in a much more readable vertical format.

Once complete, open your development database using your favorite database editor (I prefer to use the MySQL CLI), and confirm the `todolists` table has been created:

```
1  mysql> show tables;
2  +----------------------------+
3  | Tables_in_dev_todoparrot_com |
4  +----------------------------+
5  | todolists                  |
6  | migrations                 |
7  +----------------------------+
8  2 rows in set (0.03 sec)
```

Indeed it has! Let's next check out the `todolists` table schema:

```
1  mysql> describe todolists;
2  +-------------+------------------+---+----------------+
3  | Field       | Type             |...| Extra          |
4  +-------------+------------------+---+----------------+
5  | id          | int(10) unsigned |...| auto_increment |
6  | name        | varchar(255)     |...|                |
7  | description | text             |...|                |
8  | created_at  | timestamp        |...|                |
9  | updated_at  | timestamp        |...|                |
10 +-------------+------------------+---+----------------+
11 5 rows in set (0.00 sec)
```

Sure enough, an automatically incrementing integer column has been created, in addition to the `name`, `description`, `created_at` and `updated_at` columns.

Suppose you realize a mistake was made in the migration (perhaps you forgot to add a column or used an incorrect datatype). You can easily roll back the changes using the following command:

```
1  $ php artisan migrate:rollback
2  Rolled back: 2015_01_14_215705_create_todolists_table
```

After rolling back the changes check your database and you'll see both the `todolists` table has been removed and the relevant record in the `migrations` table. Of course, we'll actually want to use the `todolists` table, so run `php artisan migrate` anew before moving on.

Dealing with Unsupported Data Types

Laravel supports all of the most commonly used data types however support is not exhaustive. For instance if you wanted to define a column using `VARBINARY` or `MEDIUMBLOB` you will need to alter the table using a raw query following table creation:

```
1  public function up()
2  {
3
4      Schema::create('todolists', function(Blueprint $table)
5      {
6        ...
7        $table->string('name');
8        ...
9      });
10
11     \DB::statement('ALTER TABLE `todolists` MODIFY
12         `name` VARBINARY(100);');
13
14 }
```

In this example I first create the `name` column using a `VARCHAR`, and then subsequently alter the column. You could also just not bother with using a placeholder column and instead add the column using `ALTER TABLE`.

Defining Column Modifiers

In many cases it simply isn't enough to just define a table's column names and associated types. You'll additionally often want to further constrain the columns using modifiers. For instance, to set a column default you can use the `default` method:

```
1   $table->boolean('confirmed')->default(false);
```

To allow null values you can use the `nullable` method:

```
1   $table->string('comments')->nullable();
```

To set an integer column to unsigned, use the `unsigned` method:

```
1   $table->tinyInteger('age')->unsigned();
```

You can chain multiple modifiers to ensure even more sophisticated constraints:

```
1   $table->tinyInteger('age')->unsigned()->default(0);
```

You can review a complete list of supported column types and other options in the Laravel documentation[99].

Adding and Removing Columns

To add or remove a table column you'll generate a migration just as you did when creating a table in the previous section, the only difference being you'll use the `--table` option to identify the table you'd like to modify:

```
1   $ php artisan make:migration
2     add_note_to_tasks_table --table=todolists
3   Created Migration:
4     2015_02_18_152547_add_note_to_todolists_table
```

Next open up the newly created migration file, creating the desired column in the `up` method, and making sure you drop the column in the `down` method:

---

[99]http://laravel.com/docs/master/migrations

```
1  public function up()
2  {
3    Schema::table('todolists', function(Blueprint $table)
4    {
5        $table->string('note');
6    });
7  }
8
9  /**
10  * Reverse the migrations.
11  *
12  * @return void
13  */
14  public function down()
15  {
16    Schema::table('todolists', function(Blueprint $table)
17    {
18        $table->dropColumn('note');
19    });
20  }
```

Controlling Column Ordering

One of the beautiful aspects of an ORM such as Eloquent and migrations is the ability to essentially forget about matters such as column ordering. However as I tend to obsess over irrelevant minutiae, I often desire to manage the order of columns, typically placing the primary and foreign keys at the beginning of the table. If you later add a column and would like to insert it into a specific location within the table structure, use the after method:

```
1  $table->string('city', 100)->after('street');
```

Checking Migration Status

Sometimes you'll create several migrations and lose track of which ones were moved into the database. Of course, you could visually confirm the changes in the database, however Laravel offers a more convenient command-line solution. The following example uses the migrate:status command to review the status of the TODOParrot project at some point during development:

```
1  $ php artisan migrate:status
2  +------+-------------------------------------------+
3  | Ran? | Migration                                 |
4  +------+-------------------------------------------+
5  | ☐    | 2015_01_14_215705_create_todolists_table  |
6  | ☐    | 2014_01_15_155245_create_users_table      |
7  | ☐    | 2014_01_16_030306_add_user_id_to_todolists |
8  | ☐    | 2014_01_16_175008_create_tasks_table      |
9  +------+-------------------------------------------+
```

With our first model and corresponding table created, let's spend some time creating and manipulating a few lists.

Defining Accessors, Mutators, and Methods

It's important to remember Laravel models are just POPOs (Plain Old PHP Objects) that by way of extending the `Model` class have been endowed with some special additional capabilities. This means you're free to take advantage of PHP's object-oriented features to further enhance the model, including adding getters, setters, and methods.

Defining Accessors

You'll use an *accessor* (also known as a *getter*) when you'd like to encapsulate the retrieval of a model attribute. You'll define an accessor using the convention `getAttributeName`. Suppose some future version of TODOParrot allowed users to create accounts. It was determined users' usernames would only ever be referenced using lowercase characters, regardless of how the username was originally entered into the system. You could use an accessor to modify how the username is retrieved using a method named `getUsername`:

```
1  class User extends Model {
2
3    public function getUsername()
4    {
5      return strtolower($this->username);
6    }
7
8  }
```

Frankly I'm not a fan of accessors, as even if the lowercase username requirement were a strict business requirement it is still a presentational matter and therefore I'd argue the task should be left to your application's presentational logic. Much more useful in my opinion is the creation of virtual

accessors, used to combine multiple attributes together. For instance, suppose the hypothetical `User` model separates the user's name into `first_name` and `last_name` attributes. However you'd like the option of easily retrieving the user's full name, which logically always consists of the first name followed by the last name. You can define an accessor to easily retrieve this virtual attribute:

```php
class User extends Model {

  public function getFullnameAttribute()
  {
    return $this->first_name . " " . $this->last_name;
  }

}
```

Once saved, you can access the virtual `fullname` attribute as you would any other:

```php
use todoparrot\User;

...

$list = User::find(12);
echo $list->fullname;
```

Incidentally, if you're using the PsySH shell (via Tinker) to experiment with the models and Eloquent you'll need to first declare the namespace:

```
>>> namespace todoparrot;
=> null
>>> $list = User::find(12);
>>> echo $list->fullname;
```

Defining Mutators

You'll use a *mutator* (also known as a *setter*) when you'd like to modify the value of an attribute. Staying with the theoretical `User` model, users would logically sign in to TODOParrot using a username or e-mail address and password. For security reasons the password should be stored in the database using a hashed format, meaning it's theoretically impossible to recreate the original value even when the hashed value is known. You would want to be absolutely certain the password is only saved to the database using the chosen hashing algorithm, and therefore might consider creating a mutator for the `password` attribute. Laravel recognizes mutators when the method is defined using the `setAttributeNameAttribute` convention, meaning you'll want to create a method named `setPassword`:

```
1  class User extends Model {
2
3    public function setPasswordAttribute($password)
4    {
5        $this->attributes['password'] = \Hash::make($password);
6    }
7
8  }
```

This example uses Laravel's `Hash` class to generate a hash (learn more about this class here[100]), accepting the plaintext password passed into the method, generating the hash, and assigning the hash to the class instance's `password` attribute. Here's an example:

```
1  $user = new User;
2  $user->password = 'blah';
3  echo $user->password;
4  $2y$10$e3ufaNvBFWM/SeFc4ZyAhe8u5UR/K0ZUc5IjCPUvOYv6IVuk7Be7q
```

Defining Methods

Custom methods can greatly reduce the amount of logic cluttering your controllers, not to mention help you to stay DRY. For instance, suppose a future version of the `List` model includes an due date attribute which the user can use to define a date in which the list should be completed. You can define a convenience method to determine whether the list's due date has arrived:

```
1  use Carbon\Carbon;
2
3  ...
4
5  class Todolist extends Model {
6
7      public function isDueToday()
8      {
9        $now = \Carbon::now();
10       if ($this->created_at->diff($now)->days == 0) {
11         return true;
12       } else {
13         return false;
14       }
15     }
16
17 }
```

---

[100]http://laravel.com/docs/4.2/security#storing-passwords

This example uses the fantastic Carbon[101] library, a PHP 5.3+ DateTime extension. With the isDueToday method in place, you can easily determine whether a list's due date has arrived:

```
1   $list = Todolist::find(12);
2
3   if ($list->isDueToday()) {
4     echo "This list is due today!";
5   }
```

Validating Your Models

Readers familiar with frameworks such as Ruby on Rails are used to defining validation rules in the model and then using native methods such as valid? to determine whether a model object's attributes are set to expectations. Laravel supports a similar approach, although it does require developers to do a bit of additional work in order to achieve a desirable validation workflow. In this section I'll run you through a simple example in which we'll add validators to the List model, and then use the validator to ensure the supplied data conforms to the defined rules.

> If you are only planning on inserting, updating and deleting model-based data via a Web interface, then you don't have to fret so much about model-based validation and instead can take advantage of Laravel 5's new form requests feature, introduced in Chapter 5.

Open up the Todolist model (app/Todolist.php) and add the following property:

```
1   class Todolist extends Model {
2
3     private $rules = [
4       'name' => 'required',
5       'description' => 'required'
6     ];
7
8   }
```

The $rules array just serves as a convenient structure for defining the validations rules associated with each model attribute. This example depicts a pretty simple set of rules, stating only that the name and description attributes are required without imposing additional constraints such as for instance a minimum string length. If you do want to attach multiple validators to an attribute you'll separate each with the pipe character, like so:

---

[101]https://github.com/briannesbitt/Carbon

```
1    private $rules = [
2       'email' => 'required|email|unique:users'
3    ];
```

Incidentally, Laravel offers a wide variety of native rules useful for validating e-mail addresses, URLs, integers, string length, dates, and more. You can view a complete list of available validators here[102].

Next, you'll want to use those rules to ensure the supplied data is conformant before saving it to the database. You'll use Laravel's `Validator` class in conjunction with these rules to perform the validation and generate the error messages should validation fail. We can encapsulate the validation logic in a method:

```
1    class Todolist extends Model {
2
3      ...
4
5      public function validate()
6      {
7
8        $v = \Validator::make($this->attributes, $this->rules);
9        if ($v->passes()) return true;
10       $this->errors = $v->messages();
11       return false;
12
13     }
14
15   }
```

As an example, suppose you wanted to validate a user-supplied list before saving it to the database. The logic flow might look something like this:

---

[102]http://laravel.com/docs/master/validation

```
1  $data = [
2    'name' => 'San Juan Vacation',
3    'description' => 'Things to do before leaving for vacation'
4  ];
5
6  $list = new Todolist($data);
7
8  if ($list->validate()) {
9    $errors = $list->errors();
10 } else {
11   $list->save();
12 }
```

While the `create` method is convenient, Laravel requires you to take some additional safeguards to ensure that a malicious user doesn't inject an undesired attribute into an array that might for instance be passed from a form into the `create` method. You can use a protected property named `$fillable` to determine which model attributes can be set using mass-assignment (in the fashion demonstrated via the above example. Because the current `Todolist` model doesn't identify any "fillable" attributes, the above example will actually fail. In order for it to succeed, you need to set the `$fillable` property like so:

```
1  class Todolist extends Model {
2
3    protected $fillable = ['name', 'description'];
4
5  }
```

Alternatively, suppose your table is fairly large and you're fine with allowing mass assignment for all but a few select attributes. Rather than maintain an unwieldy list in `$fillable`, you can instead identify only those you do *not* want to be mass-assigned using the `$guarded` property:

```
1  class Todolist extends Model {
2
3    protected $guarded = ['some_important_attr'];
4
5  }
```

> **ℹ** I am purposefully not couching these examples in the context of a controller action because Laravel 5 offers a pretty slick forms processing and validation feature known as *form requests*. You'll probably want to use form requests when processing form data for use in conjunction with a model. See Chapter 5 for more information about this great feature.

While this approach isn't terrible, it's a bit tedious to integrate validation logic into every new model. Additionally, I'm just not interested in treating the validation and persistence process as two separate tasks, instead preferring to save the data and if it fails due to validation, just receive the validation errors in return. If you prefer a more succinct approach, several third-party packages can add convenient validation capabilities to your models. I'll introduce you to two of the most popular solutions next.

Creating a RESTful Controller

You'll interact with the `Todolist` model like you would any other PHP class, keeping in mind that a number of special methods and properties are additionally made available to the class because it extends Laravel's `Model` class. We'll logically want to interact with the model within the TODOParrot application in a variety of fashions. Notably we'll want to retrieve lists, learn more about a specific list, create a new list, update a list, and delete a list. These tasks are so central to web applications that most popular web frameworks, Laravel included, implement *representational state transfer* (REST), an approach to designing networked applications that codify the way in which these tasks (create, retrieve, update, and delete) are implemented. RESTful applications use the HTTP protocol and a series of well-defined URL endpoints to implement the seven actions defined in the following table (first presented in chapter 2 but recreated here for your convenience):

HTTP Method	Path	Controller	Description
GET	/lists	lists#index	Display all TODO lists
GET	/lists/new	lists#create	Display an HTML form for creating a new TODO list
POST	/lists	lists#store	Create a new TODO list
GET	/lists/:id	lists#show	Display a specific TODO list
GET	/lists/:id/edit	lists#edit	Display an HTML form for editing an existing TODO list
PUT	/lists/:id	lists#update	Update an existing TODO list
DELETE	/lists/:id	lists#destroy	Delete an existing TODO list

The `:id` included in several of the paths is a placeholder for a record's primary key. For instance, if you wanted to view the list identified by the primary key 427, then the URL path would look like `/lists/427`. At first glance, it might not be so obvious how some of the other paths behave; for instance REST newcomers are often confused by `POST /lists` or `PUT /lists/:id`. Not to worry! We'll sort all of this out in the sections to come.

As mentioned, Laravel natively supports RESTful routing. You can use Laravel's `make:controller` command to create a REST-enabled controller:

```
1   $ php artisan make:controller ListsController
2   Controller created successfully.
```

Regardless of which generator you use, you'll find the generated controller in app/Http/Con-
trollers/ListsController.php. Open the newly created app/Http/Controllers/ListsController.php
file and you'll find the following code (comments removed for reasons of space):

```php
1    <?php
2
3    namespace todoparrot\Http\Controllers;
4
5    use Illuminate\Http\Request;
6
7    use todoparrot\Http\Requests;
8    use todoparrot\Http\Controllers\Controller;
9
10   class ListsController extends Controller {
11
12       public function index()
13       {
14       }
15
16       public function create()
17       {
18       }
19
20       public function store()
21       {
22       }
23
24       public function show($id)
25       {
26       }
27
28       public function edit($id)
29       {
30       }
31
32       public function update($id)
33       {
34       }
35
```

```
36      public function destroy($id)
37      {
38      }
39
40  }
```

A controller is just a typical PHP class that extends Laravel's BaseController class. Because Laravel generates RESTful controllers by default, seven methods (referred to as *actions* in framework parlance) have been created, with each intended to correspond with an endpoint defined in the earlier table. However, Laravel doesn't know you intend to use the newly generated controller in a RESTful fashion until the route definitions are defined.

With the controller in place, you'll next need to update the routes.php file. Open app/Http/routes.php and add the following line:

```
1  Route::resource('lists', 'ListsController');
```

If you try to access /lists you'll be greeted with a blank page. This is because the ListsController's index action does not identify a view to be rendered, nor does a view even yet exist for that matter! Let's create a simple view for use in conjunction with the index action.

Begin by creating a new directory named lists inside resources/views. This is where all of the views associated with ListController will be housed. Next, create a file named index.blade.php, placing it inside this newly created directory. Add the following contents to it:

```
1  @extends('layouts.master')
2
3  @section('content')
4
5  <h1>Lists</h1>
6
7  @endsection
```

Next, open app/Http/Controllers/ListsController and modify the index action to look like this:

```
1  public function index()
2  {
3    return view('lists.index')
4  }
```

Save the file and navigate to /lists. You should see the application layout (created in Chapter 2) and the h1 header found in index.blade.php. Congratulations, you've just implemented your first RESTful Laravel controller! Next, we'll integrate the Todolist model into the ListsController controller.

Interacting with the Todolist Model

With the RESTful controller defined, we'll begin integrating model-related logic and work towards creating, retrieving, updating and deleting lists, in addition to carrying out other useful tasks. However before doing so let's focus solely on the syntax used to interact with the model. Fortunately, we can easily do so using the Tinker console first introduced in Chapter 1 and save a new list to the database. Begin by opening a new Tinker session:

```
1  $ php artisan tinker
2  Psy Shell v0.4.4 (PHP 5.5.21 — cli) by Justin Hileman
3  >>>
```

Create a new `Todolist` object. To save some typing, you can declare the namespace as demonstrated here:

```
1  >>> namespace todoparrot;
2  >>> $list = new Todolist;
```

Next, assign a list name and description:

```
1  >>> $list->name = 'San Juan Vacation';
2  => 'San Juan Vacation'
3  >>> $list->description = 'Pre-vacation planning';
4  => 'Pre-vacation planning'
```

You can retrieve the `$list` object's class name using PHP's `get_class()` function:

```
1  >>> echo get_class($list);
2  => todoparrot\Todolist
```

Finally, we'll use the Eloquent ORM's `save()` method to save the `$list` object to the database:

```
1  >>> $list->save();
2  => true
```

Once the record is saved, the object will be assigned an `id` value (presuming you're using auto-incrementing keys). You can see that value by referencing the `id` attribute:

```
1  >>> echo $list->id;
2  => 1
```

I'm jumping ahead a bit here, but you can also easily see how many records are in the database using Eloquent's count method:

```
1  >>> echo Todolist::all()->count();
2  => 1
```

The save, all, and count methods are just a few of the many features made available to your models thanks to the Eloquent ORM. We'll learn about many more in the sections to follow.

Open the database and you should see the newly added record. TODOParrot uses MySQL, and so I'll use the mysql command line client for the purposes of demonstration:

```
1  mysql> select * from todolists;
2  +----+----------+----------------------+---------------------+----------------\
3  ----+
4  | id | name     | description          | created_at          | updated_at     \
5       |
6  +----+----------+----------------------+---------------------+----------------\
7  ----+
8  |  1 | San Juan | Pre-vacation planning | 2015-06-25 23:38:34 | 2015-06-26 16:05\
9  :24 |
10 +----+----------+----------------------+---------------------+----------------\
11 ----+
12 1 row in set (0.00 sec)
```

Feel free to spend some more time experimenting with the Todolist model inside Tinker. In particular, be sure to add a few more records as we'll use them in the next section. Once you're done, exit the console like this:

```
1  >>> exit;
```

Integrating a Model Into Your Controller

Now that you have a bit of experience interacting with a Laravel model, let's integrate a Todolist model into the Lists controller. We'll return to the Lists controller's index action, which currently looks like this:

```
1  public function index()
2  {
3    return view('lists.index')
4  }
```

If you recall from the earlier introduction a RESTful controller's `index` action is typically used to display a list of records. So let's use the `Todolist` model in conjunction with this action and corresponding view to display a list of lists. Begin by importing the `todoparrot\Todolist` namespace into the controller. Strictly speaking you aren't obligated to do this but it will save some typing. The import should be placed at the very top of the controller alongside the other use statements:

```
1   <?php
2
3   namespace todoparrot\Http\Controllers;
4
5   use Illuminate\Http\Request;
6
7   use Illuminate\Routing\Controller;
8   use todoparrot\Todolist;
9
10  class ListsController extends Controller {
11
12  ...
13
14  }
```

Next, modify the `index` action to look like this:

```
1  public function index()
2  {
3    $lists = Todolist::all();
4    return view('lists.index')->with('lists', $lists);
5  }
```

Pretty simple, right? We're using the `all` method introduced in the last section to retrieve all of the `Todolist` records found in the `todolists` table. This returns an object of type `Illuminate\Database\Eloquent\Collection` which is among other things iterable! We want to iterate over that collection of records in the view, and so `$lists` is passed into the view.

Next, open the corresponding view (`resources/views/lists/index.blade.php`), and modify the `content` section to look like this:

```
1   <h1>Lists</h1>
2
3   @if ($lists->count() > 0)
4     <ul>
5       @foreach ($lists as $list)
6         <li>{{ $list->name }}</li>
7       @endforeach
8     </ul>
9   @else
10    <p>
11      No lists found!
12    </p>
13  @endif
```

The updated `index` view uses Eloquent's `count()` function to determine whether `$lists` contains at least one element. If so, the Blade templating engine's `@foreach` directive is used to iterate over `$lists`, with each retrieved element being an object of type `todoparrot\Todolist`. Each object's properties are exposed using PHP's standard object notation, meaning you could for instance access an object's `name` property using `$list->name`.

Save the changes, navigate to `/lists` and you should see a bulleted list of any TODO lists found in the `todolists` table! While an exciting development, I promise you we're just getting started!

Seeding the Database

At this point in the chapter I'm going to make what on the surface might be considered an odd segue. However because we're going to spend a lot of time experimenting with retrieving, updating and deleting records, I would like to eliminate much of the tedium otherwise required to create realistic test data. Fortunately, we can do so by *seeding* the database with test data. As an added bonus, you can use Laravel's database seeding capabilities for several other purposes, including conveniently populating helper tables. For instance if you wanted users to provide their city and state, then you'd probably require the user to choose the state from a populated select box of valid values (Ohio, Pennsylvania, Indiana, etc.). These values might be stored in a table named `states`. It would be quite time consuming to manually insert each state name and other related information such as the ISO abbreviation (OH, PA, IN, etc). Instead, you can use Laravel's seeding feature to easily insert even large amounts of data into your database.

You'll seed a database by executing the following artisan command:

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

If you execute this command now, nothing will happen. Actually, something *did* happen, it's just not obvious what. The `db:seed` command executes the file `database/seeds/DatabaseSeeder.php`, which looks like this:

```php
1   <?php
2
3   use Illuminate\Database\Seeder;
4   use Illuminate\Database\Eloquent\Model;
5
6   class DatabaseSeeder extends Seeder {
7
8     /**
9      * Run the database seeds.
10     *
11     * @return void
12     */
13    public function run()
14    {
15
16      Model::unguard();
17
18      // $this->call('UserTableSeeder');
19
20      Model::reguard();
21
22    }
23
24  }
```

You'll use DatabaseSeeder.php to pre-populate, or *seed*, your database. The run() method is where all of the magic happens. Inside this method you'll reference other seeder files, and when db:seed executes, the instructions found in those seeder files will be executed as well. The Laravel developers provide a commented-out example of how you'll reference this files:

```php
1   // $this->call('UserTableSeeder');
```

The UserTableSeeder.php file doesn't actually exist, but if it did it would be found in the database/seeds/ directory. Let's create a seeder for populating a few Todolist records. Create a new file named TodolistTableSeeder, placing it in database/seeds/. Add the following contents to it:

```php
1   <?php
2
3   use Illuminate\Database\Seeder;
4   use Illuminate\Database\Eloquent\Model;
5   use todoparrot\Todolist;
6
7   class TodolistTableSeeder extends Seeder {
8
9     public function run()
10    {
11
12      Todolist::create([
13        'name' => 'San Juan Vacation',
14        'description' => 'Things to do before we leave for Puerto Rico!'
15      ]);
16
17      Todolist::create([
18        'name' => 'Home Winterization',
19        'description' => 'Winter is coming.'
20      ]);
21
22       Todolist::create([
23        'name' => 'Rental Maintenance',
24        'description' => 'Cleanup and improvements for new tenants'
25      ]);
26
27    }
28
29  }
```

Save this file and then replace the `DatabaseSeeder.php` line `$this->call('UserTableSeeder');` with the following lines:

```php
1   DB::table('todolists')->delete();
2   $this->call('TodolistTableSeeder');
```

After saving the changes you'll need to rebuild the classmap:

```
1   $ composer dump-autoload
2   Generating autoload files
```

Finally, run the seeder anew:

```
1  $ php artisan db:seed
2  Seeded: TodolistTableSeeder
```

Check your project database and you should see several new records in the todolists table!

Creating Large Amounts of Sample Data

The above approach works fine when you'd like to create a small set of sample data, but what if you wanted to simulate a real-world data set involving hundreds or thousands of lists? Surely it wouldn't be a wise use of time to manually create each record as carried out in the previous example. Fortunately you can use the fantastic Faker library[103] (which is now included in Laravel 5.1+ projects by default) to easily create large amounts of sample data. Let's modify TodolistTableSeeder.php to generate 50 lists using Faker.

If you have a look at your project's composer.json file, you'll see the Faker package is included in require-dev as presented below:

```
1  "require-dev": {
2    "fzaninotto/faker": "~1.4",
3    ...
4  },
```

We'll use Faker to create fifty lists. Modify the TodolistTableSeeder's run() method to look like this:

```
1  class TodolistTableSeeder extends Seeder {
2
3    public function run()
4    {
5
6      $faker = \Faker\Factory::create();
7
8      Todolist::truncate();
9
10     foreach(range(1,50) as $index)
11     {
12
13         Todolist::create([
14           'name' => $faker->sentence(2),
15           'description' => $faker->sentence(4),
```

---
[103]https://github.com/fzaninotto/Faker

```
16            ]);
17
18        }
19
20    }
21
22  }
```

In this example a new instance of the `Faker` class is created. Next the `Todolist`'s underlying table (`todolists`) is truncated, meaning all of the existing records are removed. Strictly speaking you might not have to do this depending upon how your database tables are configured, however presuming you are using seeding as a development aid then you'll probably want to just repeatedly rebuild the same data set. Next, a `foreach` statement is used to loop over the `Todolist::create` fifty times, with each iteration resulting in Faker creating two random Lorem Ipsum[104]-style sentences consisting of two and four words, respectively.

After saving the changes, run `php artisan db:seed` again. After the command completes, enter your database and you should see fifty records that look similar to the records found below:

```
1  mysql> select name, description from todolists limit 4;
2  +--------------------+--------------------------------------------+
3  | name               | description                                |
4  +--------------------+--------------------------------------------+
5  | Sed voluptates.    | Accusamus sit et excepturi voluptas.       |
6  | Debitis dignissimos. | Cum quia ut.                             |
7  | Earum et dolore.   | Quis necessitatibus magnam deserunt error id. |
8  | Nesciunt sequi.    | Porro ratione non non.                     |
9  +--------------------+--------------------------------------------+
```

Random sentence generation is only a small part of what Faker can do. You can also use Faker to generate random numbers, lorem ipsum paragraphs, male and female names, U.S. addresses, U.S. phone numbers, company names, e-mail addresses, URLs, credit card information, colors, and more! Be sure to check out the Faker documentation[105] for examples of these other generators. We'll also return to Faker throughout the remainder of the book to generate various data sets, so stay tuned!

Finding Data

Most Laravel queries are very straightforward in that they'll simply involve retrieving a record based on its primary key or some other filter, while others require more sophisticated approaches involving

---

[104]http://en.wikipedia.org/wiki/Lorem_ipsum
[105]https://github.com/fzaninotto/Faker

multiple parameters, complex sorting, and table joins. Fortunately Laravel offers an incredibly rich set of methods for querying data in a variety of fashions. In this section I'll show you the many ways in which data can be retrieved from your application's database.

Retrieving All Records

Perhaps the easiest query involves retrieving all of a table's records using the `all` method, which you were introduced to earlier in the chapter. To recap, the following example will retrieve all of the lists:

```
1  $lists = Todolist::all();
```

The `$lists` variable is an instance of `Illuminate\Database\Eloquent\Collection`, which is among other things iterable. This means you can loop over the records using standard PHP syntax. Fire up tinker and try it for yourself:

```
1  >> namespace todoparrot;
2  >>> $lists = Todolist::all();
3  => <Illuminate\Database\Eloquent\Collection...
4  >>> foreach ($lists as $list) {
5  ... printf("%s\n", $list->name);
6  ... }
7  San Juan Vacation
8  Home Winterization
9  Rental Maintenance
```

Alternatively, because the results are returned as an Eloquent *collection*, you have access to a variety of useful methods, including `each`. In the following example I'll use `each` in conjunction with a closure to arguably more eloquently iterate over the results:

```
1  >>> $lists = todoparrot\Todolist::all();
2  >>> $lists->each(function($list) {
3  ... echo $list->name;
4  ... });
```

You'll likely rarely want to use the `all()` method unless the target model is associated with a trivial (less than one thousand) number of records. However if you would like to provide a solution for viewing a large number of records, consider *paginating* the results. I'll introduce Laravel's pagination feature in the later section, "Paginating Results".

> If you're using a custom Artisan command to process large numbers of records, check out Laravel's `chunk` method.

Retrieving Records by Primary Key

When viewing a list detail page or user profile, or updating a particular record, you'll want to unmistakably retrieve the desired record, done by querying for the record using its primary key. To do so you'll use the `find` method, passing along the primary key:

```
1  [1] > $list = todoparrot\Todolist::find(3);
2  [2] echo $list
3  { "id":3,
4    "name":"Rental Maintenance",
5    "description":"Cleanup and improvements for new tenants",
6    "created_at":"2015-06-02 17:38:55",
7    "updated_at":"2015-06-02 17:38:55"
8  }
```

Implementing the RESTful Show Action

Now that you know how to use the `find` method, let's implement the `Lists` controller's `show` action. The action currently looks like this:

```
1  public function show($id)
2  {
3  }
```

Because the `show` action is intended to display a specific instance of a particular resource, the action is automatically configured to pass along the resource's ID via the `$id` variable. We'll use the `find` method to retrieve the desired record, and then pass the `Todolist` object into the view. The updated `show` action looks like this:

```
1  use todoparrot\Todolist;
2
3  ...
4
5  public function show($id)
6  {
7    $list = Todolist::find($id);
8    return view('lists.show')->with('list', $list);
9  }
```

Next we'll create the corresponding view. Create a file named `show.blade.php`, placing it in the directory `resources/views/lists`. Add the following contents to it:

```
1  @extends('layouts.master')
2
3  @section('content')
4
5  <h1>{{ $list->name }}</h1>
6
7  <p>
8  Created on: {{ $list->created_at }}
9  Last modified: {{ $list->updated_at }}<br />
10 </p>
11
12 <p>
13 {{ $list->description }}
14 </p>
15
16 @endsection
```

After saving the changes, navigate to the `Lists` controller's `show` URI (append an appropriate ID to `/lists/`) and you should see output that looks something like this (HTML tags included for clarity):

```
1  <h1>Rental Maintenance</h1>
2  <p>
3  Created on: 2015-06-02 17:38:55 Last modified: 2015-06-02 17:38:55<br />
4  Cleanup and improvements for new tenants
5  </p>
```

Brilliant! We're now able to view more information about a specific list. But what happens if the user attempts to access a record that doesn't exist, such as `/lists/23245`? The `find` method would return a null value, meaning any attempts to retrieve a property of a non-object within the view would result in a view exception. Logically you'll want to avoid this sort of mishap, and can do so using the `findOrFail` method instead:

```
1  $list = Todolist::findOrFail($id);
```

If the desired record is not found, an exception of type `ModelNotFoundException` will be thrown. You can listen for these exceptions by registering an error handler in `app/Exceptions/Handler.php`, and redirecting users encountering a `ModelNotFoundException` exception to a custom 404 view. Begin by creating a file named `404.blade.php` in `resources/views/errors/404.blade.php`. Add a simple message to it for the moment, something like `File not found` will suffice. Obviously you'll want to further customize this 404 page before deploying to production.

Next, open up `app/Exceptions/Handler.php` and add the following method:

```
1   protected function renderModelNotFoundException(ModelNotFoundException $e)
2   {
3
4     if (view()->exists('errors.404'))
5     {
6       return response()->view('errors.404', [], 404);
7     }
8     else
9     {
10      return (new SymfonyDisplayer(config('app.debug')))
11        ->createResponse($e);
12    }
13  }
```

This method borrows heavily from the native renderHttpException method found in the following file:

```
1   vendor/laravel/framework/src/Illuminate/Foundation/Exceptions/Handler.php
```

It will render the previously created 404.blade.php view if the view exists, or otherwise display a more terse exception-related message. Next, modify the app/Exceptions/Handler.php's render method to look like this:

```
1   use Illuminate\Database\Eloquent\ModelNotFoundException;
2
3   ...
4
5   public function render($request, Exception $e)
6   {
7
8     if ($e instanceof ModelNotFoundException)
9     {
10      return $this->renderModelNotFoundException($e);
11    }
12    else
13    {
14      return parent::render($request, $e);
15    }
16
17  }
```

The modified code extends what's already being done to render HTTP exceptions, adding additional functionality to handle exceptions of type `ModelNotFound` (`ModelNotFoundException`).

Finally, test it out by accessing some record you know to not exist, such as `/lists/asdf`. The `asdf` key will be passed to `findOrFail`, presumably not be found, and send the user to the 404 page.

Selecting Specific Columns

For performance reasons you should to construct queries that retrieve the minimal data required to complete the desired task. For instance if you're constructing a list view that only displays the list name and description, there is no reason to retrieve the `id`, `created_at`, and `updated_at` columns. You can restrict which columns are selected using the `select` method, as demonstrated here:

```
1  $ php artisan tinker
2  >>> $lists = todoparrot\Todolist::select('name', 'description')->first();
3  >>> echo $lists->name;
4  Sed voluptates.
5  >>> echo $lists;
6  {"name":"Sed voluptates.","description":"Accusamus sit et excepturi."}
```

Counting Records

To count the number of records associated with a given model, use the `count` method:

```
1  >> echo todoparrot\Todolist::count();
2  50
```

You can also use `count` to determine how many records have been selected:

```
1  $lists = Todolist::all();
2  ...
3  {{ $lists->count() }} records selected.
```

Ordering Records

You can order records using the `orderBy` method. You'll use this method in conjunction with `get`. The following example will retrieve all `Todolist` records, ordered by name:

```
1  $lists = Todolist::orderBy('name')->get();
```

> You'll use the `get()` method to retrieve records using methods other than `all` or `find`.

Laravel will by default sort results in ascending order. You can change this default behavior by passing the desired order (`ASC` or `DESC`) as a second argument to `orderBy`:

```
1    $lists = Todolist::orderBy('name', 'DESC')->get();
```

You can order results using multiple column by calling `orderBy` multiple times:

```
1    $lists = Todolist::orderBy('created_at', 'DESC')
2      ->orderBy('name', 'ASC')
3      ->get();
```

This is equivalent to executing the following SQL statement:

```
1    SELECT * FROM todolists ORDER BY created_at DESC, name ASC;
```

Using Conditional Clauses

While the `find` method is useful for retrieving a specific record, you'll often want to find records using other attributes. You can do so using the `where` method. Suppose the `Todolist` model included a Boolean `complete` attribute, intended to denote whether the user considered the list completed. You could use `where` to retrieve a set of completed lists:

```
1    $lists = Todolist::where('complete', '=', 1)->get();
```

Notice how the attribute, comparison operator, and value are passed into `where` as three separate arguments. This is done as a safeguard against attacks such as SQL injection. If the comparison operator is =, you can forgo providing the equal operator altogether:

```
1    $lists = Todolist::where('complete', 1)->get();
```

However, if you're using an operator such as › or ‹, you are logically required to expressly supply the operator. You can alternatively use the `whereRaw` method (without sacrificing security) to accomplish the same result:

```
1    $lists = Todolist::whereRaw('complete = ?', 1)->get();
```

Grouping Records

Grouping records according to a shared attribute provides opportunities to view data in interesting ways, particularly when grouping is performed in conjunction with an aggregate SQL function such as `count()` or `sum()`. Laravel offers a method called `groupBy` that facilitates this sort of query. Suppose you wanted to retrieve the years associated with all lists' creation dates and the count of lists associated with each year. You could construct the query in MySQL like so:

```
 1  mysql> select year(created_at) as `year`, count(name) as `count`
 2      -> from todolists
 3      -> group by `year` order by `count` desc;
 4  +---------------+
 5  | year | count |
 6  +------+-------+
 7  | 2015 |   398 |
 8  | 2014 |   247 |
 9  | 2013 |   112 |
10  | 2012 |   92  |
11  | 2011 |   14  |
12  +---------------+
13  5 rows in set (0.00 sec)
```

This query can be reproduced in Laravel like so:

```
 1  use DB;
 2
 3  ...
 4
 5  $lists = Todolist::select(DB::raw('year(created_at) as year'),
 6          DB::raw('count(name) as `count`'))
 7          ->groupBy('year')
 8          ->orderBy('`count` desc')->get();
```

Another new concept was introduced with this example: DB::raw. Eloquent currently does not support aggregate functions however you can use Laravel's Query Builder interface in conjunction with Eloquent as a convenient workaround. The DB::raw method injects raw SQL into the query, thereby allowing you to use aggregate functions within select. I'll talk more about Query Builder's capabilities in the later section, "Introducing Query Builder".

You can then iterate over the year and count attributes as you would any other:

```
 1  @if ($lists->count() > 0)
 2    <ul>
 3      @foreach ($lists as $list)
 4        <li>{{ $list->count }} lists created in {{ $list->year }}</li>
 5      @endforeach
 6    </ul>
 7  @else
 8    <p>
 9      No lists found!
10    </p>
11  @endif
```

You'll often want to group records in conjunction with a filter. For instance, what if you only wanted to retrieve a grouped count of arcade games released by year in the years after 2010? You could use `groupBy` in conjunction with `where`:

```
1  $lists = Todolist::select(
2          DB::raw('year(created_at) as year'),
3          DB::raw('count(name) as count'))
4          ->groupBy('year')
5          ->where('year', '>', '2010')->get();
```

This works because you're filtering on the non-aggregated field. What if you wanted to instead retrieve the same information, but only those years in which more than 50 lists were created? At first blush it would seem you could use `group` in conjunction with `where` to filter the results, however as it turns out you can't use `where` to filter anything calculated by an aggregate function, because `where` applies the defined condition *before* any results are calculated, meaning it doesn't know anything about the `count` alias at the time it attempts to perform the filter. Instead, when you desire to filter on the aggregated result, you'll use `having`. Let's revise the previous broken example to use `having` instead of `where`:

```
1  $lists = Todolist::select(
2          DB::raw('year(created_at) as year'),
3          DB::raw('count(name) as count'))
4          ->groupBy('year')
5          ->having('year', '>', '2010')->get();
```

Limiting Returned Records

Sometimes you'll want to just retrieve a small subset of records, for instance the five most recently added lists. You can do so using the oddly-named `take` method:

```
1  $lists = Todolist::take(5)->orderBy('created_at', 'desc')->get();
```

If you wanted to retrieve a subset of records beginning at a certain offset you can combine `take` with `skip`. The following example will retrieve five records beginning with the sixth record:

```
1  $lists = Todolist::take(5)->skip(5)->orderBy('created_at', 'desc')->get();
```

If you're familiar with SQL the above command is equivalent to executing the following statement:

```
1    SELECT * from todolists ORDER BY created_at DESC LIMIT 5 OFFSET 5;
```

Retrieving the First or Last Record

It's often useful to retrieve just the first or last record found in a collection. For instance you might want to offer users a visual aide highlighting the most recently created list:

```
1    $list = Todolist::orderBy('created_at', 'desc')->first();
```

To retrieve a collection's last record, you can also use `first()` and reverse the order. For instance to retrieve the oldest list, you'll use the same snippet as above but instead order the results in ascending fashion:

```
1    $list = Todolist::orderBy('created_at', 'asc')->first();
```

Retrieving a Random Record

There are plenty of reasons you might wish to retrieve a random record from your project database. Perhaps a future version of TODOParrot would highlight an incomplete list in the hopes of spurring the user into action. You might at first glance conclude the following approach is the most straightforward:

```
1    $list = Todolist::all()->random(1);
```

Eloquent collections support the `random` method, which retrieves one or more random records from a collection. However unless the model's underlying table size is vanishingly small, you should not use this approach, because it requires *all* records to first be retrieved from the database! Instead, you can use the `DB::raw` method first used in the section "Grouping Records" to pass the SQL `RAND()` function into `orderBy`, as demonstrated here:

```
1    $list = Todolist::orderBy(DB::raw('RAND()'))->first();
```

This is equivalent to executing the following SQL:

```
1    select * from `todolists` order by RAND() asc limit 1
```

Determining Existence

If your sole goal is to determine whether a particular record exists, *without* needing to actually load the record if it does, use the `exists` method. For instance to determine whether a list category named `Home` exists use the following statement:

```
1  $exists = Todolist::where('name' , '=', 'San Juan Vacation')->exists();
```

Using `exists` instead of attempting to locate a record and then examining the object or counting results is preferred for performance reasons, because `exists` produces a query that just counts records rather than retrieving them:

```
1  select count(*) as aggregate from `todolists`
2  where `name` = 'San Juan Vacation'
```

Paginating Results

If users only planned on maintaining a few lists then retrieving and displaying the lists using the `all` method is going to do the job nicely. However the TODOParrot team is intent on becoming the world's most popular TODO list management company, hopefully culminating in users creating and maintaining dozens if not hundreds of lists for all of life's activities. To help users quickly and easily find a desired list you'll probably want to *paginate* them across multiple pages.

Database pagination is accomplished using a series of queries involving `limit` and `offset` clauses. For instance, to retrieve lists in batches of 10 sorted by name you would execute the following queries (MySQL, PostgreSQL and SQLite):

```
1  SELECT id, name FROM todolists ORDER BY name ASC LIMIT 10 OFFSET 0
2  SELECT id, name FROM todolists ORDER BY name ASC LIMIT 10 OFFSET 10
```

Incidentally, MySQL, PostgreSQL and SQLite all use a 0-based index, meaning executing the first query is the same as executing:

```
1  SELECT id, name FROM locations ORDER BY name ASC LIMIT 10
```

Therefore when creating a pagination solution you would need to keep track of the current offset and limit values, the latter of which might be variable if you gave users the opportunity to adjust the number of items presented per page. Further, you would also need to create a user interface for allowing the user to navigate from one page to the next. Fortunately, pagination is a key feature of many web applications, meaning turnkey solutions are often incorporated into frameworks, Laravel included!

To paginate results, you'll use the `paginate` method:

```
1  $lists = Todolist::orderBy('created_at', 'desc')->paginate(10);
```

In this example we're overriding the default number of records retrieved (15), lowering the number retrieved to 10. Once retrieved, you can iterate over the collection using `@foreach` just as you would were pagination not being used. You'll however want to make a slight modification to the view, adding the pagination ribbon:

```
1   {!! $lists->render() !!}
```

This will create a stylized list of links to each available page, similar to the screenshot presented below.

Laravel's pagination ribbon

Inserting New Records

Laravel offers a few different approaches to creating new records. The first involves the save method, which was briefly introduced earlier in the chapter. To save a record using save, you'll first create a new instance of the desired model, update its attributes, and then execute the save method:

```
1   $list = new Todolist;
2   $list->name = 'San Juan Vacation';
3   $list->description = 'Pre-vacation planning';
4   $list->save();
```

Presuming your underlying table incorporates the default id, created_at and updated_at fields, Laravel will automatically update the values of these fields for you.

You can alternatively use the create method, simultaneously setting and saving the model attributes:

```
1   $list = Todolist::create(
2           array('name' => 'San Juan Vacation',
3                   'description' => 'Pre-vacation planning')
4           );
```

Creating a Record if It Doesn't Exist

It's also possible to create a new record only if a record with a matching attribute isn't found:

```
1  $list = Todolist::firstOrCreate(array('name' => 'San Juan Vacation'));
```

Keep in mind however that `firstOrCreate` will fail should you neglect to provide values for any fields not associated with default values. Of course, this is a catch-22, because identifying these attributes and values within `firstOrCreate` means the filter will additionally use those attributes in an attempt to find a matching record, which is likely not the behavior you desire. Instead, you'll probably want to use `firstOrNew`, because it will just create a new model instance if a record matching the provided attribute isn't found:

```
1  $list = Todolist::firstOrNew(array('name' => 'San Juan Vacation'));
2  $list->description('Too much to do before vacation!');
3  $list->save();
```

However, if your intent is to update a record if it exists or create a new record if not match is found, you might consider using `updateOrCreate`. It *does* allow you to specify an attribute argument separately from the values you'd like to create or update depending upon whether a record is found:

```
1  $list = Todolist::updateOrCreate(
2    array('name' => 'San Juan Vacation'),
3    array('description' => 'Too much to do before vacation!')
4  );
```

The first array defines the attributes used to determine whether a matching record exists, and the second array identifies the attributes and values which will be inserted or updated based on the outcome. If the former, then the attributes found in the first array will be inserted into the new record along with the attributes found in the second array.

ⓘ Implementing the RESTful Insert Feature

The RESTful `Lists` controller created earlier in the chapter includes two actions (`create` and `store`) that work together to insert new records into the `lists` table. The `create` action is responsible for presenting the web form used to input the new list. This form is submitted to the `store` action, which is responsible for inserting the data into the database. Because several key concepts which have not yet been introduced play an integral role in implementation of these two actions (namely form, validation, and Laravel 5's new form request feature), If you simply can't stand the suspense, jump ahead to Chapter 5 to review the implementation.

Updating Existing Records

Users will logically want to update existing lists, perhaps tweaking the list name or description. To do so, you'll typically retrieve the desired record using its primary key, update the attributes as necessary, and use the save method to save the changes:

```
1  $list = Todolist::find(14);
2  $list->name = 'San Juan Holiday';
3  $list->save();
```

ⓘ Implementing the RESTful Update Feature

As with the aforementioned RESTful insertion feature, I'll hold off on demonstrating how to implement the RESTful update feature until additional key concepts are introduced.

Deleting Records

To delete a record you'll use the delete method:

```
1  $list = Todolist::find(12);
2  $list->delete();
```

You can optionally consolidate these two commands using the destroy method:

```
1  Todolist::destroy(12);
```

Implementing the RESTful Destroy Method

The Lists controller's destroy method is the easiest to implement, because typically a companion view isn't required. You'll just delete the desired record and redirect the user to a designated location. Below is the modified Lists controller's destroy method, which accepts the ID of the record designated for deletion and then redirects the user to the Lists controller's index action:

```
1   public function destroy($id)
2   {
3     Todolist::destroy($id);
4     return \Redirect::route('lists.index');
5   }
```

While this bit of logic is easy enough, it doesn't shed any insight into how the user actually executes this action, particularly because as the table found in the earlier section "Creating a RESTFul Controller" indicates, this route is only accessible via the DELETE method. You're probably familiar with the GET and POST methods, however unless you have prior experience implementing RESTFul applications then DELETE is probably entirely unfamiliar. Not to worry! In Chapter 5 I'll go into detail regarding how Laravel implements the DELETE method.

Soft Deleting Records

In many cases you won't ever actually want to truly remove records from your database, but instead annotate them in such a way that users perceive them to be deleted. This is known as a *soft delete*. Laravel natively supports soft deletion, requiring just a few configuration changes to ensure a model's records aren't actually deleted when delete or destroy are executed. As an example let's modify the Todolist model to support soft deletion. Begin by creating a new migration that adds a column named deleted_at to the todolists table:

```
1   $ php artisan make:migration add_soft_delete_to_todolists --table=todolists
2   Created Migration: 2015_06_07_203253_add_soft_delete_to_todolists
3   Generating optimized class loader
```

Next open up the newly created migration (found in the database/migrations directory), and modify the up and down methods to look like the following:

```
1   public function up()
2   {
3     Schema::table('todolists', function(Blueprint $table)
4     {
5       $table->softDeletes();
6     });
7   }
8
9   public function down()
10  {
11    Schema::table('todolists', function(Blueprint $table)
12    {
13      $table->dropColumn('deleted_at');
14    });
15  }
```

Save the changes and run the migration:

```
1   $ php artisan migrate
```

After the migration has completed you'll next want to open up the target model (again I'm using `Todolist` as an example although this feature isn't actually integrated into TODOParrot) and use the `SoftDeleting` trait:

```php
1   <?php namespace todoparrot;
2
3   use Illuminate\Database\Eloquent\Model;
4   use Illuminate\Database\Eloquent\SoftDeletes;
5
6   class Todolist extends Model {
7
8       use SoftDeletes;
9
10      protected $dates = ['deleted_at'];
11
12  }
```

Although not strictly necessary, adding the `deleted_at` attribute to the `$dates` array as demonstrated above will cause any returned `deleted_at` values to be of type `Carbon` as first discussed in the earlier section "Defining Methods".

After saving these changes, the next time you delete a record associated with this model, the `deleted_at` column will be set to the current timestamp. Any record having a set `deleted_at` timestamp will not be included in any retrieved results, thereby seemingly having been deleted. Of course, there are plenty of practical reasons why you might want to at some point include soft deleted records in your results (for instance giving users the ability to recover a previously deleted record). You can do so using the `withTrashed` method:

```php
1   $lists = Todolist::withTrashed()->get();
```

Introducing Query Builder

Chances are you're going to be able to successfully carry out 99% of the database operations you desire using Eloquent, however you'll occasionally want to exercise a bit of additional control over your queries. Enter *Query Builder*, Laravel's alternative approach to querying your project database. Because the majority of your projects will be Eloquent-driven I don't want to dwell on Query Builder too much, but think this chapter would be incomplete without at least a brief introduction.

You can retrieve all of the records found in the `todolists` table using Query Builder like this:

```
1   $lists = DB::table('todolists')->get();
```

This returns an array of objects of type stdClass, meaning you can iterate over the returned objects like this:

```
1   foreach ($lists as $list) {
2     echo $list->name;
3   }
```

If you're looking for a specific record and want to search for it by ID, you can use find:

```
1   $list = DB::table('todolists')->find(52);
```

If you're only interested in retrieving the name column, there's no sense retrieving the descriptions and other columns. You can use select to limit the results accordingly:

```
1   $lists = DB::table('todolists')->select('name')->get();
```

Finally, there are instances where it makes more sense to directly execute raw SQL. You can do this using several different approaches. To select data, you can use DB:select:

```
1   $lists = DB::select('SELECT * from todolists');
```

This returns an array of objects as was the case with the introductory example in this section. If you wanted to insert, update, or delete data using raw SQL, you can use the DB::insert, DB::update, and DB::delete methods, respectively:

```
1   DB::insert('insert into todolists (name, description) values (?, ?)',
2     array('San Juan Vacation', 'Things to do before vacation');
3
4   DB::update('update todolists set completed = 1 where id = ?', array(52));
5
6   DB::delete('delete from todolists where completed = 1');
```

If you wanted to run SQL that isn't intended to interact with the data directly, perhaps something of an administrative nature, you can use DB::statement:

```
1   $lists = DB::statement('drop table todolists');
```

As mentioned, this isn't intended to be anything more than a brief introduction to Query Builder. See the documentation[106] for a much more comprehensive summary of what's available.

---

[106]http://laravel.com/docs/master/queries

Creating Sluggable URLs

Frameworks such as Laravel do a great job of creating user-friendly URLs by default, meaning the days of creating applications sporting ugly URLs like this are long gone:

```
1   http://todoparrot.com/lists.php?id=12
```

Instead, Laravel will transform a URL like the above into something much more readable, such as:

```
1   http://todoparrot.com/lists/12
```

However, while an improvement this URL really isn't particular informative. After all, what does the 12 even mean to the user? You and I know it is an integer value representing the primary key of a record found in the lists table, but it would be much more practical to instead use a URL that looks like this:

```
1   http://todoparrot.com/lists/san-juan-vacation
```

This string-based parameter is known as a *slug*, and thanks to the eloquent-sluggable[107] package it's surprisingly easy to integrate sluggable URLs into your Laravel application. Begin by adding the eloquent-sluggable package to your composer.json file:

```
1   $ composer require cviebrock/eloquent-sluggable >=3.0.0-alpha
```

> ⚠ At the time of this writing I had to instead reference the master (dev-master) branch in order to workaround a conflict with the latest version of Laravel. Have a look at the project's composer.json file to see how this is done.

Next, add the SluggableServiceProvider to your config/app.php file's providers array:

```
1   'providers' => [
2       ...
3       'Cviebrock\EloquentSluggable\SluggableServiceProvider',
4       ...
5   ],
```

After saving these changes, publish the package's configuration file by executing the following command:

---

[107]https://github.com/cviebrock/eloquent-sluggable

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

You'll find the configuration file in `config/sluggable.php`. While there's nothing you need to change in this file in order to begin using the package, I nonetheless suggest having a look at it as there are several opportunities to override various default settings if necessary.

Creating Sluggable Models

With the eloquent-sluggable package installed you'll need to update your models and underlying tables to enable the sluggable feature. This is fortunately incredibly easy to do. For instance, to add slugs to the `Todolist` model, modify it like so:

```php
1  use Cviebrock\EloquentSluggable\SluggableInterface;
2  use Cviebrock\EloquentSluggable\SluggableTrait;
3
4  class Todolist extends Model implements SluggableInterface {
5
6      use SluggableTrait;
7
8      protected $sluggable = array(
9          'build_from' => 'name',
10         'save_to'    => 'slug',
11     );
12
13     ...
14
15 }
```

There are several important changes to this model, including:

- The eloquent-sluggable interface and trait are referenced at the top of the file.
- The model continues to extend `Model` but additionally implements `SluggableInterface`.
- Inside the class body you'll see the model uses the `Sluggable` trait.
- An array-based property named `$sluggable` is defined which identifies the database column which should be used to create the slug and the database column to which the slug should be saved.

After saving the model changes you'll need to create a migration which adds the `slug` column to the table. You can do this manually or preferably use the following Artisan command eloquent-sluggable makes available to you:

```
1  $ php artisan sluggable:table todolists
```

After running the migration, the eloquent-sluggable package will *automatically* create the slugs for you any time a new record is added to the `todolists` table! Of course, if you're adding this capability to an existing table that already contains records then you'll need to update each record. One of the easiest ways to do this is by entering the `Tinker` console, selecting all records and then saving them back to the database after using the eloquent-sluggable's `resluggify` method:

```
1  $ php artisan tinker
2  >>> namespace Todoparrot;
3  >>> $lists = Todolist::all();
4  >>> foreach ($lists as $list) {
5  ... $list->resluggify();
6  ... $list->save();
7  ... }
8  >>>
```

With the slugs in place, all you need to do is retrieve the desired record by slug rather than the integer ID. The package provides a useful helper method for doing so called `findBySlug`. For instance to find the list record associated with the slug `san-juan-vacation` you'll execute the following command:

```
1  $category = Todolist::findBySlug('san-juan-vacation');
```

Of course, when working within an action you'll be passing along the slug as the ID so for instance you could modify the `Lists` controller's `show` action you'll see the `$id` parameter is passed into the `findBySlug` method like so:

```
1  public function show($id)
2  {
3      $list = Todolist::findBySlug($id);
4      return view('todolist.show')->with('list', $list);
5  }
```

As this section hopefully indicates, integrating sluggable URLs into your application is incredibly easy, and certainly improves the readability of your URLs!

Testing Your Models

Testing your models to ensure they are performing as desired is a crucial part of the application development process. Mind you, the goal isn't to test Eloquent's features; one can presume Eloquent

continues to be tested by the Laravel development team and community at large. Instead, you want to focus on confirming proper functioning of features you incorporate into the application models, such as whether your model accessors and mutators are properly configured, whether your custom methods are behaving as expected, whether your validators are properly constraining input, and as you'll learn in the next chapter, whether features such as relations and scopes are correctly defined. With this in mind, let's take some time to investigate a few testing scenarios. I'll presume you've successfully configured your Laravel testing environment as described in Chapter 1.

Configuring the Test Database

Because you'll want to test your application in conjunction with some realistic data you'll need to configure a test-specific database. If you're using PHPUnit the easiest way to do so in Laravel 5 is by overriding the `.env` configuration variables at the time the tests are executed. You can easily do this by adding the database configuration environment variables to the `phpunit.xml` file:

```
1  <php>
2      ...
3      <env name="DB_DATABASE" value="testing_todoparrot_com"/>
4  </php>
```

This will result in `config/database.php` using the same `DB_HOST`, `DB_USERNAME`, and `DB_PASSWORD` environment variables as those defined in your `.env` file but use the overridden `DB_DATABASE` configuration variable defined in `phpunit.xml`. Of course if you feel the need to configure a different username and password for the test database, you can easily override those variables as well within `phpunit.xml`.

Automatically Running Database Migrations

It is crucial for you to ensure that your database structure and test data are in a *known state* prior to the execution of each and every test, otherwise you're likely to introduce all sorts of uncertainty into the very situations you're trying to verify. One foolproof way to do this is by completely tearing down and rebuilding your test database structure prior to and following each test, respectively. In prior versions of this book I showed you a fairly involved solution for doing so, however as of Laravel 5.1 it is a trivial task! All you need to do is add the following `use DatabaseMigrations` statement to your test class:

```php
1   <?php
2
3   use Illuminate\Foundation\Testing\WithoutMiddleware;
4   use Illuminate\Foundation\Testing\DatabaseMigrations;
5   use Illuminate\Foundation\Testing\DatabaseTransactions;
6
7   use todoparrot\Todolist;
8
9   class TodolistTest extends TestCase
10  {
11
12      use DatabaseMigrations;
13      ...
14
15  }
```

Once declared, Laravel will automatically rollback and execute your migrations for each and every test!

Creating a Model Factory

In addition to test-oriented database migrations, Laravel 5.1 introduces a great new feature known as a *factory*. Factories are useful for generating sample data which can then be used in your tests. Laravel 5.1+ applications include an example User model factory (this model is introduced in chapter 5), found in database/factories/ModelFactory.php:

```php
1   $factory->define(todoparrot\User::class, function ($faker) {
2       return [
3           'name' => $faker->name,
4           'email' => $faker->email,
5           'password' => str_random(10),
6           'remember_token' => str_random(10),
7       ];
8   });
```

This factory uses the previously introduced Faker package to generate a placeholder name and e-mail address, and additionally uses PHP's str_random() function to generate a random password and password recovery token. Keep in mind you're free to set any column using a static value, although tools such as Faker will save you quite a bit of hassle when you'd like to create a large number of sample records.

Add the following Todolist factory below the factory defined above:

```
1  $factory->define(todoparrot\Todolist::class, function ($faker) {
2      return [
3          'name' => 'blah',
4          'description' => $faker->name
5      ];
6  });
```

With the factory created you can then reference it within your tests like so:

```
1  $listFactory = factory('todoparrot\Todolist')->create();
```

Executing the factory in conjunction with `create()` will cause the record to be *saved* to the database. You could optionally just create an object of type `Todolist` containing the information found in the factory by instead using `make()`:

```
1  $listFactory = factory('todoparrot\Todolist')->make();
```

Perhaps not surprisingly the latter approach will be faster, and so is recommended when there is no need to subsequently retrieve factory-generated data from the database during the course of the test.

Creating Your First Model Test

To begin, create a directory named `models` inside your project's `test` directory. Keep in mind this is purely for organizational purposes, and you're free to create any directory you please (or use none at all, however for the remainder of this chapter I'll presume you're following my cues. Next, create a file named `TodolistTest.php`, placing it in the `models` directory. Add the following contents to this file:

```
1  use Illuminate\Foundation\Testing\WithoutMiddleware;
2  use Illuminate\Foundation\Testing\DatabaseMigrations;
3  use Illuminate\Foundation\Testing\DatabaseTransactions;
4
5  use todoparrot\todolist;
6
7  class TodolistTest extends TestCase
8  {
9
10     use DatabaseMigrations;
11
12     public function testCanInstantiateTodolist()
```

```
13   {
14
15      $list = new Todolist;
16
17      $this->assertEquals(get_class($list), 'todoparrot\Todolist');
18
19   }
20
21 }
```

This is a pretty trivial test, intended to confirm we can instantiate the `Todolist` class, that it is part of the `todoparrot` namespace, and that it is of type `Todolist`. Execute the test like so:

```
1  $ vendor/bin/phpunit tests/models/TodolistTest.php
2  PHPUnit 4.7.2 by Sebastian Bergmann and contributors.
3
4  .
5
6  Time: 147 ms, Memory: 14.50Mb
7
8  OK (1 test, 1 assertion)
```

Great! Let's try something a tad more involved. Earlier in the chapter we configured a presence validator for the `Todolist` model's `name` attribute. Let's confirm the validator is indeed working as expected. Add the following method to the `TodolistTest` class:

```
1  public function testNotValidWhenNameMissing()
2  {
3
4    $t = new Todolist;
5
6    $this->assertFalse($t->validate());
7
8  }
```

How about a test involving the previously created `Todolist` factory? Add the following method to the `TodolistTest` class:

```
1    public function testTodolistRecordCount()
2    {
3
4        $listFactory = factory('todoparrot\Todolist')->create();
5
6        $lists = Todolist::all();
7
8        $this->assertEquals($lists->count(), 1);
9
10   }
```

What other tests would be useful? Try adding a method to your model and confirming it is producing the intended outcome!

Summary

Now that you have a rudimentary understanding of how to create, extend, and validate models, retrieve and manipulate data, seed your project database, and test your models, let's move on to some more advanced model-related concepts that will really kick your application into high gear!

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

Thus far we've been taking a fairly simplistic view of the TODOParrot database, creating and interacting with a single model (`Todolist`) and its underlying `todolists` table. However, in the real world an application's database tables are like an interconnected archipelago, with bridges connecting two or even more islands together. These allegorical bridges make it possible to determine which tasks are associated with a particular list, associate a specific user with a set of lists, and ensure all users identified as living in the state of Ohio can unmistakably be identified as such. Such relationships are possible thanks to a process known as *database normalization*[108], an approach to data organization that formally structures relations, eliminates redundancy, and improves maintainability. Laravel works in conjunction with a normalized database to provide powerful features useful for building and traversing relations with incredible dexterity. In this chapter I'll introduce you to these wonderful capabilities, and additionally demonstrate several other advanced model-related features such as scopes and eager loading.

Introducing Relations

Although relatively simplistic as compared to other applications, the production TODOParrot database very much resembles the allegorical archipelago mentioned at the beginning of this chapter. For instance, each list found in the `todolists` table is mapped to a single category (categories are defined in the `categories` table) and user (users are managed in the `users` table). Each task (stored in the `tasks` table) is tied to a list. So how are these relations formally defined in a Laravel application? Furthermore, how does one go about traversing a relation to for instance know specifically which tasks are associated with a given list? Thanks to a Laravel feature known as *relations*, such capabilities are surprisingly easy once you have the hang of things.

Laravel supports several types of relations, including:

- The One-to-One Relation: One-to-One relations are used when one entity can only belong with one other entity. For instance for organizational reasons you might choose to separate user authentication information (e-mail address and password) from profile-related characteristics such as their name, bio and phone number. Because one user can be associated with only one profile, and one profile can be associated with only one user, this would be an ideal one-to-one relation. In many cases one-to-one relations come about purely for organizational purposes, because there is typically little reason to otherwise manage uniquely-related data in multiple tables.

---

[108]http://en.wikipedia.org/wiki/Database_normalization

- The Belongs To Relation: The Belongs To relation assigns ownership of a particular record to another record. For instance, a task *belongs to* a list by virtue of the task record containing a foreign key identifying the list to which the task belongs.
- The One-to-Many Relation: One-to-Many relations are used when one entity can be associated with multiple entities. For instance, a list has many tasks, therefore a list *has many* tasks, and one task *belongs to* a list.
- The Many-to-Many Relation: Many-to-Many relations are used when one record can be related to multiple other records, and vice versa. For instance, if we were to expand TODOParrot to include an online store selling self-help books, a book could be assigned to many categories, and each category could be associated with many books.
- The Has Many Through Relation: The Has Many Through relation is useful when you want to interact with a table through an intermediate relation. Suppose TODOParrot associated users with a country, and you wanted to display all public lists according to country. Logically the user's country ID is stored in the `users` table, meaning it's not possible to know whether a particular list is associated with a user living in Japan without also examining the associated user. You can simplify the process used to perform this sort of analysis using the Has Many Through relation.
- The Polymorphic Relation: Polymorphic relations are incredibly useful when you want a *model* (as opposed to a record) to belong to more than one other model. Perhaps the most illustrative example of polymorphic relation's utility involves wishing to associate comments with multiple different application models (TODO lists, blog entries, and products, for instance). It would be inefficient to create separate comment-oriented models for maintaining comments associated with these different models, and so you can instead use a polymorphic relation to relate a single comment model to as many other models as you please without sacrificing capabilities. If you're not familiar with polymorphic relations then I'd imagine this sounds a bit like magic however I promise it will soon all make sense.

Introducing One-to-One Relations

One-to-one relationships link one row in a database table to one (and only one) row in another table. In my opinion there are generally few uses for a one-to-one relationship because the very nature of the relationship indicates the data could be consolidated within a single record. However, for the sake of demonstration let's suppose your application offered user authentication and profile management, and you wanted to separate the user's authentication (e-mail address, password) and profile (name, phone number, gender) data into two separate tables. This relationship is depicted in the below diagram.
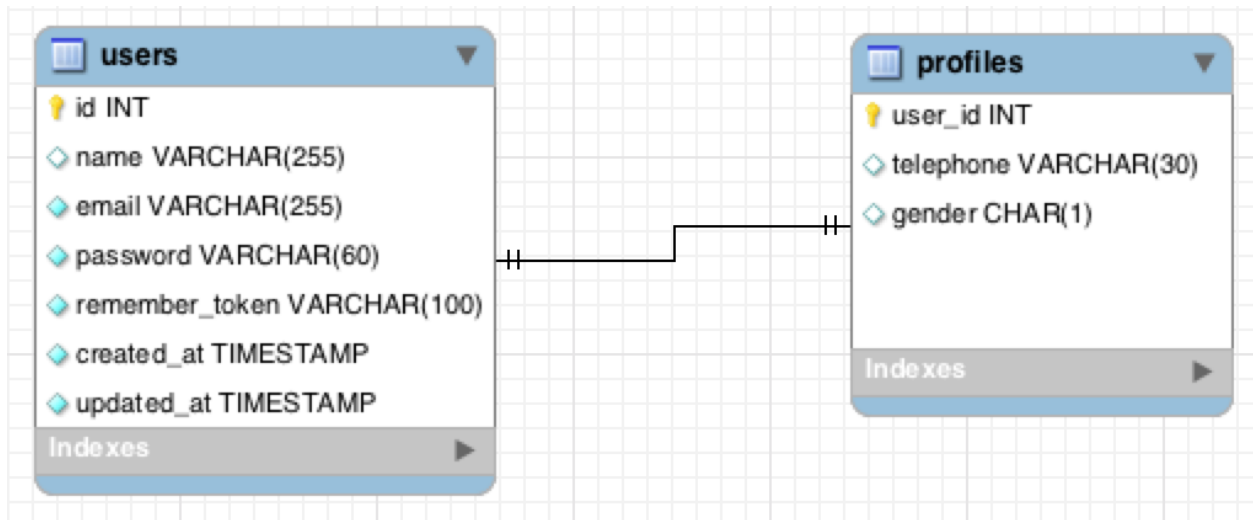
An example one-to-one relationship

To manage this relationship in Laravel you'll associate the `User` model (created automatically with every new Laravel 5 project; I'll formally introduce this model in Chapter 7) with the model responsible for managing the profiles, which we'll call `Profile`. To create the model you can use the Artisan generator:

```
1  $ php artisan make:model Profile -m
```

You'll find the newly generated model inside `app/Profile.php`:

```
1  use Illuminate\Database\Eloquent\Model;
2
3  class Profile extends Model {
4
5      //
6
7  }
```

Laravel will also generate a migration for the model's corresponding database table (`profiles`). Open up the newly created migration (inside `database/migrations`) and modify the `up` method to look like this:

```
 1  public function up()
 2  {
 3    Schema::create('profiles', function(Blueprint $table)
 4    {
 5      $table->increments('id');
 6      $table->integer('user_id')->unsigned()->nullable();
 7      $table->foreign('user_id')->references('id')->on('users');
 8      $table->string('name');
 9      $table->string('telephone');
10      $table->timestamps();
11    });
12  }
```

The bolded lines are the only four you'll need to add. The first line results in the addition of an integer-based column named user_id. The second line identifies this column as being a foreign key which references the users table's id column.

You must specify an integer column as unsigned when it's intended to be used as a foreign key, otherwise the migration will fail.

After saving the changes run the following command to create the table:

```
 1  $ php artisan migrate
 2  Migrated: 2015_01_20_201647_create_profiles_table
```

With the tables in place it's time to formally define the relations within the Laravel application.

Defining the One-to-One Relation

You'll define a one-to-one relation by creating a public method typically having the same name as the related model. The method will return the value of the hasOne method, as demonstrated below:

```
1  class User extends Model {
2
3    public function profile()
4    {
5      return $this->hasOne('todoparrot\Profile');
6    }
7
8  }
```

Once defined, you can retrieve a user's profile information by calling the user's `profile` method. Because the relations can be chained, you could for instance retrieve a user's telephone number like this:

```
1  $user = User::find(212)->profile->telephone;
```

To retrieve the telephone number, Laravel will look for a foreign key in the `profiles` table named `user_id`, matching the ID stored in that column with the user's ID.

The above example demonstrates how to traverse a relation, but how is a relation created in the first place? I'll show you how to do this next.

Creating a One-to-One Relation

You can easily create a One-to-One relation by creating the child object and then saving it through the parent object, as demonstrated in the below example:

```
1  $profile = new Profile;
2  $profile->telephone = '614-867-5309';
3
4  $user = User::find(212);
5  $user->profile()->save($profile);
```

Deleting a One-to-One Relation

Because a profile should not exist without a corresponding user, you'll just delete the associated profile record in the case you want to end the relationship:

```
1  $user = User::find(212);
2  $user->profile()->delete();
```

However, if a user record were to be deleted from the database you wouldn't want its corresponding profile record to be orphaned. One way to avoid this is by deleting the related profile record after deleting the user record (via Eloquent's `delete` method), but chances are this two step process will eventually be neglected, leaving orphaned records strewn about the database. Instead, you'll probably want to automate this process by taking advantage of the underlying database's ability to delete child tables when the parent table is deleted. You can specify this requirement when defining the foreign key in your table migration. I've modified the relevant lines of the earlier migration used to create the `profiles` table, attaching the `onDelete` option to the foreign key:

```
1  $table->integer('user_id')->unsigned();
2  $table->foreign('user_id')->references('id')
3         ->on('users')->onDelete('cascade');
```

With the cascading delete option in place, deleting a user from the database will automatically result in the deletion of the user's corresponding profile.

Introducing the Belongs To Relation

Using the `hasOne` relation demonstrated in the `User` model as demonstrated above, it's possible to retrieve a profile attribute via a user, such as a phone number, but *not* possible to retrieve a user via a given profile. This is because the `hasOne` relation is a one-way definition. You can make the relation bidirectional by defining a `belongsTo` relation in the `Profile` model, as demonstrated here:

```
1  class Profile extends Model {
2
3      public function user()
4      {
5          return $this->belongsTo('todoparrot\User');
6      }
7
8  }
```

Because the `profiles` table contains a foreign key representing the user (via the `user_id` column), each record found in `profiles` "belongs to" a record found in the `users` table. Once defined, you could retrieve a profile's associated user e-mail address based on the profile's telephone number like so:

```
1  $email = Profile::where('telephone', '614-867-5309')
2    ->get()->first()->user->email;
```

The Belongs To association certainly isn't limited to use in conjunction with One-to-One. Throughout the remainder of this chapter we'll use it in conjunction with several other relations.

Introducing One-to-Many Relations

The One-to-Many (also known as the Has Many) relationship is useful when you want to relate one table record to one or many other table records. The One-to-Many relation is used throughout TODOParrot, so in this section we'll look at some actual code used to power the application. To recap from the chapter introduction, the One-to-Many relation is used when you want to relate a single table record to multiple table records. For instance a list can have multiple tasks, therefore one list is related to many tasks, meaning we'll need to relate the `Todolist` and `Task` models using a One-to-Many relation.

Creating the Task Model

In the last chapter we created the `Todolist` model, meaning we'll need to create the `Task` model in order to begin associating tasks with lists. Use Artisan to generate the `Task` model:

```
1  $ php artisan make:model Task -m
```

You'll find the newly generated model inside `app/Task.php`:

```
1  use Illuminate\Database\Eloquent\Model;
2
3  class Task extends Model {
4
5      //
6
7  }
```

Open the newly created corresponding migration file (found in `database/migrations`) and modify it to look like this:

```
1  public function up()
2  {
3    Schema::create('tasks', function(Blueprint $table)
4    {
5      $table->increments('id');
6      $table->integer('todolist_id')->unsigned();
7      $table->foreign('todolist_id')
8        ->references('id')->on('todolists')
9        ->onDelete('cascade');
10     $table->string('name');
11     $table->text('description');
```

```
12      $table->boolean('done')->default(false);
13      $table->timestamps();
14    });
15  }
16
17  public function down()
18  {
19    Schema::drop('tasks');
20  }
```

Notice we're including an integer-based column named `todolist_id` in the `tasks` table, followed by a specification that this column be defined as a foreign key. In doing so, Laravel will ensure that the column is indexed, and additionally you'll optionally be able to determine what happens to these records should the parent be updated or deleted (more about this latter matter in a moment). After saving the changes, run the migration:

```
1  $ php artisan migrate
2  Migrated: 2014_10_30_164456_create_tasks_table
```

Defining the One-to-Many Relation

With the `Task` model and underlying table created, it's time to create the relation. Open the `Todolist` model and create a public method named `tasks`, inside it referencing the `hasMany` method:

```
1  class Todolist extends Model {
2
3    ...
4
5    public function tasks()
6    {
7      return $this->hasMany('todoparrot\Task');
8    }
9
10  }
```

You'll likely also want to define the opposite side of the relation within the `Task` model using the `belongsTo` method:

```
 1  class Task extends Model {
 2
 3    ...
 4
 5    public function todolist()
 6    {
 7      return $this->belongsTo('todoparrot\Todolist');
 8    }
 9
10  }
```

If you don't understand why we'd want to use the `belongsTo` relation here, please refer back to the earlier section, "Introducing the Belongs To Relation".

With the relation defined, let's next review how to associate tasks with a list.

Associating Tasks with a TODO List

To assign a task to a list, you'll first create a new `Task` object and then save it through the `Todolist` object, as demonstrated here:

```
 1  $list = Todolist::find(245);
 2
 3  $task = new Task;
 4  $task->name = 'Walk the dog';
 5  $task->description = 'Walk Barky the Mutt';
 6
 7  $list->tasks()->save($task);
 8
 9  $task = new Task;
10  $task->name = 'Make tacos for dinner';
11  $task->description = 'Mexican sounds really yummy!';
12
13  $list->tasks()->save($task);
```

With two tasks saved, you can now iterate over the list's tasks within a view like you would any other collection. Let's modify the `Lists` controller's `show` action/view (created in the last chapter) to additionally display list tasks. The `Lists` controller's `show` action doesn't actually change at all, but I'll include it here anyway for easy reference:

```
1  public function show($id)
2  {
3    $list = Todolist::find($id);
4    return view('lists.show')->with('list', $list);
5  }
```

We'll only need to update the view (`resources/views/lists/show.blade.php`) to iterate over the tasks. I'll present the modified view here:

```
1  @extends('layouts.master')
2
3  @section('content')
4
5  <h1>{{ $list->name }}</h1>
6
7  <p>
8  Created on: {{ $list->created_at }}
9  Last modified: {{ $list->updated_at }}<br />
10 </p>
11
12 <p>
13 {{ $list->description }}
14 </p>
15
16 <h2>Tasks</h2>
17
18 @if ($list->tasks->count() > 0)
19   <ul>
20   @foreach ($list->tasks as $task)
21
22     <li>{{ $task->name }}</li>
23
24   @endforeach
25   </ul>
26 @else
27  <p>
28   You haven't created any tasks.
29   <a href="{{ URL::route('lists.tasks.create', [$list->id]) }}"
30      class='btn btn-primary'>Create a task</a>
31 </p>
32 @endif
33
34 @endsection
```

Filtering Related Records

You'll often wish to retrieve a filtered collection of related records. For instance the user might desire to only see a list of incomplete list tasks. You can do so by filtering on the `tasks` table's `done` column:

```
1  $completedTasks = Todolist::find(1)->tasks()->where('done', true)->get();
```

Introducing Many-to-Many Relations

You'll use the many-to-many relation when the need arises to relate a record in one table to one or several records in another table, and vice versa. Consider some future version of TODOParrot that allowed users to classify lists using one or more categories, such as "leisure", "exercise", "work", "vacation", and "cooking". A list titled "San Juan Vacation" might be associated with several categories such as "leisure" and "vacation", and the "leisure" category would likely be associated with more than one list, meaning a list can be associated with many categories, and a category can be associated with many lists. See the below diagram for an illustrative example of this relation.
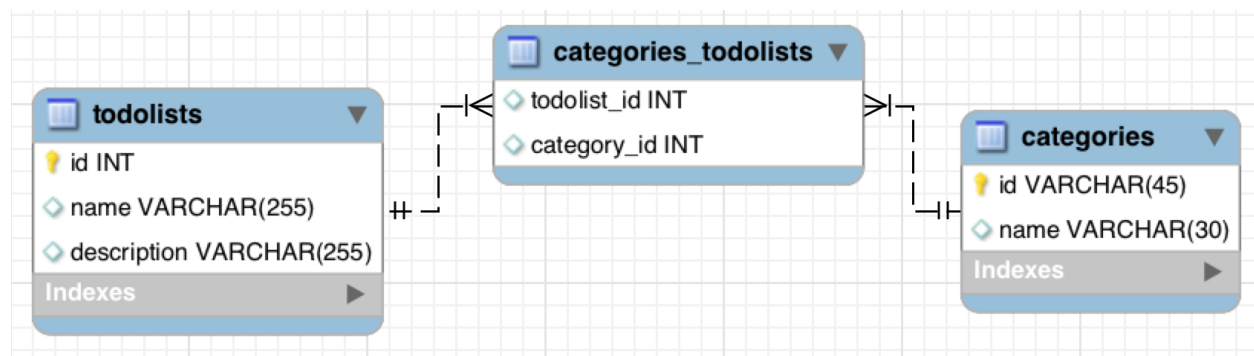
An example many-to-many relationship

In this section you'll learn how to create the intermediary table used to manage the relation (known as a *pivot table*), define the relation within the respective models, and manage the relation data.

Creating the Pivot Table

Many-to-many relations require an intermediary table to manage the relation. The simplest implementation of the intermediary table, known as a *pivot table*, would consist of just two columns for storing the foreign keys pointing to each related pair of records. Laravel presumes the pivot table is named by concatenating the two related model names together with an underscore separating the names. The names should appear in alphabetical order. Therefore if we were creating a many-to-many relationship between the `Todolist` and `Category` models, the pivot table name would be `category_todolist`. Of course, the `Category` model and corresponding `categories` table also needs to exist, so let's begin by generating the model:

```
1  $ php artisan make:model Category -m
```

You'll find the newly generated model inside `app/Category.php`:

```
1  use Illuminate\Database\Eloquent\Model;
2
3  class Category extends Model {
4
5    protected $fillable = [];
6
7  }
```

Next, modify the newly created migration file's `up()` method to look like this:

```
1  public function up()
2  {
3    Schema::create('categories', function(Blueprint $table)
4    {
5      $table->increments('id');
6      $table->string('name');
7      $table->timestamps();
8    });
9  }
```

Finally, run Artisan's `migrate` command to create the table:

```
1  $ php artisan migrate
```

With the `Category` model and corresponding `categories` table created, let's next create the `category_todolist` table:

```
1  $ php artisan make:migration --create=category_todolist create_category_todolist\
2  _table
```

Next, open up the newly created migration (`database/migrations/`) and modify the `up` method to look like this:

```
1  public function up()
2  {
3    Schema::create('category_todolist', function(Blueprint $table)
4    {
5        $table->integer('category_id')->unsigned()->nullable();
6        $table->foreign('category_id')->references('id')
7              ->on('categories')->onDelete('cascade');
8
9        $table->integer('todolist_id')->unsigned()->nullable();
10       $table->foreign('todolist_id')->references('id')
11             ->on('todolists')->onDelete('cascade');
12
13       $table->timestamps();
14   });
15 }
```

After saving the changes run Artisan's `migrate` command to create the table:

```
1  $ php artisan migrate
```

Defining the Many-to-Many Relation

With the tables in place it's time to define the many-to-many relation within the respective models. Open the `Todolist` model and add the following method to the class:

```
1  public function categories()
2  {
3      return $this->belongsToMany('todoparrot\Category')
4        ->withTimestamps();
5  }
```

Notice I've chained the `withTimestamps` method to the return statement. This instructs Laravel to additionally update the `category_todolist` timestamps when saving a new record. If you choose to omit the `created_at` and `updated_at` timestamps from this pivot table (done by removing the call to $table->timestamps from the migration), you can omit the `withTimestamps` method).

Save the changes and then open the `Category` model, adding the following method to the class:

```
1  public function todolists()
2  {
3      return $this->belongsToMany('todoparrot\Todolist')
4          ->withTimestamps();
5  }
```

After saving these changes you're ready to begin using the relation!

Associating Records Using the Many-to-Many Relation

You can associate records using the many-to-many relation in the same way as was demonstrated for one-to-many relations; just traverse the relation and use the save method, as demonstrated here:

```
1  $tl = Todolist::find(1);
2
3  $category = new Category(array('name' => 'Vacation'));
4
5  $tl->categories()->save($category);
```

> In order for this particular example to work you'll need to make sure name has been added to the Category model's fillable property.

After executing this code you'll see the new category has been created and the association between this newly created category and the list has been made:

```
 1  mysql> select * from categories;
 2  +----+----------+---------------------+---------------------+
 3  | id | name     | created_at          | updated_at          |
 4  +----+----------+---------------------+---------------------+
 5  |  1 | Vacation | 2014-11-03 20:44:11 | 2014-11-03 20:44:11 |
 6  +----+----------+---------------------+---------------------+
 7
 8  mysql> select * from category_todolist;
 9  +-------------+-------------+------------+------------+
10  | category_id | todolist_id | created_at | updated_at |
11  +-------------+-------------+------------+------------+
12  |           1 |           1 | ...        | ...        |
13  +-------------+-------------+------------+------------+
```

The above example involves the creation of a new category. You can easily associate an existing category with a list using similar syntax:

```
1  $list = Todolist::find(2);
2
3  $category = Category::find(1)
4
5  $list->categories()->save($category);
```

You can alternatively use the `attach` and `detach` methods to associate and disassociate related records. For instance to both associate and immediately persist a new relationship between a list and category, you can either pass in the `Category` object or its primary key into `attach`. Both variations are demonstrated here:

```
1  $list = Todolist::find(2);
2
3  $category = Category::find(1)
4
5  // In this example we're passing in a Category object
6  $list->categories()->attach($category);
7
8  // The number 5 is the primary key of another category
9  $list->categories()->attach(5);
```

You can also pass an array of IDs into `attach`:

```
1  $list->categories()->attach([3,4]);
```

To disassociate a category from a list, you can use `detach`, passing along either the `Category` object, an object's primary key, or an array of primary keys:

```
1  // Pass the Category object into the detach method
2  $list->categories()->detach(Category::find(3));
3
4  // Pass a category's ID
5  $list->categories()->detach(3);
6
7  // Pass along an array of category IDs
8  $list->categories()->detach([3,4]);
```

Determining if a Relation Already Exists

Laravel will not prevent you from duplicating an association, meaning the following code will result in a list being associated with the same category twice:

```
1  $list = Todolist::find(2);
2
3  $category = Category::find(1)
4
5  $list->categories()->save($category);
6  $list->categories()->save($category);
```

If you have a look at the database you'll see that the `Todolist` record associated with the primary key 2 has been twice related to the `Category` record associated with the primary key 1, which is surely not the desired behavior:

```
1  mysql> select * from category_todolist;
2  +-------------+-------------+------------+------------+
3  | category_id | todolist_id | created_at | updated_at |
4  +-------------+-------------+------------+------------+
5  |           1 |           2 | ...        | ...        |
6  |           1 |           2 | ...        | ...        |
7  +-------------+-------------+------------+------------+
```

You can avoid this by first determining whether the relation already exists using the `contains` method:

```
1  $list = Todolist::find(2);
2
3  $category = Category::find(1)
4
5  if ($list->categories->contains($category))
6  {
7
8    return Redirect::route('lists.show', array($list->id))
9      ->with('message', 'Category could not be assigned. Duplicate entry!');
10
11  } else {
12
13    $list->categories()->save($category);
14
15    return Redirect::route('lists.show', array($list->id))
16      ->with('message', 'The category has been assigned!');
17
18  }
```

Saving Multiple Relations Simultaneously

You can use the `saveMany` method to save multiple relations at the same time:

```
1   $list = Todolist::find(1);
2
3   $categories = [
4     new Category(array('name' => 'Vacation')),
5     new Category(array('name' => 'Tropical')),
6     new Category(array('name' => 'Leisure')),
7   ];
8
9   $list->categories()->saveMany($categories);
```

Traversing the Many-to-Many Relation

You'll traverse a many-to-many relation in the same fashion as described for the one-to-many relation; just iterate over the collection:

```
1   $list = Todolist::find(2);
2
3   ...
4
5   @if ($list->categories->count() > 0)
6
7     <ul>
8
9     @foreach($language->lists->categories as $category)
10
11      <li>{{ $category->name }}</li>
12
13    @endforeach
14
15    </ul>
16
17  @endif
```

Because the relation is defined on each side, you're not limited to traversing a list's categories! You can also traverse a category's lists:

```
1   $category = Category::find(2);
2
3   ...
4
5   @if ($category->lists->count() > 0)
6
7     <ul>
8
9       @foreach($category->lists as $list)
10
11        <li>{{ $list->name }}</li>
12
13      @endforeach
14
15    </ul>
16
17  @endif
```

Synchronizing Many-to-Many Relations

Suppose you provide users with a multiple selection box that allows users to easily associate a list with one or more categories. Because the user can both select and deselect categories, you must take care to ensure that not only are the selected categories associated with the list, but also that any *deselected* categories are disassociated with the list. This task is a tad more daunting than it may at first seem. Fortunately, Laravel offers a method named sync which you can use to synchronize an array of primary keys with those already found in the database. For instance, suppose categories associated with the IDs 7, 12, 52, and 77 were passed into the action where you'd like to synchronize the list and categories. You can pass the IDs into sync as an array like this:

```
1   $categories = [7, 12, 52, 77];
2
3   $tl = Todolist::find(2);
4
5   $tl->categories()->sync($categories);
```

Once executed, the Todolist record identified by the primary key 2 will be associated *only* with the categories identified by the primary keys 7, 12, 52, and 77, even if prior to execution the Todolist record was additionally associated with other categories.

Managing Additional Many-to-Many Attributes

Thus far the many-to-many examples presented in this chapter have been concerned with a join table consisting of two foreign keys and optionally the created_at and updated_at timestamps. But what

if you wanted to manage additional attributes within this table, such as why a list's category was chosen? I realize this is perhaps a contrived example since the necessity of including such a reason seems to be a bit overkill, but cut me some slack since I'm making this up as I go along.

Believe it or not adding other attributes is as simple as including them in the table schema. For instance let's create a migration that adds a column named `description` to the `category_todolist` table created earlier in this section:

```
1  $ php artisan make:migration add_description_to_category_todolist_table
2  Created Migration: 2015_01_20_221931_add_description_to_category_todolist_table
```

Next, open up the newly generated migration file and modify the `up()` and `down()` methods to look like this:

```
1  public function up()
2  {
3    Schema::table('category_todolist', function($table)
4    {
5        $table->string('description');
6    });
7  }
8
9  public function down()
10 {
11   Schema::table('category_todolist', function($table)
12   {
13       $table->dropColumn('description');
14   });
15 }
```

Save the changes and After generating the migration be sure to migrate the change into the database:

```
1  $ php artisan migrate
2  Created Migration: 2015_01_20_221931_add_description_to_category_todolist_table
```

With the additional column in place all you'll need to do is adjust the syntax used to relate categories with the list. You'll pass along the category's ID along with the description key and desired value, as demonstrated here:

```
1  $list = Todolist::find(2);
2  $list->categories()->attach(
3    [3 => ['description' => 'Because San Juan is a tropical island']]
4  );
```

If you later wished to update an attribute associated with an existing record, you can use the `updateExistingPivot` method, passing along the category's foreign key along with an array containing the attribute you'd like to update along with its new value:

```
1  $list->categories()->updateExistingPivot(3,
2    ['description' => 'Sun, beaches and rum!']
3  );
```

Introducing Has Many Through Relations

Suppose TODOParrot's CEO has just returned from the "Mo Big Data Mo Money" conference, flush with ideas regarding how user data can be exploited and sold to advertisers. To kick things off he's asked you to create a new feature that summarizes the numbers of lists created according to country. You recently integrated a country of residence field into the user registration form (which means each user belongs to a country, and each country conceivably has many users), so you can tally up users according to country. To quickly recap this means the user/country relations would look like this:

```
1  class User extends Model {
2
3    public function country()
4    {
5      return $this->belongsTo('todoparrot\Country');
6    }
7
8  }
9
10 class Country extends Model {
11
12   public function users()
13   {
14       return $this->hasMany('todoparrot\User');
15   }
16
17 }
```

Because the `users` table that contains the foreign key reference to the `countries` table's ID, and not the user's lists, how can you relate lists with countries? The SQL query used to mine this sort of data is pretty elementary:

```
1  SELECT count(todolists.id), countries.name FROM todolists
2    LEFT JOIN users on users.id = todolists.user_id
3    LEFT JOIN countries ON countries.id = users.country_id
4    GROUP BY countries.name;
```

But how might you implement such a feature within your Laravel application? Enter the Has Many Through relation. The Has Many Through relation allows you to create a shortcut for querying data available through distantly related tables. This is actually incredibly easy to implement; just add the following relation to the `Country` model:

```
1  public function lists()
2  {
3      return $this->hasManyThrough('todoparrot\Todolist', 'todoparrot\User');
4  }
```

This relation gives the `Country` model the ability to access the `Todolist` model *through* the `User` model. After saving the model, you'll be able to for instance iterate over all lists created by user's residing in Italy:

```
1  $country = Country::where('name', 'Italy')->get()->first();
2
3  ...
4
5  <ul>
6    @foreach($country->lists as $list) {
7      <li>{{$list->name}}</li>
8    @endforeach
9  </ul>
```

Introducing Polymorphic Relations

When considering an interface for commenting on different types of application data (products and blog posts, for example), one might presume it is necessary to manage each type of comment separately. This approach would however be repetitive because each comment model would presumably consist of the same data structure. You can eliminate this repetition using a *polymorphic relation*, resulting in all comments being managed via a single model.

Let's work through an example that would use polymorphic relations to add commenting capabilities to the `User` and `Todolist` models. Begin by creating a new model named `Comment`:

```
1   $ php artisan make:model Comment -m
```

You'll find the newly generated model inside `app/Comment.php`:

```
1   use Illuminate\Database\Eloquent\Model;
2
3   class Comment extends Model {
4
5       //
6
7   }
```

Next, open up the newly generated migration file and modify the `up()` method to look like this:

```
1   Schema::create('comments', function(Blueprint $table)
2   {
3     $table->increments('id');
4     $table->text('body');
5     $table->integer('commentable_id');
6     $table->string('commentable_type');
7     $table->timestamps();
8   });
```

Finally, save the changes and run the migration:

```
1   $ php artisan migrate
2   Migrated: 2015_01_20_223902_create_comments_table
```

Because the `Comment` model serves as a central repository for comments associated with multiple different models, we require a means for knowing both which model and which record ID is associated with a particular comment. The `commentable_type` and `commentable_id` fields serve this purpose. For instance, if a comment is associated with a list, and the list record associated with the comment has a primary key of 453, then the comment's `commentable_type` field will be set to `Todolist` and the `commentable_id` to 453.

Logically you'll want to attach other fields to the `comments` table if you plan on for instance assigning ownership to comments via the `User` model, or would like to include a title for each comment.

Next, open the `Comment` model and add the following method:

```
1   class Comment extends Model {
2
3       public function commentable()
4       {
5           return $this->morphTo();
6       }
7
8   }
```

The morphTo method defines a polymorphic relationship. Personally I find the name to be a poor choice; when you read it just think "belongs To" but for polymorphic relationships, since the record will belong to whatever model is defined in the commentable_type field. This defines just one side of the relationship; you'll also want to define the inverse relation within any model that will be commentable, creating a method that determines which model is used to maintain the comments, *and* referencing the name of the method used in the polymorphic model:

```
1   class Todolist extends Model {
2
3     public function comments()
4     {
5         return $this->morphMany('\todoparrot\Comment', 'commentable');
6     }
7
8   }
```

With these two methods in place, it's time to begin using the polymorphic relation! The syntax for adding, removing and retrieving comments is straightforward; in the following example we'll attach a new comment to a list:

```
1   $list = Todolist::find(1);
2
3   $c = new Comment();
4
5   $c->body = 'Great work!';
6
7   $list->comments()->save($c);
```

After saving the comment, review the database and you'll see a record that looks like the following:

```
1   mysql> select * from comments;
2   +----+------------+---------------+--------------------+------------+-------\
3   ----+
4   | id | body       | commentable_id | commentable_type  | created_at | updated\
5   _at |
6   +----+------------+---------------+--------------------+------------+-------\
7   ----+
8   |  1 | Great work! |              1 | todoparrot\Todolist | 2015-...   | 2015-..\
9   .   |
10  +----+------------+---------------+--------------------+------------+-------\
11  ----+
```

The list's comments are just a collection, so you can easily iterate over it. You'll retrieve the list within the controller per usual:

```
1   public function index()
2   {
3     $list = Todolist::find(1);
4     return view('lists.show')->with('list', $list);
5   }
```

In the corresponding view you'll iterate over the comments collection:

```
1     @foreach ($list->comments as $comment)
2       <p>
3         {{ $comment->body }}
4       </p>
5     @endforeach
```

To delete a comment you can of course just delete the comment using its primary key.

Eager Loading

There's a matter known as the "N + 1 Queries" problem that has long confused web developers to the detriment of their application's performance. To understand the nature of the issue, consider the following seemingly innocent query:

```
1   $users = User::take(5)->get();
```

In the corresponding application view you then iterate over the retrieved locations like so:

```
1  <ul>
2    @foreach($users as $user)
3      <li>{{ $user->first_name }}: {{ $user->state->name }}</li>
4    @endforeach
5  </ul>
```

Pretty innocent bit of code, right? It certainly seems so until you realize these two snippets result in the execution of 6 distinct queries! Thus the name "N + 1", because we're executing one query to retrieve the ten locations, and then 5 queries to retrieve the name of each user's state name! In situations where you know you're going to need to access a relation's attribute you can use the `with` method to inform Laravel of your intent to subsequently access this relation and therefore preload the data:

```
1  $users = User::with('state')->take(5)->get();
```

When you subsequently access a `User` object's state name, the data will be immediately available because each user's state-related data was preloaded along with the original query!

Introducing Scopes

Applying conditions to queries gives you to power to retrieve and present filtered data in every imaginable manner. Some of these conditions will be used more than others, and Laravel provides a solution for cleanly packaging these conditions into easily readable and reusable statements. Consider a filter that only retrieves completed list tasks. You could use the following `where` condition to retrieve those tasks:

```
1  $completedTasks = Task::where('done', true)->get();
```

You might however wish to use a query such as this at multiple locations throughout an application. If so, you can DRY the code up a bit by instead using a *scope*. A scope is just a convenience method you can add to your model which encapsulates the syntax used to execute a query such as the above. Scopes are defined by prefixing the name of a method with `scope`, as demonstrated here:

```
1   class Task extends Model
2   {
3
4       public function scopeDone($query)
5       {
6           return $query->where('done', 1);
7       }
8
9   }
```

With the scope defined, you can execute it like so:

```
1   $completedTasks = Task::done()->get();
```

Creating Dynamic Scopes

If you wanted to create a scope capable of returning both completed and incomplete tasks based on a supplied argument, just define an input parameter like you would any model method:

```
1   class Task extends Model {
2
3       public function scopeDone($query, $flag)
4       {
5           return $query->where('done', $flag);
6       }
7
8   }
```

With the input parameter defined, you can use the scope like this:

```
1   // Get completed tasks
2   $completedTasks = Task::done(true)->get();
3
4   // Get incomplete tasks
5   $incompleteTasks = Task::done(false)->get();
```

Using Scopes with Relations

You'll often want to use scopes in conjunction with relations. For instance, you can retrieve a list of tasks associated with a specific list:

```
1  $list = Todolist::find(34);
2  $completedTasks = $list->tasks()->done(true)->get();
```

Summary

I'd imagine this to be the most difficult chapter in the book, primarily because you not only have to understand the syntax used to manage and traverse relations but also be able to visualize at a conceptual level the different ways in which your project data should be structured. Although it's a tall order, once you do have a solid grasp on the topics presented in this chapter there really will be no limit in terms of your ability to build complex database-driven Laravel projects! As always if you don't understand any topic discussed in this chapter, or would like to offer some input regarding how any of the material can be improved, be sure to e-mail me at wj@wjgilmore.com.

Chapter 5. Integrating Web Forms

Chances are you're going to spend quite a bit of time building Laravel applications that require various forms and models to work together in a seamless fashion. For instance TODOParrot users rely on a series of forms for managing manage lists and list items, as well as to get in touch with the administrators should they be experiencing a problem or otherwise wish to provide feedback[109]. While creating the HTML used to display these forms is easy enough, integrating them to work with your models and other parts of the Laravel application can quickly become confusing. Never fear though because in this chapter I'll show you how to wield total control over your Laravel forms, covering a variety of form integration scenarios you're sure to encounter when embedding forms into your future applications.

Web Form Fundamentals

While all readers are familiar with web forms from the user's perspective, I'd imagine at least a few of you could benefit from a quick introduction to a few technical aspects of forms development. If you're a knowledgeable web developer with plenty of experience working with web forms, then by all means skip ahead to the next section.

The following example incorporates HTML5-specific markup[110], PHP-specific security features, and Bootstrap-specific markup[111] to produce a flexible, secure, and responsive form. Take a moment to examine the form markup before moving on to the summary found below.

```
1  <form method="POST" action="http://todoparrot.com/contact"
2    accept-charset="UTF-8" class="form">
3
4  <input name="_token" type="hidden" value="YLMaxbvKETQ4Tz6zVuWhd6XblhatvPtVVboSJv\
5  Hh">
6
7  <div class="form-group">
8    <label for="name">Your Name</label>
9    <input class="form-control" placeholder="Your name"
10     name="name" type="text" required="required">
11  </div>
12
```

---

[109]http://todoparrot.com/contact
[110]http://diveintohtml5.info/forms.html
[111]http://getbootstrap.com/

```
13  <div class="form-group">
14    <label for="email">Your E-mail Address</label>
15    <input class="form-control" placeholder="Your e-mail address"
16      name="email" type="text" required="required">
17  </div>
18
19  <div class="form-group">
20    <label for="message">Your Message</label>
21    <textarea class="form-control" name="message"
22      required="required"></textarea>
23  </div>
24
25  <div class="form-group">
26    <input class="btn btn-primary" type="submit" value="Contact Us!">
27  </div>
28  </form>
```

Let's review the pertinent characteristics of this form, beginning with the `form` enclosure. The `form` tag encloses the fields and other markup comprising the form:

```
1  <form method="POST" action="http://todoparrot.com/contact"
2    accept-charset="UTF-8" class="form">
3    ...
4  </form>
```

Let's review the relevant `form` attributes:

The `method` attribute defines *how* the data will be sent to the destination. You might be somewhat familiar with the most common methods `GET` and `POST`, but not understand the important difference between the two. The short answer is that you should use `GET` for "safe" (also known as *idempotent*) tasks which could conceivably be repeatedly executed without negative consequences, and `POST` for those deemed to be "unsafe" if executed more than once. For instance, you would typically use `GET` in conjunction with a search form, because search results could be cached, bookmarked and shared without negatively affecting the site nor the users. The `POST` method should be used for tasks that if executed more than once would pose a problem, such as sending a support request or charging a credit card. You've likely at one point or another mistakenly attempted to resend a `POST` form and been greeted with a message such the one presented by Firefox:
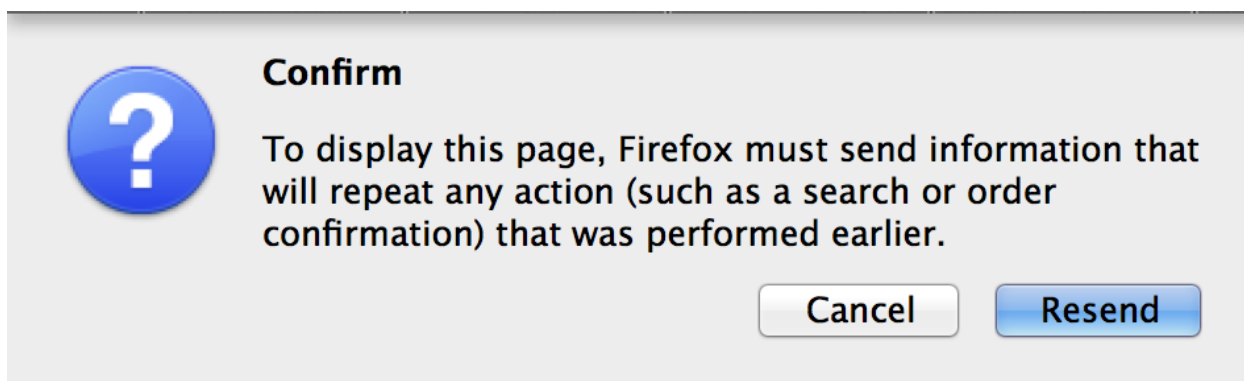
A Firefox POST warning

However, if you reload Google search results, you'll receive no such warning. This is because browser developers are aware of this important difference between GET and POST, and build in this warning as a cautionary step for users *should the website developer not have taken additional steps to prevent unwanted consequences* when a POST form be submitted more than once.

The accept-charset attribute defines the character encoding intended to be used in conjunction with the form, should you desire to override the default character set defined in the page header.

The action attribute defines where the user will be taken when the form is submitted. Logically this should match a route definition found in your routes.rb file. Thanks to Laravel's powerful route customization capabilities, there are some pretty nifty things you can do in regards to defining these endpoints, a topic I'll devote to a later section.

Next you'll find a hidden input element named _token:

```
1  <input name="_token" type="hidden" value="pEa4MGDfD2ESgIeeGxWxGmVmAfKEDdVKEP5ic5\
2  HT">
```

This field is a security feature that prevents cross-site request forgery (CSRF)[112] by storing the randomly generated value in the field and also in a session. When the form is submitted, that field's value is compared with the session stored on the client to ensure they match; lacking such a feature it would be possible for third parties to attempt to forge form input values and send them directly to the server, potentially altering or destroying application data.

Finally, the three input tags used to gather the user's name, e-mail address and message:

---

[112]http://en.wikipedia.org/wiki/Cross-site_request_forgery

```
 1  <label for="name">Your Name</label>
 2  <input class="form-control" placeholder="Your name"
 3    name="name" type="text" required="required">
 4
 5  <label for="email">Your E-mail Address</label>
 6  <input class="form-control" placeholder="Your e-mail address"
 7    name="email" type="text" required="required">
 8
 9  <label for="message">Your Message</label>
10  <textarea class="form-control" name="message"
11    required="required"></textarea>
```

There's not much to discuss regarding these fields other than to mention all three fields are identified as required. This is an HTML5 feature that when implemented by browsers will automatically provide client-side validation without additional work on the part of the developer. I recommend you use this feature *in addition to* server-side validation.

With this general overview complete, I'll spend the remainder of this chapter discussing Laravel's forms integration features, drawing upon numerous live examples found in the TODOParrot code to illustrate various concepts.

Creating a User Feedback Form

Let's kick things off by building a simple contact form consisting of three fields, including the user's name, email address, and message (see below figure).

Your Name

> Your name

Your E-mail Address

> Your e-mail address

Your Message

> Your message

Contact Us!

TODOParrot's contact form

Although we could create a `Contact` controller expressly for the purpose of displaying and processing the contact form, I prefer to manage this sort of feature in a controller that additionally handles other application-related administrative matters. In the case of TODOParrot this feature is managed by the `About` controller's `create` and `store` actions (`create` presents the form via the `GET` method and `store` processes it via `POST`). You're of course free to manage the contact feature within any controller you please, however because the `About` controller (which also houses the "About TODOParrot" page at http://todoparrot.com/about[113]) would otherwise not use the `create` and `store` actions I decided to consolidate the contact feature there. However I'm only using the RESTful `create` and `store` naming conventions for organizational purposes; the `About` controller only actually contains three actions (`index`, `create` and `store`) and doesn't concern itself with manipulation of a particular resource. Therefore in this case I suggest creating a "plain" controller using Artisan's `make:controller` command:

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

Next, to route users to the contact form using the convenient `/contact` shortcut you'll need to define two aliases in the `app/Http/routes.php` file:

---

[113]http://todoparrot.com/about

```
1   Route::get('contact',
2     ['as' => 'contact', 'uses' => 'AboutController@create']);
3   Route::post('contact',
4     ['as' => 'contact_store', 'uses' => 'AboutController@store']);
```

Next, you'll need to add the `create` and `store` actions to the newly created `About` controller, because when the `--plain` option is used in conjunction with Artisan's `make:controller` method an empty controller will be created. Modify this controller to look like this:

```php
1   <?php namespace todoparrot\Http\Controllers;
2
3   class AboutController extends Controller {
4
5       public function create()
6       {
7           return view('about.contact');
8       }
9
10      public function store()
11      {
12      }
13
14  }
```

The `create` action has been configured to serve a view named `contact.blade.php` found in the directory `resources/views/about`. However we haven't yet created this particular view so let's do so next.

Creating the Contact Form

Earlier in this chapter I showed you the *rendered* form HTML, introducing several key Laravel- and HTML5- related form features in the process. Note my emphasis on *rendered* because you won't actually hand-code the form! Instead, you'll use Laravel's fantastic form generation capabilities to manage this tedious task for you. Below I've pasted in the section of code found in TODOParrot's `resources/views/about/contact.blade.php` view that's responsible for generating the contact form:

```
 1  @extends('layouts.master')
 2
 3  @section('content')
 4
 5  <h1>Contact TODOParrot</h1>
 6
 7  <ul>
 8      @foreach($errors->all() as $error)
 9          <li>{{ $error }}</li>
10      @endforeach
11  </ul>
12
13  {!! Form::open(array('route' => 'contact_store', 'class' => 'form')) !!}
14
15  <div class="form-group">
16      {!! Form::label('Your Name') !!}
17      {!! Form::text('name', null,
18          array('required',
19                'class'=>'form-control',
20                'placeholder'=>'Your name')) !!}
21  </div>
22
23  <div class="form-group">
24      {!! Form::label('Your E-mail Address') !!}
25      {!! Form::text('email', null,
26          array('required',
27                'class'=>'form-control',
28                'placeholder'=>'Your e-mail address')) !!}
29  </div>
30
31  <div class="form-group">
32      {!! Form::label('Your Message') !!}
33      {!! Form::textarea('message', null,
34          array('required',
35                'class'=>'form-control',
36                'placeholder'=>'Your message')) !!}
37  </div>
38
39  <div class="form-group">
40      {!! Form::submit('Contact Us!',
41        array('class'=>'btn btn-primary')) !!}
42  </div>
```

```
43    {!! Form::close() !!}
44
45    @endsection
```

This form uses the form builder made available through Laravel's HTML package[114]. I explained how to install this package in Chapter 2 (the section "Integrating Images, CSS and JavaScript"), but at that point we just configured the HTML Facade. To take advantage of the form-specific tags you'll need to additionally add the following alias to the `config/app.php` `aliases` array:

```
1    'Form'=> Collective\Html\FormFacade::class
```

If this is your first encounter with the `Form::open` helper then I'd imagine this example looks rather scary. However once you build a few forms in this fashion I promise you'll wonder how you ever got along without it. Let's break down the key syntax used in this example:

```
1    {!! Form::open(array('route' => 'contact_store', 'class' => 'form')) !!}
2    ...
3    {!! Form::close() !!}
```

The `Form::open` and `Form::close()` methods work together to generate the form's opening and closing tags. The `Form::open` method accepts an array containing various settings such as the route alias which in this case points to the `About` controller's `store` method, and a class used to stylize the form. The default method is `POST` however you can easily override the method to instead use `GET` by passing `'method' => 'get'` into the array. Additionally, the `Form::open` method will ensure the aforementioned CSRF-prevention `_token` hidden field is added to the form.

Next up you'll see the following `@foreach` block:

```
1    <ul>
2        @foreach($errors->all() as $error)
3            <li>{{ $error }}</li>
4        @endforeach
5    </ul>
```

This block is used to output any validation errors should one or more of the user-supplied field values not pass the validation tests (more on this in a moment).

Next you'll see a series of methods used to generate the various form fields. This is a relatively simplistic form therefore only a few of the available field generation methods are used, including `Form::label` (for creating form field labels), `Form::text` (for creating form text fields), `Form::textarea` (for creating a form text area), and `Form::submit` (for creating a submit button).

---

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

Note how the `Form::text` and `Form::textarea` methods all accept as their first argument a model attribute name (`name`, `email`, and `message`, respectively). All of the methods also accept an assortment of other options, such as class names and HTML5 form attributes.

Once you add this code to your project's `resources/views/about/contact.blade.php` file, navigate to `/contact` and you should see the same form as that found at `http://todoparrot.com/contact`!

With the form created, we'll next need to create the logic used to process the form contents and send the feedback to the site administrator via e-mail.

Creating the Contact Form Request

Laravel 5 introduces a new feature known as a *form request*. This feature is intended to remove form authorization and validation logic from your controllers by encapsulating this logic in a separate class. TODOParrot uses form requests in conjunction with each form used throughout the site and I'm pleased to report this feature works meets its goal quite nicely.

To create a new form request you can use Artisan's `make:request` feature:

```
1  $ php artisan make:request ContactFormRequest
2  Request created successfully.
```

This created a file named `ContactFormRequest.php` that resides in the directory `app/Http/Requests/`. The class skeleton looks like this (comments removed):

```php
1  <?php namespace todoparrot\Http\Requests;
2
3  use todoparrot\Http\Requests\Request;
4
5  class ContactFormRequest extends Request {
6
7    public function authorize()
8    {
9      return false;
10   }
11
12   public function rules()
13   {
14     return [
15       //
16     ];
17   }
18
19 }
```

The `authorize` method determines whether the current user is authorized to interact with this form. I'll talk more about the purpose of this method in Chapter 7. Because we want any visitor to be able to use this form you should just modify the method to return `true` instead of `false`:

```php
public function authorize()
{
  return true;
}
```

The `rules` method defines the validation rules associated with the fields found in the form. The contact form has three fields, including `name`, `email`, and `message`. All three fields are required, and the `email` field must be a syntactically valid e-mail address, so you'll want to update the `rules` method to look like this:

```php
public function rules()
{
  return [
    'name' => 'required',
    'email' => 'required|email',
    'message' => 'required',
  ];
}
```

The `required` and `email` validators used in this example are just a few of the many available via Laravel's validation class. See Chapter 3 for more information about these rules. In the examples to come I'll provide additional examples demonstrating other available validators. Additionally, note how you can use multiple validators in conjunction with a form field by concatenating the validators together using a vertical bar (`|`).

After saving the changes to `ContactFormRequest.php` open the `About` controller and modify the `store` method to look like this:

```php
...

use todoparrot\Http\Requests\ContactFormRequest;

class AboutController extends Controller {

  public function store(ContactFormRequest $request)
  {

    return \Redirect::route('contact')
```

```
11            ->with('message', 'Thanks for contacting us!');
12
13      }
14
15  }
```

While we haven't yet added the e-mail delivery logic, believe it or not this action is otherwise
complete. This is because the `ContactForm` form request will handle the validation *and* display of
validation error messages should validation fail. For instance submitting the contact form without
completing any of the fields will result in three validation error found presented in the below
screenshot being displayed:

Contact TODOParrot

- The name field is required.
- The email field is required.
- The message field is required.

Your Name

Your name

Your E-mail Address

Your e-mail address

Displaying contact form validation errors

These errors won't appear out of thin air of course; they'll be displayed via the `$errors` array
included in the `contact.blade.php` view:

```
1  <ul>
2    @foreach($errors->all() as $error)
3      <li>{{ $error }}</li>
4    @endforeach
5  </ul>
```

You'll also want to inform the user of a successful form submission. To do so you can use a flash
message, which is populated in the `store` method ("Thanks for contacting us!"). The variable passed
into the `with` method is automatically added to the Laravel's flash data which can subsequently
be retrieved via the `Session::get` method. For instance you'll find the following snippet in
TODOParrot's `master.blade.php` so flash messages can be retrieved and displayed above any view:

```
1  @if(Session::has('message'))
2      <div class="alert alert-info">
3          {{ Session::get('message') }}
4      </div>
5  @endif
```

Only one step remains before the contact form is completely operational. We'll need to configure Laravel's mail component and integrate e-mail delivery functionality into the `store` method. Let's complete these steps next.

Configuring Laravel's Mail Component

Thanks to integration with the popular SwiftMailer[115] package, it's easy to send e-mail through your Laravel application. All you'll need to do is make a few changes to the `config/mail.php` configuration file. In this file you'll find a number of configuration settings:

- `driver`: Laravel supports several mail drivers, including `SMTP`, PHP's `mail` function, the Sendmail MTA, and the Mailgun[116] and Mandrill[117] e-mail delivery services. You'll set the `driver` setting to the desired driver, choosing from `smtp`, `mail`, `sendmail`, `mailgun`, and `mandrill`. You could also optionally set `driver` to `log` in order to send e-mails to your development log rather than bother with actually sending them out during the development process.
- `host`: The `host` setting is used to set the host address of your SMTP server should you be using the `smtp` driver.
- `port`: The `port` setting is used to set the port used by your SMTP server should you be using the `smtp` driver.
- `from`: If you'd like all outbound application e-mails to use the same sender e-mail and name, you can set them using the `from` and `address` settings defined in this array.
- `encryption`: The `encryption` setting specifies the encryption protocol used when sending e-mails.
- `username`: The `username` setting defines the SMTP account username should you be using the `smtp` driver.
- `password`: The `password` setting defines the SMTP account password should you be using the `smtp` driver.
- `sendmail`: The `sendmail` setting defines the server Sendmail path should you be using the `sendmail` driver.
- `pretend`: The `pretend` setting will cause Laravel to ignore the defined driver and instead send e-mail to your application log, a useful option while your application is still under development.

---

[115]http://swiftmailer.org/

[116]http://www.mailgun.com/

[117]https://mandrill.com/

Although you'll commonly find tutorials demonstrating how to use a Gmail account to send e-mail through a PHP application, I suggest against doing so. Using Gmail is convenient because these days pretty much every developer possesses at least one Gmail account, meaning the configuration instructions are likely to suit the majority of readers. In fact, an earlier version of this book demonstrated how to use Gmail in conjunction with the contact form.

Upon further reflection I've removed the Gmail specific material, simply because Google has made it increasingly difficult to use Gmail accounts in this manner due largely to security and spamming concerns. Instead, if you're looking for a quality I strongly suggest checking out Mandrill[118]. Mandrill is an e-mail delivery service run by the e-mail marketing service MailChimp[119]. You can use Mandrill to send up to 12,000 e-mails a month for free, and registration takes less than five minutes.

Once registered and signed into the Mandrill console, click on the `Settings` menu option and you'll be provided with the necessary mail delivery credentials which will need to be made available to your application. These include the SMTP host (`smtp.mandrillapp.com`), port (587), and your designated SMTP username and password.

In total you'll want to change the following `config/mail.php` settings:

- Change the `driver` setting to `smtp`. This is the default value.
- Change the `host` setting to `smtp.mandrillapp.com`.
- Change the `port` setting to `587`.
- Change the `encryption` setting to `tls`.
- Change the `from['address']` and `from['name']` to reflect the e-mail address and name of the sender.
- Change the `username` setting to the Mandrill API username.

Next, open the `config/services.php` file and update the `mandrill` array's `secret` key to point to your Mandrill API key.

Obviously you won't really want to embed these values directly into `mail.php` and `services.php`; Instead you'll create environment variables for each of these values, and then update `mail.php` and `services.php` to use those environment variables.

Save these changes, and then modify the `About` controller's `store` method to look like this:

---

[118]http://mandrill.com/
[119]http://mailchimp.com/

```php
 1    public function store(ContactFormRequest $request)
 2    {
 3
 4        \Mail::send('emails.contact',
 5            array(
 6                'name' => $request->get('name'),
 7                'email' => $request->get('email'),
 8                'user_message' => $request->get('message')
 9            ), function($message)
10        {
11            $message->from('jason@example.com');
12            $message->to('jason@example.com', 'Admin')
13                ->subject('TODOParrot Feedback');
14        });
15
16      return \Redirect::route('contact')
17        ->with('message', 'Thanks for contacting us!');
18
19    }
```

The `Mail::send` method is responsible for initiating delivery of the e-mail. It accepts three parameters. The first parameter defines the name of the view used for the e-mail body template. The second parameter contains an array of data which will be made available to the e-mail template. In this case, the desired data originated in the contact form and is now made available through the `$request` object. The third parameter is a closure that gives you the opportunity to define additional e-mail related options such as the sender, recipient, and subject. Be sure to check out the Laravel mail documentation[120] for a complete explanation of the `Mail::send` method's features.

> Notice I used the variable `user_message` to pass the contact form's message text into the view. This is because Laravel always passes a variable named `$message` into the e-mail views which is used for attachment-related matters, so you should take care to not override this variable otherwise seemingly mysterious errors will crop up.

Finally, you'll need to create the `contact` view which contains the email content. I suggest saving this file in `resources/views/emails`. Per the above example you'll need to name the file `contact.blade.php`. For the purposes of this example I created a very simple view that looks like this:

---

[120]http://laravel.com/docs/master/mail

```
1    You received a message from TODOParrot.com:
2
3    <p>
4    Name: {{ $name }}
5    </p>
6
7    <p>
8    {{ $email }}
9    </p>
10
11   <p>
12   {{ $user_message }}
13   </p>
```

HTML formatting is used because Laravel (unfortunately in my opinion) sends HTML-formatted e-mail by default. You can however override this default to instead send text-based e-mail. See the Laravel mail documentation[121] for more details.

After saving these changes, return to the contact form, submit valid data and an e-mail should soon arrive in the inbox associated with the e-mail address supplied via the to method!

> ⚠️ If you neglected my advice against using Gmail and are, experiencing problems, it could be because of a Gmail setting pertaining to third-party access to your account. Enable the "Less secure apps" setting at https://www.google.com/settings/security/lesssecureapps[122] to resolve the issue. Even if this resolves the issue, keep in mind you nonetheless should not use your Gmail account for production purposes.

Creating New TODO Lists

Now that you understand how to use form requests, let's next create the interface and logic used to add a new list to the database. Begin by creating a RESTful controller:

```
1    $ php artisan make:controller ListsController
2    Controller created successfully.
```

With the controller created we next need to inform Laravel that we'd like to declare the controller as RESTful. Open app/Http/routes.php and add the following line:

---

[121]http://laravel.com/docs/master/mail
[122]https://www.google.com/settings/security/lesssecureapps

```
1  Route::resource('lists', 'ListsController');
```

Save the changes to `routes.php` and then open the newly created `Lists` controller. You'll find seven actions (each representing one of the RESTful routes introduced in Chapter 3). For easy reference I've pasted in the newly created controller, leaving only the two actions (`create` and `show`) we'll use to add a new list:

```
1  <?php namespace todoparrot\Http\Controllers;
2
3  class ListsController extends Controller {
4
5    public function create()
6    {
7    }
8
9    public function store()
10   {
11   }
12
13 }
```

As a reminder, the `create` action is responsible for serving the form, and `store` is responsible for processing the submitted form data.

> ℹ️ BTW, don't actually delete the other actions because we'll use them later in the chapter.

With the controller created and routes defined, it's time to create the form.

Creating the TODO List Form

Before creating the form you'll first need to create the `List` controller's `create` view. Begin by creating a directory named `lists`, placing it in the directory `resources/views`. Inside this directory create a file named `create.blade.php` and add the following contents to it:

```
 1  @extends('layouts.master')
 2
 3  @section('content')
 4
 5  <h1>Create a New List</h1>
 6
 7  <ul>
 8      @foreach($errors->all() as $error)
 9          <li>{{ $error }}</li>
10      @endforeach
11  </ul>
12
13  {!! Form::open(array('route' => 'lists.store', 'class' => 'form')) !!}
14
15  <div class="form-group">
16      {!! Form::label('List Name') !!}
17      {!! Form::text('name', null,
18        array('required', 'class'=>'form-control',
19              'placeholder'=>'San Juan Vacation')) !!}
20  </div>
21
22  <div class="form-group">
23      {!! Form::label('List Description') !!}
24      {!! Form::textarea('description', null,
25        array('required', 'class'=>'form-control',
26              'placeholder'=>'Things to do before leaving for vacation')) !!}
27  </div>
28
29  <div class="form-group">
30      {!! Form::submit('Create List', array('class'=>'btn btn-primary')) !!}
31  </div>
32  {!! Form::close() !!}
33
34  @stop
```

Presuming you've reviewed the earlier section regarding the contact form, then most of the form syntax is familiar to you. After creating the form, you'll need to modify the List controller's create action to serve the view:

```
1  public function create()
2  {
3    return view('lists.create');
4  }
```

After saving the changes to the `List` controller, navigate to `/lists/create` and you should see the form presented in the below screenshot!

Create a New List

List Name

San Juan Vacation

List Description

Things to do before leaving for vacation

Create List

Creating a new TODO List

With the form in place and the `create` action updated, it's time to create the Form Request class used to validate the submitted form data.

Creating the List Form Request Class

In this section we'll create a form request that will be used to validate the form data. Begin by creating the form request class skeleton:

```
1  $ php artisan make:request ListFormRequest
2  Request created successfully.
```

Open the newly created form request class (`app/Http/Requests/ListFormRequest.php`) and you should see the following contents:

<?php namespace todoparrotHttpRequests;

use todoparrotHttpRequestsRequest;

class ListFormRequest extends Request {

```
1   public function authorize()
2   {
3       return false;
4   }
5
6   public function rules()
7   {
8       return [
9           //
10      ];
11  }
```

}

As a reminder, the `rules` method is used to define the validation rules which will be used in conjunction with the form fields. The list name and description are both logically required, so modify the method to look like this:

```
1     public function rules()
2     {
3         return [
4             'name' => 'required',
5             'description' => 'required'
6         ];
7     }
```

You'll also want to modify the `authorize` method to return `true` instead of `false`, because at this point in time we're going to allow anybody to use the form (I'll show you how to restrict access in Chapter 7):

```
1   public function authorize()
2   {
3     return true;
4   }
```

Updating the List Controller's Store Action

With the other pieces of the puzzle in place, all that remains is to update the `List` controller's `store` action to process the form contents. Of course, `ListFormRequest` handles the tiresome matter of validation, leaving us to focus solely on what to do with the data should it pass muster. In this instance all we need to do is save the data to the database, as demonstrated in the below revised `store` method:

```php
use todoparrot\Todolist;
use todoparrot\Http\Requests\ListFormRequest;

...

public function store(ListFormRequest $request)
{

    $list = new Todolist(array(
        'name' => $request->get('name'),
        'description' => $request->get('description')
    ));

    $list->save();

    return \Redirect::route('lists.create')->with('message', 'Your list has be\
en created!');

}
```

Once the list is saved, user are redirected to the list creation form should they desire to create another.

Updating a TODO List

Users will understandably occasionally wish to change a list name or description, so you'll need to provide a mechanism for updating an existing list. This feature's implementation is practically identical to that used for the list creation feature, with a few important differences. For starters, just as RESTful creation requires two actions (`create` and `store`), RESTful updates require two actions (`edit` and `update`). Open the `Lists` controller and you'll see these two action method skeletons are already in place:

```
1    public function edit($id)
2    {
3    }
4
5    public function update($id)
6    {
7    }
```

The `edit` action is responsible for serving the form (which is filled in with the existing list's data), and the `store` action is responsible for saving the updated form contents to the database. Notice how both actions accept as input an `$id`. This is the primary key of the list targeted for modification. If you recall from the RESTful discussion chapter 3, these two actions are accessed via (in the case of the `Lists`) controller `GET /lists/:id/edit` and `PUT /lists/:id`, respectively. If the `PUT` method is new to you, not to worry because Laravel handles all of the details associated with processing `PUT` requests, meaning all you have to do is construct the form and point it to the `update` route. Let's take care of this next.

Creating the TODO List Update Form and Edit Action

The form used to update a record is in most cases practically identical to that used to create a new record, with one very important difference; instead of `Form::open` you'll use `Form::model`:

```
1    {!! Form::model($list,
2      array('route' => ['lists.update', $list->id],
3            'class' => 'form')) !!}
4    ...
5    {!! Form::close() !!}
```

The `Form::model` method *binds* the enclosed form fields to the contents of a model record. Additionally, be sure to take note of how the list ID is passed into the `lists.update` route. This record is passed into the view like you would any other:

```
1    public function edit($id)
2    {
3
4      $list = Todolist::find($id);
5
6      return view('lists.edit')->with('list', $list);
7
8    }
```

When you pass a `Todolist` record into the `Form::model` method, it will bind the values of any attributes to form fields with a matching name. Let's create the entire form, however before doing so you'll need to create a new view named `edit.blade.php` and place it in the `resources/views/lists` directory. Then place the following contents into this view:

```
{!! Form::model($list, array('method' => 'put', 'route' => ['lists.update', $lis\
t->id], 'class' => 'form')) !!}

<div class="form-group">
    {!! Form::label('List Name') !!}
    {!! Form::text('name', null,
      array('required', 'class'=>'form-control',
            'placeholder'=>'San Juan Vacation')) !!}
</div>

<div class="form-group">
    {!! Form::label('List Description') !!}
    {!! Form::textarea('description', null,
      array('required', 'class'=>'form-control',
            'placeholder'=>'Things to do before leaving for vacation')) !!}
</div>

<div class="form-group">
    {!! Form::submit('Update List', array('class'=>'btn btn-primary')) !!}
</div>
{!! Form::close() !!}
```

Take special note of the form's method declaration. The `put` method is declared because we're creating a REST-conformant update request. After saving the changes to the `Lists` controller and `edit.blade.php` view, navigate to the list edit route, being sure to supply a valid list ID (e.g. `http://localhost:8000/lists/2/edit`) and you should see a populated form!

> In cases where the form used to create and edit a record are identical in every fashion except for the use of `Form::open` and `Form::model`, consider storing the form fields in a partial view and then inserting that partial into the `create` and `edit` views between the form opener and `Form::close` method.

Updating the List Controller's Update Action

With the `edit` action and corresponding view in place all that remains is to update the `update` action. In most cases you'll be able to simply reuse the form request helper created for use in conjunction with the `create` action, and in this case we'll go ahead and do so:

```
1    public function update($id, ListFormRequest $request)
2    {
3
4        $list = Todolist::find($id);
5
6        $list->update([
7          'name' => $request->get('name'),
8          'description' => $request->get('description')
9          ]);
10
11       return \Redirect::route('lists.edit',
12         array($list->id))->with('message', 'Your list has been updated!');
13
14   }
```

Make sure you update the input parameters passed into update to include the ListFormRequest request object. Once saved you should be able to edit existing lists!

Deleting TODO Lists

You typically won't need to create a form of any sort when deleting a record, however because you'll typically build this feature into an administration interface alongside facilities for inserting and updating records it seems most logical to discuss the matter of deletion within this chapter. When using RESTful controllers the destroy action is responsible for deleting the record, however this action is by default only accessible via the DELETE method.

```
1    $ php artisan route:list
2    +--------+--------------------+--------------+---------------------------+--\
3    ----------+
4    | Domain | URI                | Name         | Action                    | M\
5    iddleware |
6    +--------+--------------------+--------------+---------------------------+--\
7    ----------+
8    | ...    | ...                | ...          | ...                       | \
9            |
10   |        | DELETE lists/{lists} | lists.destroy | ...ListsController@destroy | \
11           |
12   +--------+--------------------+--------------+---------------------------+--\
13   ----------+
```

This means you can't just create a hyperlink pointing users to the lists.destroy, because hyperlinks by default use the GET method. Instead you'll use a form with a stylized button to create the appropriate link, as demonstrated below:

```
1  {!! Form::open(array('route' => array('lists.destroy', $list->id), 'method' => '\
2  delete')) !!}
3      <button type="submit">Delete List</button>
4  {!! Form::close() !!}
```

Notice how the `Form::open` method's `method` attribute is overridden (the default is `POST`) to instead use `DELETE`. The form's `route` attribute identifies the `lists.destroy` route as the submission destination, passing in the list ID. When submitted, the user will be taken to the `Lists` controller's `destroy` action, which looks like this:

```
1  public function destroy($id)
2  {
3
4    Todolist::destroy($id);
5
6    return \Redirect::route('lists.index')
7      ->with('message', 'The list has been deleted!');
8
9  }
```

Associating Tasks with Categories

As you learned in Chapter 4 it's really easy to programmatically associate categories with a list using a many-to-many relationship. To quickly recap, you can associate a new task with an existing list within your project controller like so:

```
1  $list = Todolist::find(1);
2
3  $task = new Task;
4
5  $task->name = 'Walk the dog';
6  $task->description = 'Walk Barky the Mutt around the block';
7
8  $list->save();
9
10 // Associate categories 3 and 4 with this list
11 $list->categories()->attach([3,4]);
```

But how might you go about effectively integrating this feature into a web form, as depicted in the below screenshot? The answer is easier than you think. As a bonus I'll introduce two very useful features you'll likely use repeatedly when building Laravel-driven forms.

Create a New List

List Name

> Gym Workout

List Description

> Exercises at today's gym session.

Categories

Categories

> Leisure
> **Exercise**
> Work
> Home Remodeling
> Landscaping

Create List

Creating a Nested List

To demonstrate how you might implement this feature, I'll revise the form used in the earlier section, "Creating the TODO List Form", adding a few additional fields for inputting several starter tasks:

```
1   {!! Form::open(array('route' => 'lists.store', 'class' => 'form')) !!}
2
3   <div class="form-group">
4       {!! Form::label('List Name') !!}
5       {!! Form::text('name', null,
6         array('required', 'class'=>'form-control',
7               'placeholder'=>'San Juan Vacation')) !!}
8   </div>
9
10  <div class="form-group">
11      {!! Form::label('List Description') !!}
12      {!! Form::textarea('description', null,
```

```
13          array('required', 'class'=>'form-control',
14                  'placeholder'=>'Things to do before leaving for vacation')) !!}
15      </div>
16
17      <h3>Categories</h3>
18
19      <div class="form-group">
20          {!! Form::label('Categories') !!}
21          {!! Form::select('categories', $categories, null,
22            array('multiple'=>'multiple','name'=>'categories[]')) !!}
23      </div>
24
25      <div class="form-group">
26          {!! Form::submit('Create List', array('class'=>'btn btn-primary')) !!}
27      </div>
28      {!! Form::close() !!}
```

This newly added bit of code creates a multiple select box containing a list of categories:

```
1    {!! Form::select('categories[]', $categories, null,
2      array('multiple'=>'multiple')) !!}
```

The Form::select method accepts four parameters. The first identifies the name of the field. The second identifies array used to populate the select field's id and name values for each option. The third field, which in this example is set to null, identifies any options (by ID) that should be selected by default. The fourth field identifies any HTML attributes which should be set. In this example we're ensuring the user can select multiple values. When rendered to the browser using the above code the multiple select box will look like this:

```
1    <select name="categories[]" multiple="multiple">
2      <option value="1">Leisure</option>
3      <option value="2">Exercise</option>
4      <option value="3">Work</option>
5      <option value="4">Home Remodeling</option>
6      <option value="5">Landscaping</option>
7    </select>
```

Next, you'll need to modify the Lists controller's create method to retrieve the list of categories used to populate the select field. Because you only want the categories table's id and name columns, you can use a convenient helper named lists which will create an array from the retrieved data, using the provided two columns for the array values and IDs. Here's an example executed within Tinker:

```
1  [1] > $c = \todoparrot\Category::lists('name', 'id');
2  // array(
3  //   1 => 'Leisure',
4  //   2 => 'Exercise',
5  //   3 => 'Work',
6  //   4 => 'Home Remodeling',
7  //   5 => 'Landscaping'
8  // )
```

ℹ Laravel 5.1 Update Alert

As of Laravel 5.1 the `lists` method's behavior has changed in the sense that it returns a collection instead of an array. You can however ensure `lists` continues to return an array by chaining the `all` method, such as `\todoparrot\Category::lists('name', 'id')->all()`.

Admittedly I find it weird you identify the column used for the array value before that used for the index, but in any case the method works great provided you keep this in mind, so all you'll need to do is retrieve the desired data using the `lists` method and pass it into the view:

```
1  public function create()
2  {
3    $categories = Category::lists('name', 'id');
4    return view('lists.create')->with('categories', $categories);
5  }
```

Finally, you'll update the `Lists` controller's `store` action to ensure any desired categories are attached to the newly created list:

```
1  public function store(ListFormRequest $request)
2  {
3
4    $list = new Todolist(array(
5        'name' => $request->get('name'),
6        'description' => $request->get('description')
7    ));
8
9    $list->save();
10
11   if (count($request->get('categories')) > 0) {
```

```
12        $list->categories()->attach($request->get('categories'));
13    }
14
15    return \Redirect::route('lists.create')
16        ->with('message', 'Your list has been created!');
17
18 }
```

Note how we first check the `categories` field to confirm it contains at least one category; if so the `attach` method is used to associate the selected categories with the newly created list. Of course, if the user is required to choose at least one category then consider encapsulating this validation within the associated form helper.

Uploading Files

I'm currently working on a Laravel 5 application which includes a restricted administration console used to manage products sold through an online catalog. Each product includes a name, SKU, price, description, and image. The image is uploaded using the `Form::file` helper made available through the LaravelCollective/html[123] package, validated alongside the other form inputs using a Laravel 5 form request, and if valid, stored in a special directory. In this section I'll show you how to integrate a similar form and upload capabilities into your own application.

Let's begin with a simplified version of the form used in the project. Again, this uses the LaravelCollective/html[124] package's form helpers to generate the various form fields:

```
1  {!! Form::open(
2    array(
3      'route' => 'admin.products.store',
4      'class' => 'form',
5      'novalidate' => 'novalidate',
6      'files' => true)) !!}
7
8  <div class="form-group">
9      {!! Form::label('Product Name') !!}
10     {!! Form::text('name', null, array('placeholder'=>'Chess Board')) !!}
11 </div>
12
13 <div class="form-group">
14     {!! Form::label('Product SKU') !!}
15     {!! Form::text('sku', null, array('placeholder'=>'1234')) !!}
```

---

[123]https://github.com/LaravelCollective/html
[124]https://github.com/LaravelCollective/html

```
16   </div>
17
18   <div class="form-group">
19       {!! Form::label('Product Image') !!}
20       {!! Form::file('image', null) !!}
21   </div>
22
23   <div class="form-group">
24       {!! Form::submit('Create Product!') !!}
25   </div>
26   {!! Form::close() !!}
27   </div>
```

Specific to the matter of file uploading there are two key characteristics of this form you'll need to keep in mind when implementing your own uploader:

- The `Form::open` method sets the `'files' => true` attribute. This results in the form data being encoded as "multipart/form-data", which is required when files will be included as form data.
- The `Form::file` helper is used to generate the file upload control.

When rendered to the browser the form looks like this:

Create a New Product

Product Name

Chess Board

Product SKU

1234

Product Image

Choose File   No file chosen

Create Product!

The file upload form

As you can see, the form is submitted to a route named `admin.products.store`. As is typical of any Laravel 5 application, the submitted form data is first routed through a form request. The validation rules are found in the form request's `rules()` method. Here's an example which validates the supplied image to ensure one is present and that it is specifically a PNG (image file):

```php
public function rules()
{
  return [
    'name'  => 'required',
    'sku'   => 'required|unique:products,sku,' . $this->get('id'),
    'image' => 'required|mimes:png'
  ];
}
```

You can validate uploads using plenty of other approaches such as ensuring it is a Word document or PDF. See the Laravel documentation[125] for more information. This request is passed into the `Admin/ProductController.php`'s `store` method, which looks like this:

---

[125]http://laravel.com/docs/master/validation

```
 1   public function store(ProductRequest $request)
 2   {
 3
 4     $product = new Product(array(
 5       'name' => $request->get('name'),
 6       'sku'  => $request->get('sku')
 7     ));
 8
 9     $product->save();
10
11     $imageName = $product->id . '.' .
12       $request->file('image')->getClientOriginalExtension();
13
14     $request->file('image')->move(
15       base_path() . '/public/images/catalog/', $imageName
16     );
17
18     return \Redirect::route('admin.products.edit',
19       array($product->id))->with('message', 'Product added!');
20
21   }
```

In this action we first save the product, and then process the image. There are *plenty* of different approaches to processing the uploaded image; I'm keeping this simple and just saving the image using a name matching the product ID, so for instance if the saved product using the ID 42 then the associated uploaded image will be named 42.png. The image name is first created (and stored in $imageName) and then it is moved into the application's /public/images/catalog directory.

Believe it or not, uploading an image using Laravel 5 is really that simple!

Summary

This was one of the more entertaining chapters to write because it really illustrates how you can begin introducing interactive features into your Laravel application. Stay tuned as in forthcoming revisions I'll continue to expand this chapter and demonstrate more complicated form features!

Chapter 6. Introducing Middleware

In a nutshell, middleware is non-domain specific code that can nonetheless interact with your application's request/response cycle. Examples of such code include authentication and authorization, caching, performance monitoring and content compression; while all of these features are crucial, none are domain-specific and therefore shouldn't require you to pollute your project's code in order to take advantage of them. Laravel 5 adds support for middleware, and even includes several useful middleware solutions which you can begin using within your applications right now. In this chapter I'll introduce you to the middleware included in your project, and even show you how to write your own custom middleware solution.

Introducing Laravel's Default Middleware

Open your project's `app/Http/Middleware` directory and you'll find three ready-made middleware solutions, including:

- `Authenticated.php`: This middleware is used to confirm a user is signed into the application. If not, the user is redirected to the login page. See Chapter 7 for more information about this middleware, although I'll also talk tangentially about it in the later section, "How Route-Level Middleware Works".
- `RedirectIfAuthenticated.php`: This middleware is used to confirm a user is *not* signed into the application. If so, the user is redirected to the home page. See Chapter 7 for more information about this middleware.
- `VerifyCsrfToken.php`: This middleware is used to manage CSRF protection. As of Laravel 5.1 you can disable CSRF protection on a per-URI basis by updating this middleware's `$except` array, or by altogethre removing the middleware from the `app/Http/Kernel.php`'s `$middleware` array, and then selectively enabling it on a per-route basis (I'll show you how to do this later in the chapter).

The `VerifyCsrfToken` middleware is automatically enabled for every route, however the other two (`Authenticated` and `RedirectIfAuthenticated`) are intended to be selectively applied according to specific route requests. You'll find the latter two route-level middlewares registered in `app/Http/Kernel.php`, in addition to a third middleware found under the `Illuminate` namespace:

```
1  protected $routeMiddleware = [
2  'auth' => \todoparrot\Http\Middleware\Authenticate::class,
3  'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
4  'guest' => \todoparrot\Http\Middleware\RedirectIfAuthenticated::class,
5  'iplogger' => \todoparrot\Http\Middleware\RequestLogger::class,
6  ];
```

In Chapter 7 I'll talk more about Laravel and authentication middleware, although you'll learn more about their general operation in the later section, "How Route-Level Middleware Works".

There are also several application-level middlewares, which you'll find defined in app/Http/Kernel.php's $middleware array:

```
1  protected $middleware = [
2      \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
3      \Illuminate\Cookie\Middleware\EncryptCookies::class,
4      \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
5      \Illuminate\Session\Middleware\StartSession::class,
6      \Illuminate\View\Middleware\ShareErrorsFromSession::class,
7      \App\Http\Middleware\VerifyCsrfToken::class,
8  ];
```

The EncryptCookies, AddQueuedCookiesToResponse, StartSession, and ShareErrorsFromSession middlewares are used by Laravel to manage various session-related features. The CheckForMaintenanceMode middleware is used to determine whether the site administrator has placed the application in maintenance mode (I'll show you how this feature works in Chapter 8). CheckForMaintenanceMode is identified as application-level middleware because you want all users to immediately be presented with the maintenance message should it be enabled, meaning this middleware must execute in conjunction with every request in order to respond accordingly. Finally, VerifyCsrfToken is defined as application-level middleware because anytime a CSRF token is submitted along with a form, the token must be verified, meaning the VerifyCsrfToken middleware must execute with every request to confirm whether one has been passed.

How Application-Level Middleware Works

Application-level middleware is intended to execute in conjunction with *every* request with the thinking that the event the middleware is intended to filter could occur anywhere within the application. In this section you'll learn more about how application-level middleware works by examining the VerifyCsrfToken middleware internals. The VerifyCsrfToken.php file is located here:

```
1  vendor/laravel/framework/src/Illuminate/
2   Foundation/Http/Middleware/VerifyCsrfToken.php
```

If you open it you'll find the following class:

```php
1  <?php
2
3  namespace Illuminate\Foundation\Http\Middleware;
4
5  use Closure;
6  use Illuminate\Support\Str;
7  use Symfony\Component\HttpFoundation\Cookie;
8  use Illuminate\Contracts\Encryption\Encrypter;
9  use Illuminate\Session\TokenMismatchException;
10
11  class VerifyCsrfToken {
12
13    protected $encrypter;
14
15    protected $except;
16
17    public function __construct(Encrypter $encrypter)
18    {
19      $this->encrypter = $encrypter;
20    }
21
22    public function handle($request, Closure $next)
23    {
24        if ($this->isReading($request) || $this->shouldPassThrough($request)
25          || $this->tokensMatch($request)) {
26            return $this->addCookieToResponse($request, $next($request));
27        }
28
29        throw new TokenMismatchException;
30    }
31
32    protected function shouldPassThrough($request)
33    {
34        foreach ($this->except as $except) {
35            if ($request->is($except)) {
36                return true;
37            }
```

```
38             }
39
40         return false;
41     }
42
43     protected function tokensMatch($request)
44     {
45         $token = $request->input('_token') ?:
46           $request->header('X-CSRF-TOKEN');
47
48         if (!$token && $header = $request->header('X-XSRF-TOKEN')) {
49             $token = $this->encrypter->decrypt($header);
50         }
51
52         return Str::equals($request->session()->token(), $token);
53     }
54
55     protected function addCookieToResponse($request, $response)
56     {
57         $response->headers->setCookie(
58             new Cookie('XSRF-TOKEN', $request->session()->token(),
59             time() + 60 * 120, '/', null, false, false)
60         );
61
62         return $response;
63     }
64
65
66     protected function isReading($request)
67     {
68         return in_array($request->method(), ['HEAD', 'GET', 'OPTIONS']);
69     }
70 }
```

All middleware implements a method named handle, which is responsible for processing the incoming request if the middleware implementation's parameters for doing so are met. In the case of VerifyCsrfToken, the HTTP method used for the request must include HEAD, GET, or OPTIONS (handled by the isReading method) and the token passed in via the $request object's input method must match that which was saved to the session when the form was originally generated (see Chapter 5 for more information about CSRF tokens if this doesn't make any sense). If they do match, a new cookie named XSRF-TOKEN is set which tells Laravel the tokens do indeed match, and the response is returned; if they don't match an exception of type TokenMismatchException is thrown.

So the bottom line is that when implementing a middleware you'll need to implement the `Middleware` contract, which at present just contains the single `handle` method. Provided you do that, and return the response as demonstrated in the above code, you're free to modify or respond to the request in any way you please. I'll show you a concrete example of examining and responding to a particular request in the later section, "Creating Your Own Middleware Solution".

How Route-Level Middleware Works

Route-level middleware works identically to application-level middleware, except that you can configure it to selectively execute in conjunction with a specific route or set of routes. For example if you look at the default `Auth` controller (`app/Http/Controllers/Auth/AuthController.php`) you'll see that the `guest` middleware (`guest` is defined as the alias for the `RedirectIfAuthenticated` middleware in `app/Http/Kernel.php`) is enabled in the class constructor:

```
1  public function __construct()
2  {
3      $this->middleware('guest', ['except' => 'getLogout']);
4  }
```

This means that the `RedirectIfAuthenticated` middleware will intercept every request made to an endpoint associated with the `Auth` controller *except* for the `getLogout` action. This means any already signed-in user attempting to access the login or registration endpoints defined in this controller will be redirected to the home page because it doesn't make any sense for them to register or login anew. If you have a look at the `RedirectIfAuthenticated` class (`app/Http/Middleware/RedirectIfAuthenticated.` you'll see the `handle` method is really simple to understand in that it uses Laravel's built-in authentication capabilities to determine whether the user is already signed in (via the `check` method, which is introduced in the next chapter).

Creating Your Own Middleware Solution

As an exercise let's create a simple route-level middleware solution that sends a message to the Laravel log when invoked. Begin by creating a new middleware skeleton using Artisan's `make:middleware` command:

```
1  $ php artisan make:middleware RequestLogger
2  Middleware created successfully.
```

This command created a new middleware class skeleton named `RequestLogger` that resides in the directory `app/Http/Middleware`. The class currently looks like this:

```php
1   <?php
2
3   namespace todoparrot\Http\Middleware;
4
5   use Closure;
6
7   class RequestLogger
8   {
9       public function handle($request, Closure $next)
10      {
11          return $next($request);
12      }
13  }
```

Modify the `handle` method to log the visitor's IP address to the Laravel log, and then pass on the request:

```php
1   public function handle($request, Closure $next)
2   {
3     \Log::info($request->ip());
4     return $next($request);
5   }
```

It is worth noting an important distinction here; when you return `$next($request)` you're instructing Laravel to execute this middleware *before* the request is processed. If you want to execute middleware *after* the request has been processed, you should change the `handle` logic to look like this:

```php
1   public function handle($request, Closure $next)
2   {
3     $response = $next($request);
4     \Log::info($request->ip());
5     return $response;
6   }
```

Next, open up `app/Http/Kernel.php` and register the new `RequestLogger` middleware:

```
1  protected $routeMiddleware = [
2  'auth' => \todoparrot\Http\Middleware\Authenticate::class,
3  'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
4  'guest' => \todoparrot\Http\Middleware\RedirectIfAuthenticated::class,
5  'iplogger' => \todoparrot\Http\Middleware\RequestLogger::class,
6  ];
```

Finally, just reference the `iplogger` alias whenever you'd like to execute the middleware. Just for the sake of demonstration I'll place the `middleware` call in the `Welcome` controller's constructor:

```
1  class HomeController extends Controller {
2
3    public function __construct()
4    {
5      $this->middleware('iplogger');
6    }
7
8    ...
9
10 }
```

After referencing the new middleware, reload an endpoint associated with the reference and check your log (`storage/logs`). If you are working from your local laptop you'll see a line that looks like this:

```
1  [2015-06-08 20:52:15] local.INFO: 127.0.0.1
```

The `127.0.0.1` is your local IP address. If you're running this code remotely, then you'll see a more recognizable IP address, such as `123.456.789.000`.

Using Middleware Parameters

New to Laravel 5.1 is support for middleware parameters. This is a long-awaited and tremendously useful feature. To illustrate its practicality, suppose you wanted to create a middleware (we'll call it `ProMiddleware`) which granted only those users having accumulated a particular number of forum points access to a particular set of controllers.

```
1  public function handle($request, Closure $next, $points)
2  {
3      if (! $request->user()->totalPoints() < $points) {
4          return \Redirect::route('welcome')
5      }
6
7      return $next($request);
8  }
```

When defining the middleware-restricted route, you'll identify the middleware name and the parameter like so:

```
1  Route::group(['middleware' => 'pro:10000'], function()
2  {
3      Route::resource('pro', 'ProController');
4  });
```

Summary

At the time of this writing middleware was still very new to Laravel 5 and so we haven't yet seen much traction in terms of community middleware contributions, however they are undoubtedly on the way! If you implement and open source any middleware I'd love to hear about it so I could mention it in a forthcoming revision. Be sure to e-mail me at wj@wjgilmore.com.

Chapter 7. Authenticating and Managing Your Users

Providing users with the ability to create and manage an account opens up a whole new world of possibilities in terms of enhanced interactivity and the creation and management of custom content. However, there are a great many matters one has to take into consideration in order to integrate account management features into an application, including user registration, secure storage of user credentials, user sign in and sign out, lost password recovery, profile management, and general integration of tailored features into your web application.

Fortunately, Laravel 5 removes numerous headaches associated with implementing many of these aforementioned features for you! In this chapter I'll show you how to configure and incorporate these bundled authentication features into your own application. You'll also learn how to enable Laravel 5.1's new authentication throttling feature, a useful tool for helping to keep the bad guys out.

Configuration Laravel Authentication

The `config/auth.php` file houses Laravel's authentication settings. In most cases you can leave the default settings untouched, however let's review the settings so you have a clear understanding of what's available:

- `driver`: This setting determines how users will be retrieved and authenticated. It's set to `eloquent`, meaning there are certain expectations such as the use of a model for managing the user and credentials. The other supported driver, `database`, causes Laravel to instead directly interact with the database rather than do so through a model. Unless you know what you're doing I suggest sticking with `Eloquent`.
- `model`: This setting tells Laravel what model will be used to maintain the user information (e-mail address, password, remember token). By default it's set to `App\User`. Why this is so will become clear later in the chapter.
- `table`: This setting identifies the database table used to store the user information (e-mail address, password, remember token, etc.). By default it's set to `users`. As with the `model` default setting, you'll soon understand why the Laravel developers chose this particular value.
- `password`: Laravel 5 stubs out some of the infrastructure required to manage and process password recovery requests. This setting identifies the view containing the content of the e-mail sent to requesting users, the database table used to manage the password recovery requests, and the number of minutes before the password recovery requests become invalid.

Registering Users

Implementing the user registration feature seems to be a particularly logical place to begin. Although it's fairly straightforward, this section is easily the longest in the chapter because there are a few other matters I necessarily need to introduce, beginning with the model used to manage the user accounts.

Introducing the User Model

The Laravel developers have saved you the hassle of building the model used to manage user accounts, placing a User model in the app directory.

```php
<?php

namespace Todoparrot;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable
  as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword
  as CanResetPasswordContract;

class User extends Model
  implements AuthenticatableContract, CanResetPasswordContract
{
    use Authenticatable, CanResetPassword;

    /**
     * The database table used by the model.
     *
     * @var string
     */
    protected $table = 'users';

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name', 'email', 'password'];
```

```
31
32      /**
33       * The attributes excluded from the model's JSON form.
34       *
35       * @var array
36       */
37      protected $hidden = ['password', 'remember_token'];
38  }
```

This `User` model looks rather different from the models we've created in earlier chapters, notably because in addition to extending the `Model` class it *implements* two contracts, including `AuthenticatableContract` and `CanResetPasswordContract`. A contract defines an interface to a particular implementation of a set of features. For instance, `AuthenticatableContract` defines an interface for obtaining the user's unique identifier and password and for managing the "remember me" token should it be enabled. Because the interface ensures the functionality is loosely coupled, you're free to easily swap out the Laravel implementation for another.

The contracts work in unison with the *traits* (a feature new to PHP as of version 5.4). As you can see, the `User` model uses two traits, including `Authenticatable` and `CanResetPassword`. Traits offer a useful alternative to multiple inheritance (something PHP can't do natively), allowing you to inherit methods from several different classes, thereby avoiding code duplication. Therefore the contract defines the interface, and the trait identifies the interface implementation. You could of course swap out the implemented traits for your own implementations, provided you meet the requirements defined in the contract.

> Philip Brown penned a great introductory tutorial[126] to traits. It's definitely worth taking the time to read now if this concept is new to you.

You learned about the `$table` property in Chapter 3; as a reminder it specifies the name of the model's underlying table. The `$fillable` property identifies the columns that can be inserted/updated by way of mass assignment. Finally, the `$hidden` property is used to identify columns that should not be passed into JSON or arrays. Logically we don't want to expose the password (even if in hashed format) nor the session remember token, and so these are identified in the `$hidden` property.

Introducing the Users Table

In addition to creating the `User` model, the Laravel developers also created the corresponding `users` table migration, which you'll find in `database/migrations/2014_10_12_000000_create_users_-table.php`. Because by this point in the book you likely already ran `artisan:migrate`, then the `users` table already exists in your development database. The table looks like this:

---

[126]http://culttt.com/2014/06/25/php-traits/

```
 1  mysql> describe users;
 2  +---------------+------------------+------+-----+...
 3  | Field         | Type             | Null | Key |...
 4  +---------------+------------------+------+-----+...
 5  | id            | int(10) unsigned | NO   | PRI |...
 6  | name          | varchar(255)     | NO   |     |...
 7  | email         | varchar(255)     | NO   | UNI |...
 8  | password      | varchar(60)      | NO   |     |...
 9  | remember_token | varchar(100)    | YES  |     |...
10  | created_at    | timestamp        | NO   |     |...
11  | updated_at    | timestamp        | NO   |     |...
12  +---------------+------------------+------+-----+...
13  7 rows in set (0.01 sec)
```

Fortunately, you won't have to construct any custom logic to interact with this table, because it's already in place! Read on to learn more.

Introducing the Account Registration Feature

Laravel 5.0 included several convenient registration-related routes in `app/Http/routes.php`, however these were inexplicably removed with the version 5.1 release. You can however easily add them back:

```
1  Route::get('auth/register', 'Auth\AuthController@getRegister');
2  Route::post('auth/register', 'Auth\AuthController@postRegister');
```

The `Auth` controller is responsible for managing both new user registration and user authentication. You'll find this controller in `app/Http/Controllers/Auth/AuthController.php`.

With the routes in place, you'll next need to create the registration view. An example view was included by default with Laravel 5.0 applications but was removed as of version 5.1. Creating one is easy however provided you name it `register.blade.php` and place it inside a directory named `auth` found in `resources/views`. Here is an example, Bootstrap-stylized form (which also takes advantage of the `LaravelCollective/Html` form facade):

```
 1  @extends('layouts.master')
 2
 3  @section('content')
 4  <div class="col-md-6">
 5  {!! Form::open(array('url' => '/auth/register',
 6    'class' => 'form')) !!}
 7
 8  <h1>Create a TODOParrot Account</h1>
 9
10  @if (count($errors) > 0)
11    <div class="alert alert-danger">
12      There were some problems creating an account:
13      <ul>
14        @foreach ($errors->all() as $error)
15          <li>{{ $error }}</li>
16        @endforeach
17      </ul>
18    </div>
19  @endif
20
21  <div class="form-group">
22      {!! Form::label('name', 'Your Name') !!}
23      {!! Form::text('name', null, array('class'=>'form-control', 'placeholder'=>'\
24  Name')) !!}
25  </div>
26  <div class="form-group">
27      {!! Form::label('Your E-mail Address') !!}
28      {!! Form::text('email', null,
29        array(
30        'class'=>'form-control',
31        'placeholder'=>'Email Address')
32        ) !!}
33  </div>
34  <div class="form-group">
35      {!! Form::label('Your Password') !!}
36      {!! Form::password('password',
37        array('class'=>'form-control', 'placeholder'=>'Password')) !!}
38  </div>
39  <div class="form-group">
40      {!! Form::label('Confirm Password') !!}
41      {!! Form::password('password_confirmation',
42        array(
```

```
43              'class'=>'form-control',
44              'placeholder'=>'Confirm Password')
45          ) !!}
46   </div>
47
48   <div class="form-group">
49      {!! Form::submit('Create My Account!',
50        array('class'=>'btn btn-primary')) !!}
51   </div>
52   {!! Form::close() !!}
53   </div>
54   @endsection
```

After saving this view, you'll be able to access the registration form by navigating to /auth/regis-ter. The rendered form is presented in the below screenshot.

Create a TODOParrot Account

Your Name

Name

Your E-mail Address

Email Address

Your Password

Password

Confirm Password

Confirm Password

Create My Account!

An example registration form

Go ahead and register, and you'll see that a new record will be inserted into the users table. Further, Laravel will automatically sign you into the site and attempt to redirect to the /home URI. If this URI doesn't exist then you'll receive a 404 error. You can either define the route in your routes file, or you can override the destination URI by overriding the $redirectTo property within the AuthController.php file:

```
1   protected $redirectTo = '/';
```

Introducing the Account Sign In Feature

Like user registration, Laravel applications include all of the logic required to authenticate registered users, however as of version 5.1 you'll need to do a bit of additional work to integrate the feature into your application. Begin by adding the following two routes to your routes.php file:

```
1   Route::get('auth/login', 'Auth\AuthController@getLogin');
2   Route::post('auth/login', 'Auth\AuthController@postLogin');
```

Next, create a file named login.blade.php inside resources/views/auth, and add the following contents to it:

```
1   @extends('layouts.master')
2
3   @section('content')
4   <div class="col-md-6">
5
6   {!! Form::open(array('url' => '/auth/login', 'class' => 'form')) !!}
7
8     <h1>Sign In to Your TODOParrot Account</h1>
9
10    @if (count($errors) > 0)
11
12      <div class="alert alert-danger">
13        There were some problems signing into your account:
14        <ul>
15          @foreach ($errors->all() as $error)
16            <li>{{ $error }}</li>
17          @endforeach
18        </ul>
19      </div>
20
21    @endif
22
23    <div class="form-group">
24      {!! Form::label('email', 'Your E-mail Address') !!}
25      {!! Form::text('email', null,
26        array('class'=>'form-control', 'placeholder'=>'E-mail')) !!}
27    </div>
28
```

```
29    <div class="form-group">
30      {!! Form::label('Your Password') !!}
31      {!! Form::password('password',
32        array('class'=>'form-control', 'placeholder'=>'Password')) !!}
33    </div>
34
35    <div class="form-group">
36      <label>
37        {!! Form::checkbox('remember', 'remember') !!} Remember Me
38      </label>
39    </div>
40
41    <div class="form-group">
42      {!! Form::submit('Login', array('class'=>'btn btn-primary')) !!}
43    </div>
44
45    <a href="/password/email">Forgot Your Password?</a>
46    </div>
47    </div>
48 {!! Form::close() !!}
49
50 </div>
51 @endsection
```

With the routes and view in place, you can navigate to /auth/login to see and use the sign in form. If you used the above code you'll see the form presented in the below screenshot.

Sign In to Your TODOParrot Account

Your E-mail Address

> E-mail

Your Password

> Password

☐ **Remember Me**

Login

Forgot Your Password?

An example sign in view

Go ahead and sign in using the account you created in the last section, and as before you'll be redirected to the Home controller. Like the registration form, you can easily modify the sign in view to suit your needs by editing the view found at `resources/views/auth/login.blade.php`.

Remember from the earlier registration-related discussion that successfully authenticated users will be redirected to the `/home` URI default. You can either define a route in `routes.php`, or override the default redirection URL by adding the following property to the `Auth` controller (found in `app/Http/Controllers/Auth/AuthController.php`):

```
1  protected $redirectPath = '/';
```

Enabling Authentication Throttling

Laravel 5.1 introduced a new feature called *authentication throttling* which will prevent further authentication attempts for one minute if the user attempts to sign in more than five times without success. At the time of this writing both the attempt count and the delay were hard coded into the `ThrottlesLogins` trait (located at `vendor/laravel/framework/src/Illuminate/Foundation/Auth/ThrottlesLogins.php`), however perhaps at some point in the future these values will be defined in a configuration file. In any case, to enable authentication throttling, open your `AuthController` class (`app/Http/Controllers/Auth/AuthController.php`) and add the `ThrottlesLogins` trait:

```
1  use Illuminate\Foundation\Auth\ThrottlesLogins;
2
3  ...
4
5  class AuthController extends Controller {
6
7    use AuthenticatesAndRegistersUsers, ThrottlesLogins;
8
9    ...
10
11 }
```

Signing Out of an Account

All Laravel applications include the ability to end an authenticated session by signing the user out of his account. To enable this feature (as of version 5.1) you'll need to add the following route to your `routes.php` file:

```
1  Route::get('auth/logout', 'Auth\AuthController@getLogout');
```

Password Recovery

All Laravel applications include password recovery/reset logic however as of version 5.1 you'll need to complete some additional work to integrate the feature into your application. Begin by adding the following routes to `routes.php`:

```
1   Route::get('password/email', 'PasswordController@getEmail');
2   Route::post('password/email', 'PasswordController@postEmail');
3
4   Route::get('password/reset/{token}', 'PasswordController@getReset');
5   Route::post('password/reset', 'PasswordController@postReset');
```

Next you'll need to create a view named `password.blade.php`, adding it to `resources/views/auth`. Here's an example view:

```
1   @extends('layouts.master')
2
3   @section('content')
4   <div class="col-md-6">
5
6   {!! Form::open(array('url' => '/password/email', 'class' => 'form')) !!}
7
8   <h1>Recover Your Password</h1>
9
10  @if (count($errors) > 0)
11      <div class="alert alert-danger">
12        There were some problems recovering your password:
13        <br />
14        <ul>
15          @foreach ($errors->all() as $error)
16              <li>{{ $error }}</li>
17          @endforeach
18        </ul>
19      </div>
20  @endif
21
22  <div class="form-group">
23    {!! Form::label('email', 'Your E-mail Address') !!}
24    {!! Form::text('email', null,
25      array('class'=>'form-control', 'placeholder'=>'E-mail')) !!}
26  </div>
27
```

```
28  <div class="form-group">
29    {!! Form::submit('E-mail Password Reset Link',
30      array('class'=>'btn btn-primary')) !!}
31  </div>
32  {!! Form::close() !!}
33  </div>
34  @endsection
```

Next, you'll need to create the e-mail text which is sent to the user after submitting the form. You must name this view `password.blade.php` and place it in the directory `resources/views/emails`. Here is an example view:

```
1  Dear User,
2
3  Oh noes! You've forgotten your password. No big deal,
4  click on the following URL to reset it:
5
6  {{ url('password/reset/'.$token) }}
```

With the route and views in place, you can navigate to `/password/email` and you'll see the form presented in the below screenshot:

Recover Your Password

Your E-mail Address

E-mail

E-mail Password Reset Link

Initiating password recovery

Once submitted, if the supplied e-mail address matches a record found in the `users` table, a record such as the following is added to the `password_resets` table:

```
1  mysql> select * from password_resets;
2  +-----------------+--------------------+---------------------+
3  | email           | token              | created_at          |
4  +-----------------+--------------------+---------------------+
5  | wj@wjgilmore.com | asdfasfaasfasfasdf | 2015-06-08 19:26:02 |
6  +-----------------+--------------------+---------------------+
7  1 row in set (0.00 sec)
```

In addition to updating the `password_resets` table, an e-mail will be sent to the e-mail address supplied by the requesting user. This e-mail will contain a link to the password recovery interface, and will include the recovery token found in the `password_resets` table. When the user clicks this link he'll be able to choose a new password. The default recovery e-mail is quite sparse, however you can update it to include whatever additional information you please by modifying the file `resources/views/emails/password.blade.php`.

> ⚠ This e-mail won't be successfully sent until you configure `config/mail.php`. See Chapter 5 for more information about configuring Laravel's e-mail delivery feature.

The recovering user will have 60 minutes to click on the recovery link per the `config/auth.php` file's `password['expire']` setting. Obviously you can change this setting to whatever value you desire. For instance to give users up to 24 hours to recover the password, you'll set `expire` to `1440`.

Retrieving the Authenticated User

You can retrieve the `users` record associated with the authenticated user via `Auth::user()`. For instance, to retrieve the user's name you'll access `Auth::user()` like so:

```
1  Welcome back, {{ Auth::user()->name }}!
```

Of course, you'll want to first ensure the user is authenticated before attempting to access the record in this fashion. You can do so by consulting `Auth::check()`:

```
1  @if (Auth::check())
2    Welcome back, {{ Auth::user()->name }}!
3  @else
4    Hello, stranger! <a href="/auth/login">Login</a>
5    or <a href="/auth/register">Register</a>.
6  @endif
```

Conversely, you can flip the conditional around, instead `Auth::guest()` to determine if the user is a guest:

```
1  @if (Auth::guest())
2    <li><a href="/auth/login">Login</a></li>
3    <li><a href="/auth/register">Register</a></li>
4  @else
5    Welcome back, {{ Auth::user()->name }}!
6  @endif
```

Restricting Access to Authenticated Users

As I mentioned in the earlier section, "Introducing the Account Registration Feature", you can restrict access to a controller by referencing the auth middleware in the appropriate controller constructor:

```
1  public function __construct()
2  {
3    $this->middleware('auth');
4  }
```

When users attempt to access the restricted controller, Laravel will first check for a valid session. If the session exists, access to the controller will be granted; otherwise the user will be redirected to the sign in view.

Restricting Forms to Authenticated Users

All generated form request classes automatically include a method named authorize. This method is used to determine whether the form is available to all users or to some restricted subset. by default it is set to false, and so in chapter 5 we updated the form requests discussed therein to instead return true because we had not yet integrated user accounts. To refresh your memory here's what a default authorize method looks like inside a newly generated form request:

```
1  <?php namespace todoparrot\Http\Requests;
2
3  use todoparrot\Http\Requests\Request;
4
5  class ContactFormRequest extends Request {
6
7    public function authorize()
8    {
9      return false;
10   }
11
```

```
12    ...
13
14  }
```

If you'd like to restrict a form request to authenticated users, you can modify the `authorize()` method to look like this:

```
1  public function authorize()
2  {
3    return Auth::check();
4  }
```

Keep in mind you're free to embed into `authorize()` whatever logic you deem necessary to check a user's credentials. For instance if you wanted to restrict access to not only authenticated users but additionally only those who are paying customers, you can retrieve the user using `Auth::user()` and then traverse whatever associations are in place to determine the user's customer status.

Creating Route Aliases

Next we'll need to define routes for displaying the registration form and then processing the form submission. Open `app/routes.php` and add the following lines:

```
1  get('/signup', array('as' => 'signup',
2    'uses' => 'Auth\AuthController@getRegister'));
3  post('/signup', array('as' => 'signup', 'uses' => 'Auth\AuthController@postRegis\
4  ter'));
```

After saving the changes you should be able to navigate to `/signup` and see the registration form.

Adding New Properties to the User Model

TODO http://code.tutsplus.com/tutorials/using-laravel-5s-authentication-facade–cms-23461

Summary

User accounts undoubtedly add another level of sophistication to your application, and Laravel makes it so incredibly easy to integrate these capabilities that it almost seems a crime to not make them available!

Chapter 8. Deploying, Optimizing and Maintaining Your Application

After a great deal of planning, coding and and deliberation it's time to launch your project. While an important milestone, your work is hardly done. Among other things you'll need to ensure your application is properly optimized in order to handle the onslaught of traffic, implement a convenient and foolproof deployment process, and effectively monitor your application for hiccups and other unexpected issues. You'll also need to carry out an assortment of ongoing administrative tasks, many of which will need to execute according to a rigorous schedule. In this chapter I'll touch upon all of these subjects, hopefully helping you to sort out at least some of these mission-critical issues along the way.

Introducing the Laravel 5 Command Scheduler

Suppose you wanted to create a new TODOParrot revenue stream by adding a productivity-centric book catalog to the site. Interested readers would click through to Amazon, and you would earn money on any purchases via your Amazon Associates Account[127]. Of course, in an effort to convert as many sales as possible you'll want to ensure your book catalog always contains the latest available book covers, descriptions, and prices, something you'd rather not do manually.

Fortunately, you can automate such updates using the Amazon Product Advertising API[128]. To implement such a solution you would typically write a script using a package such as ApaiIO[129], and then schedule the script's execution using your server's Cron[130] service. While this approach certainly works, managing task scheduling outside of your application code is pretty inconvenient.

Laravel 5 removes this inconvenience with the introduction of a command scheduler. The Laravel command scheduler allows you to manage your task execution dates and times using easily understandable PHP syntax. You'll manage the task execution definitions in app/Console/Kernel.php, which is presented below. You'll see that an example task has already been defined in the schedule method to run every hour:

---

[127]https://affiliate-program.amazon.com/
[128]https://affiliate-program.amazon.com/gp/advertising/api/detail/main.html
[129]https://github.com/Exeu/apai-io
[130]http://en.wikipedia.org/wiki/Cron

```php
1  <?php namespace todoparrot\Console;
2
3  use Illuminate\Console\Scheduling\Schedule;
4  use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
5
6  class Kernel extends ConsoleKernel {
7
8      protected $commands = [
9          'todoparrot\Console\Commands\Inspire',
10     ];
11
12     protected function schedule(Schedule $schedule)
13     {
14         $schedule->command('inspire')->hourly();
15     }
16
17 }
```

The protected $commands property registers any custom commands you'd like to include in the Artisan list output. An example custom command (Inspire) is already defined, which you'll find in app/Console/Commands/Inspire.php. Whether you plan on scheduling custom Artisan commands or executing them directly from the terminal you'll need to reference You're not strictly limited to scheduling Artisan commands, although as you'll soon see it is quite easy to create custom Artisan commands containing the desired logic.

The inspire command registered in the $commands array is scheduled for execution in the schedule method. In this example you can see it has been scheduled to execute every hour (at the top of the hour). I'll talk about other scheduling options in a moment. If you execute the inspire command manually you'll be presented with a random quote:

```
1  $ php artisan inspire
2  Simplicity is the ultimate sophistication. - Leonardo da Vinci
```

Next I'll show you how you can create your own custom Artisan command and schedule it for execution using the command scheduler.

Creating a Custom Artisan Command

You can create your own Artisan commands which can neatly package any PHP logic you desire. To create a command use the make:console generator:

```
1  $ php artisan make:console UpdateCatalog --command=amazon:update
2  Console command created successfully.
```

This creates a command skeleton in app/Console/Commands/UpdateCatalog.php. For organizational purposes I've define a custom command name amazon:update, as perhaps in the future I'd like to create other Amazon-related commands and so would like them all placed under the amazon namespace. Open up app/Console/Commands/UpdateCatalog.php and you'll find the following class:

```php
1   <?php namespace todoparrot\Console\Commands;
2
3   use Illuminate\Console\Command;
4   use Symfony\Component\Console\Input\InputOption;
5   use Symfony\Component\Console\Input\InputArgument;
6
7   class UpdateCatalog extends Command {
8
9     protected $name = 'amazon:update';
10
11    protected $description = 'Command description.';
12
13    public function __construct()
14    {
15      parent::__construct();
16    }
17
18    public function fire()
19    {
20      //
21    }
22
23    protected function getArguments()
24    {
25      return [
26        ['example', InputArgument::REQUIRED,
27          'An example argument.'],
28      ];
29    }
30
31    protected function getOptions()
32    {
33      return [
```

```
34          ['example', null, InputOption::VALUE_OPTIONAL,
35              'An example option.', null],
36      ];
37  }
38
39  }
```

The $name and $description properties define the command's execution name and description, respectively, both of which will be included in the Artisan list output once we register it within the app/Console/Kernel.php $commands array. The fire method encapsulates the logic which will execute when the command is run. The getArguments and getOptions methods can be used to define both required and optional command arguments and options, respectively.

You'll see that the getArguments method defines a required argument. For the purposes of this exercise we're not interested in arguments nor options, so comment out the return statement:

```
1  protected function getArguments()
2  {
3  return [
4    // ['example', InputArgument::REQUIRED, 'An example argument.'],
5  ];
6  }
```

The Laravel documentation discusses Artisan command arguments, options, and other features. See this page[131] for more information.

Next, update the fire method to look like this:

```
1    public function fire()
2    {
3      $this->info("Amazon catalog updated!");
4    }
```

Save your changes and then register the command within app/Console/Kernel.php:

---

[131]http://laravel.com/docs/master/commands

```
1  protected $commands = [
2      'todoparrot\Console\Commands\Inspire',
3      'todoparrot\Console\Commands\UpdateCatalog'
4  ];
```

After saving the changes you should see the custom command in the Artisan `list` output:

```
1  $ php artisan list
2  ...
3  Available commands:
4  ...
5  amazon
6   amazon:update       Updates the TODOParrot book catalog.
7  ...
```

You can now execute the `amazon:update` command from the terminal:

```
1  $ php artisan amazon:update
2  Amazon catalog updated!
```

Scheduling Your Command

As was perhaps made obvious by the earlier example, scheduling your command within app/Console/Kernel.php is easy. If you'd like `amazon:update` to run hourly, you'll use the `hourly` method:

```
1  protected function schedule(Schedule $schedule)
2  {
3      $schedule->command('amazon:update')->hourly();
4  }
```

Updating Amazon product information hourly seems a bit aggressive. Fortunately, you have plenty of other options. To run a command on a daily basis (midnight), use `daily`:

```
1  $schedule->command('amazon:update')->daily();
```

To run it at a specific time, use the `dailyAt` method:

```
1  $schedule->command('amazon:update')->dailyAt('18:00');
```

If you need to run a command very frequently, you can use an `every` method:

```
1  $schedule->command('amazon:update')->everyFiveMinutes();
2  $schedule->command('amazon:update')->everyTwentyMinutes();
```

See the Laravel documentation[132] for other scheduling options.

Enabling the Scheduler

With your tasks created and scheduled, you'll need to add a single entry to your server's crontab file:

```
1  * * * * * php /path/to/artisan schedule:run 1>> /dev/null 2>&1
```

Once saved, your application's schedule:run Artisan command will run once per minute. It will in turn execute any jobs that you've defined using the Laravel command scheduler.

Other Scheduling Options

If defining a custom Artisan command seems overkill, you can optionally define some logic for execution directly within the schedule method:

```
1  $schedule->call(function()
2  {
3      // Send some e-mail
4
5  })->daily();
```

You can also schedule terminal commands for execution like so:

```
1  $schedule->exec('/path/to/some/command')->daily();
```

The new command scheduler is a pretty powerful tool that eliminates the need to separately manage regularly executing tasks. This is a supremely well-implemented feature, and may very well be my favorite Laravel 5 capability.

Optimizing Your Application

Before deploying your application you'll logically want to ensure the code has been properly optimized for a production environment. Frankly, entire books have been written about tuning web applications, and therefore this discussion could go in many directions and really never even begin scratch the surface. Therefore I think it makes the most sense to focus on a few key *Laravel-specific* optimization features, for the moment leaving third-party optimization solutions out of the discussion. In future iterations I'll selectively expand this section to cover other topics.

---

[132]http://laravel.com/docs/master/artisan#scheduling-artisan-commands

Creating a Faster Class Loader

A Laravel application's request and response life cycle obviously involves quite a few different classes. As you might imagine, loading and invoking dozens of different classes with each request can be quite a detriment to performance. Laravel offers a solution for creating an optimized class loader by way of Composer[133]. You can improve performance by using Artisan's `optimize` command to significantly improve the efficiency in which your project classes are loaded. You'll invoke `optimize` like so:

```
1 $ php artisan optimize
```

This command will by default run Composer's `dump-autoload` command with the `--optimize` option. This command will in turn create `vendor/composer/autoload_classmap.php` which contains an array consisting of all class names and the paths to their corresponding files. This file is then subsequently used to quickly load third-party classes because the array can be used for referential purposes rather than requiring the autoloader to separately find and open each class file.

Additionally, Laravel will further optimize matters by concatenating all of its own native classes into a single file found at `storage/framework/compiled.php`, and also create a file named `services.json` in the same directory. This file is used to optimize the loading of your project's service providers. Further, it will cache all of your project's views within `storage/framework/views`.

However, if your project's `APP_DEBUG` configuration setting is set to `true` (the default when in your development environment), this command will not work as intended because Laravel presumes you'll always want to be working with the very latest versions of your files rather than rely on a cache. You can override this behavior in the local environment by including the `--force` option:

```
1 $ php artisan optimize --force
2 Generating optimized class loader
3 Compiling common classes
4 Compiling views
```

Now you'll be able to peruse `storage/framework/compiled.php` and `storage/framework/services.json` even when working in the local environment.

You can replace `compiled.php` and `services.json` by running `optimize` anew, keeping in mind you'll need the `--force` option if you'd like to experiment with it locally. If you'd like to remove these files altogether, run the following command:

```
1 $ php artisan clear-compiled
```

---

[133]https://getcomposer.org/

This command (also confusingly) does not however automatically delete the compiled views which were generated when the `--force` option was provided. At this time there is not any documented solution for deleting these files via Artisan, so you could optionally (and rather easily) write your own Artisan command for doing so, or just navigate to `storage/framework/views` and manually delete all of the files.

Caching Route Definitions

The route definitions found in `app/Http/routes.php` are by default read into the framework as part of the bootstrapping process. You can cache these routes by encoding and serializing them using Artisan's `route:cache` command:

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

This cache file is stored in `vendor/routes.php`. If this file exists, Laravel will refer to it rather than parsing the source route definitions file. You can delete the route cache file using `route:clear`:

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

Optimizing Your CSS and JavaScript

You'll always want to minimize the number and size of requests required to render your site within the user's browser. One of the easiest things you can do in this regards is to optimize your CSS and JavaScript, as well as take advantage of content delivery networks.

Combining your CSS using Elixir and a CSS preprocessor such as Less[134] is pretty easy; take a look at `resources/assets/less/app.less` and you'll see how to use Less' `@import` directive to combine multiple CSS files:

```
1  @import "bootstrap/bootstrap";
2
3  @btn-font-weight: 300;
4  @font-family-sans-serif: "Roboto", Helvetica, Arial, sans-serif;
5
6  body, label, .checkbox label {
7          font-weight: 300;
8  }
```

---

[134]http://lesscss.org/

Laravel's default `gulpfile.js` uses `mix.less` to compile the `app.less` file, saving the combined CSS output to `public/css/app.css`. This is great because it combines the more than 40 Bootstrap CSS source files found in `resources/assets/less/bootstrap` and the custom Less CSS found in `app.less` into a single file. However you'll additionally want to minify the CSS (remove all whitespace and comments). You can do so by passing `--production` to `gulp`:

```
1  $ gulp --production
```

Just taking the default `app.less` and Bootstrap files into account, using `--production` results in a 20% reduction in the compiled `app.css` file size!

You can additionally combine your JavaScript files together. For instance if you are managing two separate CoffeeScript files in `resources/assets/coffee` named `test.coffee` and `test2.coffee`, and wanted to combine the compiled output into a single file within `public/js` you can update `gulpfile.js` like so:

```
1  mix.coffee().scriptsIn('public/js', 'public/js');
```

When the CoffeeScript files found in `resources/assets/coffee` are compiled and saved to `public/js`, the `scriptsIn` method will subsequently concatenate these files together and save them to a file named `all.js`. Passing `--production` to the `gulp` command will additionally minify the concatenated JavaScript file.

When relying on third-party libraries such as jQuery you'll almost certainly want to use a CDN (Content Delivery Network) rather than locally host your own copy. This may seem counterintuitive, however the reasoning behind this best practice is explained here[135].

Deploying Your Application

Because you'll presumably be regularly updating the application to include new features and bug fixes, it is imperative for you to implement a convenient and flexible deployment solution. Such as solution would not only facilitate the transfer of project files to your production server but additionally handle other crucial tasks such as database migrations and asset compilation. In this section I'll guide you through a simple deployment process involving Heroku[136], and introduce you to Laravel Forge[137]. Keep in mind however these are just two of many possible deployment solutions; in forthcoming updates I'll be sure to expand this section significantly to discuss other approaches.

---

[135]http://encosia.com/3-reasons-why-you-should-let-google-host-jquery-for-you/

[136]http://heroku.com

[137]https://forge.laravel.com

Deploying to Heroku

Heroku[138] is without a doubt my favorite hosting solution, insomuch that I'm currently writing another book devoted entirely to the topic (see "Easy Heroku for Busy Rails Developers"[139]). Heroku is a cloud platform as a service (PaaS) that in the years since its founding has become a darling of the Ruby on Rails community, however the Heroku team hasn't shied away from expanding its offerings and now supports Clojure, Java, Node.js, and PHP, among other languages.

If you're experimenting with Laravel or are planning on managing a relatively small project, you might find Heroku particularly compelling in that it offers a free entry level hosting tier. If your hosting requirements are somewhat more ambitious then you'll want to take the time to carefully review Heroku's pricing options[140] as the bills can add up rather quickly. However Heroku really does live up to the adage, "you get what you pay for", because in my opinion they offer unsurpassed service. If anything, it doesn't hurt to create a free Heroku account and follow along with the deployment instructions described in this section; you can always easily delete the deployment if you later decide Heroku isn't for you.

Creating a Heroku Account

To get started, you'll first need to create a new Heroku account (https://signup.heroku.com[141]). Doing so is free and only takes a quick moment to do. At registration time you'll be prompted to choose your desired development language. Go ahead and choose PHP however keep in mind doing so doesn't limit your ability to later use Heroku in conjunction with other supported languages.

Installing the Heroku Toolbelt

After creating your account you'll next need to install the Heroku Toolbelt (https://toolbelt.heroku.com/[142]). The Heroku Toolbelt is a terminal utility you'll use to manage various aspects of your Heroku-hosted project, including the actual deployment process, migrating your database, and interacting with the Heroku servers in various ways. To install the Heroku Toolbelt, head on over to https://toolbelt.heroku.com/[143], where you'll find either download binaries or installation instructions for OS X, Windows, Debian/Ubuntu, and other Linux distributions.

Once installed, open a terminal and execute `heroku`:

---

[138]http://heroku.com

[139]http://www.wjgilmore.com/books/easy-heroku-rails.html

[140]https://www.heroku.com/pricing

[141]https://signup.heroku.com

[142]https://toolbelt.heroku.com/

[143]https://toolbelt.heroku.com/

```
1  $ heroku
2  Usage: heroku COMMAND [--app APP] [command-specific-options]
3
4  Primary help topics, type "heroku help TOPIC" for more details:
5
6  addons      #  manage addon resources
7  apps        #  manage apps (create, destroy)
8  ...
9  update         #  update the heroku client
10 version        #  display version
```

You'll be greeted with a lengthy list of commands. Introducing all of these commands is well out of the scope of this chapter, however feel free to take a moment to read the command descriptions and learn more about them by executing `heroku help` and then the name of the command (e.g. `heroku help logs`).

Deploying Your Application

With your Heroku account created and the Heroku Toolbelt installed, it's time to deploy a Laravel application. As you'll soon see, this is incredibly easy to do. For purposes of this example let's just deploy a new application:

```
1  $ composer create-project laravel/laravel dev.herokutest.com --prefer-dist
2  Installing laravel/laravel (v5.0.0)
3  -      Installing laravel/laravel (v5.0.0)
4  ...
5  Compiling views
6  Application key [9UCBk7IDjvAGrkLOUBXw43yYKlymlqE3Y] set successfully.
```

With the project created, you'll next want to create a `Procfile`, placing this file in your Laravel project's root directory. The file's capitalization is important, and it should not have an extension. Heroku reads this `Procfile` to determine what types of processes should launch when your application is deployed to one of their servers. In the case of a Laravel application we want to declare a `web` process type, identify the web server used to serve the application, and identify the application's document root directory, which in the case of Laravel is `public`. Therefore the `Procfile` should consist of the following single line:

```
1  web: vendor/bin/heroku-php-apache2 public
```

Incidentally, other options are available; see the Heroku PHP documentation[144] for more information about what's available.

After saving these changes to the newly created `Procfile`, you'll want to place your project under version control using Git:

---

[144]https://devcenter.heroku.com/articles/custom-php-settings#setting-the-document-root

```
1  $ git init
2  Initialized empty Git repository in /Users/wjgilmore/Software/dev.herokutest.com\
3  /.git/
4  $ git add .
5  $ git commit -m "First commit"
```

You'll want to use Git in particular because not only do all new Laravel projects come with some Git-specific features (.gitignore files in the appropriate directories, namely), but Heroku will also interact with your local Git repository to make deployment even easier than it otherwise would be. If you're not familiar with Git I suggest reading at least the first few chapters of "Pro Git"[145] (free to read online) and checking out the interactive Git tutorial at https://try.github.io/[146].

With your repository created, it's time to deploy! Use the Heroku Toolbelt to initialize a new Heroku project:

```
1  $ heroku create
2  Creating lit-retreat-6653... done, stack is cedar-14
3  https://lit-retreat-6653.herokuapp.com/ | https://git.heroku.com/lit-retreat-665\
4  3.git
5  Git remote heroku added
```

Note how this command created a new name for your application (in my case, lit-retreat-6653), and then identified a URL where the application can be accessed. If you head over to your project's URL now, you'll see a standard Heroku welcome placeholder.

Additionally, it created a Git "remote". A remote repository is simply a Git repository for your project that resides somewhere else. You can push changes to these repositories, and pull changes from them. In the case of Heroku we'll only ever push changes to the newly created Git remote. I'll show you how to push these changes in just a moment but first we need to make one quick configuration change. Namely, you need to tell Heroku what *buildpack* to use. Buildpacks tell Heroku more about the software that should be configured on the server when your application is installed. You can do so using the Heroku Toolbelt's config:add command:

```
1  $ heroku config:add \
2  > BUILDPACK_URL=https://github.com/heroku/heroku-buildpack-php
3  Setting config vars and restarting lit-retreat-6653... done, v5
4  BUILDPACK_URL: https://github.com/heroku/heroku-buildpack-php
```

Incidentally, if you're managing multiple applications in Heroku you'll additionally need to specify the application name using the --app flag.

Finally, it's time to deploy! You can push your local changes to this remote by executing the following command:

---

[145]https://progit.org/
[146]https://try.github.io/

```
 1  $ git push heroku master
 2  Counting objects: 5, done.
 3  Delta compression using up to 4 threads.
 4  Compressing objects: 100% (5/5), done.
 5  Writing objects: 100% (5/5), 416 bytes | 0 bytes/s, done.
 6  Total 5 (delta 4), reused 0 (delta 0)
 7  remote: Compressing source files... done.
 8  remote: Building source:
 9  remote:
10  remote: -----> Fetching custom git buildpack... done
11  remote: -----> PHP app detected
12  remote: -----> No runtime required in composer.json, defaulting to PHP 5.6.5.
13  remote: -----> Installing system packages...
14  ...
15  remote: -----> Launching... done, v6
16  remote:        https://lit-retreat-6653.herokuapp.com/ deployed to Heroku
17  remote:
18  remote: Verifying deploy... done.
19  To https://git.heroku.com/lit-retreat-6653.git
20     4ece26b..938feb8  master -> master
```

Congratulations! Your application has been deployed. Head on over to your designated URL and you should see the default Laravel splash page.

> The URL generated when you created the Heroku application is just for testing purposes; you can easily swap it out with a custom domain. See the Heroku documentation for more details.

Migrating Your Database

If you've been closely following along and deployed a brand new Laravel application, then presumably you successfully saw the default splash page load to your designated Heroku URL. However, if your project is backed by a database, then you'll additionally need to at a minimum ensuring that any outstanding migrations are executed following deployment. However, if this is your first interaction with Heroku in the context of the new application, you'll need to provision the database, which you can do with the Heroku Toolbelt:

```
1  $ heroku addons:add heroku-postgresql:hobby-dev
2  Adding heroku-postgresql:hobby-dev on lit-retreat-6653...
3  done, v8 (free)
4  Attached as HEROKU_POSTGRESQL_NAVY_URL
5  Database has been created and is available
6   ! This database is empty. If upgrading, you can transfer
7   ! data from another database with pgbackups:restore.
8  Use `heroku addons:docs heroku-postgresql` to view documentation.
```

This command creates a new PostgreSQL database. Specifically, this database is identified by the plan *Hobby Dev*, which is free but has some significant limitations (notably a limit of 10,000 rows). If you are interested in using Heroku for any long term project I strongly suggest carefully learning more about the various PostgreSQL plans here[147].

If you're wondering why I chose to create a PostgreSQL database rather than for instance a MySQL database, it's because Heroku doesn't support MySQL out of the box. However, Heroku *does* support MySQL. Although Heroku does indeed prominently feature its PostgreSQL support, you can in fact use MySQL via the ClearDB[148] addon. However for reasons of convenience I'll stick to using PostgreSQL in this section if for any other reason because you'll find the majority of Heroku's documentation tends to be PostgreSQL-centric.

With the database created, execute the following command to learn more about your database's access credentials:

```
1  $ heroku config --app lit-retreat-6653 | grep DATABASE_URL
2  DATABASE_URL: postgres://USERNAME:PASSWORD@HOSTNAME:PORT/DATABASE
```

In the command output I've swapped out my access credentials with placeholders so you can easily identify the constituent parts. However, you don't actually need to write these down, because the DATABASE_URL variable is automatically stored in your server's configuration settings. In order to transparently manage your database configuration variables in both the development and production environments, you could save yourself quite a bit of hassle by using PostgreSQL locally and saving an identically-formatted environment variable within your local environment.

> Obviously you'll also need to install and configure PostgreSQL within your local environment if it's not already available. See http://www.postgresql.org/[149] for installation instructions.

Exactly how you'll do this will depend upon your particular operating system, so consult the appropriate online documentation for more details. However, once the local environment variable

---

[147]https://addons.heroku.com/heroku-postgresql

[148]https://devcenter.heroku.com/articles/cleardb

[149]http://www.postgresql.org/

is in place there are a variety of ways you can reference it within your code. One of the most straightforward ways involves parsing the variable as a URL using PHP's `parse_url()` function directly within the `config/database.php` file. Also, you'll need to set the database default to `pgsql`:

```
1  'default' => 'pgsql',
2
3  ...
4
5  'pgsql' => [
6      'driver'   => 'pgsql',
7      'host'     => parse_url(getenv("DATABASE_URL"))["host"],
8      'database' => substr(parse_url(getenv("DATABASE_URL"))["path"], 1),
9      'username' => parse_url(getenv("DATABASE_URL"))["user"],
10     'password' => parse_url(getenv("DATABASE_URL"))["pass"],
11     'charset'  => 'utf8',
12     'prefix'   => '',
13     'schema'   => 'public',
14 ],
```

Save the changes and consider creating a model and corresponding migration to confirm you're able to properly connect to the new PostgreSQL database. After doing so, commit your changes and push them to Heroku:

```
1  $ git add .
2  $ git commit -m "Updated database configuration"
3  $ git push heroku master
```

Next, you'll want to migrate the database. You can easily do this using the Heroku Toolbelt's `run` command:

```
1  $ heroku run php artisan migrate --app lit-retreat-6653
2  Running `php artisan migrate` attached to terminal... up, run.6981
3  *************************************
4  *      Application In Production!      *
5  *************************************
6
7  Do you really wish to run this command? [y/N] y
8  Migration table created successfully.
9  Migrated: 2014_10_12_000000_create_users_table
10 Migrated: 2014_10_12_100000_create_password_resets_table
11 Migrated: 2015_01_30_032004_create_todolists_table.php
```

Your migrations are now in place!

Introducing Laravel Forge

Like Heroku, Laravel Forge (https://forge.laravel.com/[150]) is a PaaS (Platform as a Service) founded and run by none other than Laravel creator Taylor Otwell. Laravel Forge aims to eliminate the many manual steps the typical Laravel developer would otherwise have to take in order to properly deploy and maintain a Laravel application. Among other features, Laravel Forge offers:

- Server Consistency: If you're using Homestead (introduced in Chapter 1), you'll have the benefit of deploying to an identical server environment, as all Forge servers are consistent with what's found in the Homestead virtual machine (Ubuntu 14.04, PHP 5.6, etc.).
- Push-based Deployment: You'll tell Laravel Forge where your project repository resides (GitHub, BitBucket, etc.) and when you're ready to deploy Laravel Forge will retrieve the respository and deploy it to the designated server.
- Automated Monitoring: Laravel Forge users have the option of using the popular New Relic[151] and Papertrail[152] monitoring agents.
- Simple Scheduling: If you're using Laravel Queues[153] (not currently discussed in this book but forthcoming in a future update) or other scheduled jobs, you'll be able to use Laravel Forge's web interface for configuring these jobs rather than battle with Cron or other scheduling tools.

Keep in mind Laravel Forge is used *in conjunction with* another hosting service such as Linode[154] or DigitalOcean[155] (both of which I've incidentally used in the past and highly recommend). So at a minimum you'll pay $10/month for Laravel Forge (or less if you purchase an annual subscription) in addition to the fees at one of the supported hosting services. However a Digital Ocean plan[156] costs as little as $5, you can get started using Laravel Forge and a hosting provider for just $15/month and save bunches of time and tears you would have otherwise spent configuring mundane server tasks in the process.

I actually haven't yet had the opportunity to use Laravel Forge, but plan on doing so in the near future. When I do I'll be sure to thoroughly document the deployment process and update this chapter. In the meantime I suggest having a look at Matt Stauffer's excellent and thorough summary over on his blog[157].

Placing Your Application in Maintenance Mode

You'll occasionally need to perform a somewhat more lengthy maintenance window which requires the site to be offline for a few minutes or (worse) hours. During this time you won't want visitors

---

[150]https://forge.laravel.com/

[151]http://newrelic.com/

[152]https://papertrailapp.com/

[153]http://laravel.com/docs/master/queues

[154]https://www.linode.com/

[155]https://www.digitalocean.com/

[156]https://www.digitalocean.com/pricing/

[157]http://mattstauffer.co/blog/getting-your-first-site-up-and-running-in-laravel-forge

accessing the site and so you should put it in maintenance mode. To place your application in maintenance mode you'll execute Artisan's `down` command:

```
1  $ php artisan down
2  Application is now in maintenance mode.
```

At this point you can proceed with your system upgrade. Once the maintenance is complete, don't forget to bring the application back online using the `up` command:

```
1  $ php artisan up
2  Application is now live.
```

Obviously running either of these commands locally will only result in toggling your local, development application's maintenance status, therefore be sure to run the command on your production server in order to achieve the desired result. For instance to enable maintenance mode on Heroku (see the earlier section regarding deploying to Heroku for more context) you'd run the following command:

```
1  $ heroku run php artisan down --app lit-retreat-6653
```

Summary

While it's always fun to imagine and create new features, always keep in mind that nothing is real until you actually ship the application to the world. In order to do so in the most effective way possible you'll need to establish and implement a rigorous deployment process, and continually refine that process over time. In doing so, you'll be able to more rapidly and effectively respond to your users' needs, not to mention save your sanity.

Chapter 9. Creating a Restricted Administration Console

Many applications require a certain level of ongoing monitoring and maintenance beyond the typical code-based improvements. For instance you might wish to add new categories or edit existing names and descriptions, view a comprehensive list of registered users, or keep tabs on list creation and interaction trends. Such tasks should logically be accessible only by project administrators, yet be conveniently accessible. One effective way to integrate these capabilities is via a restricted web-based administration console. In this chapter I'll show you a particularly simple yet effective solution for creating such a console.

Identifying Administrators

There are several third-party packages one can use to add role-based permissions to a Laravel 5 application, including perhaps most notably Entrust[158]. However if your goal is to simply separate typical users from administrators, then a much more simple solution is available. You'll want to add a new column to the users table named something like `is_admin`. This Boolean column will identify administrators by virtue of being set to `true`. Go ahead and create the migration now:

```
1  $ php artisan make:migration add_is_admin_to_user_table --table=users
2  Created Migration: 2015_04_01_160821_add_is_admin_to_user_table
```

Next, open the newly created migration file and modify the `up` and `down` methods to look like this:

```php
1  public function up()
2  {
3          Schema::table('users', function(Blueprint $table)
4          {
5          $table->boolean('is_admin')->default(false);
6          });
7  }
8
9  public function down()
10 {
11         Schema::table('users', function(Blueprint $table)
```

---

[158]https://github.com/Zizaco/entrust

```
12            {
13                    $table->dropColumn('is_admin');
14            });
15    }
```

After saving these changes, run the migration:

```
1    $ php artisan migrate
2    Migrated: 2015_04_01_160821_add_is_admin_to_user_table
```

After running the migration, all existing users will have their `is_admin` column set to `false` (the default as defined in the migration). Therefore to identify one or more users as administrators you'll need to login to your database and set those users' `is_admin` columns to `true`. For instance if you're using the `mysql` client you can login to the client and run the following command:

```
1    mysql> update users set is_admin = true where email = 'wj@wjgilmore.com';
```

Creating the Administration Controllers

Next we'll create an administration controller. In reality you'll likely wind up with several controllers which are collectively identified as being administrative in nature, so you can create a convenient *route grouping* which places them all under a *route prefix* and namespace. Create your first such controller by executing the following command:

```
1    $ php artisan make:controller Admin/UserController
2    Controller created successfully.
```

This `make:controller` command is a bit different from the others you've executed so far throughout the book because we are *prefixing* the controller name with a directory name. In doing so, a directory named `Admin` was created inside `app/Http/Controllers`, and inside `Admin` you'll find the `UserController.php` directory. We'll use this controller to list and manage registered users. Next let's create the route grouping which identifies both the URI prefix and the namespace:

```
1    Route::group(['prefix' => 'admin', 'namespace' => 'admin'], function()
2    {
3        Route::resource('user', 'UserController');
4    });
```

Note the user of `Route::group`. This allows you to nest controller inside the definition block without redundantly declaring the prefix and namespace. So for instance at some time in the future you might have three or four administrative controllers. You can follow the same approach used to create the `User` controller, and then add them to `routes.php` like this:

```
1  Route::group(['prefix' => 'admin', 'namespace' => 'admin'], function()
2  {
3          Route::resource('Category', 'CategoryController');
4      Route::resource('List', 'ListController');
5      Route::resource('Product', 'ProductController');
6      Route::resource('user', 'UserController');
7  });
```

With this route definition in place, create a new directory named `admin` inside `resources/views`, and inside it create a directory named `user`. This will house the views associated with the new administrative `User` controller. Inside the `user` directory create a file named `index.blade.php` and add the following contents to it:

```
1   <h1>Registered Users</h1>
2
3   <ul>
4   @forelse ($users as $user)
5
6           <li>{{ $user->name  }} ({{ $user->email }})</li>
7
8   @empty
9
10          <li>No registered users</li>
11
12  @endforelse
13  </ul>
```

Finally, open the new `User` controller (`app/Http/Controllers/Admin/UserController.php`) and update the `index` action to look like this:

```
1  public function index()
2  {
3          $users = User::orderBy('created_at', 'desc')->get();
4          return view('admin.user.index')->with('users', $users);
5  }
```

With these changes in place you should be able to navigate to `/admin/user` and see a bulleted list of any registered users! Of course, before deploying this to production you'll want to restrict access to only those users identified as administrators. Let's do this next.

Restricting Access to the Administration Console

We want to allow only those users identified as administrators (their users table record's is_admin column is set to true). You might be tempted to make this determination by embedding code such as the following into your controller actions:

```
1  if (Auth::user()->is_admin != true) {
2          return \Redirect::route('home')->with('message', 'Access denied!');
3  }
```

Don't do this! This is a job perfectly suited for custom middleware. Let's create a middleware which neatly packages this sort of logic, and then associate that middleware with our administrative controllers:

```
1  $ php artisan make:middleware AdminAuthentication
2  Middleware created successfully.
```

This command created a new middleware skeleton named AdminAuthentication.php which resides inside app/Http/Middleware. Open this file and update it to look like the following (changes emphasized):

```
1  <?php namespace Todoparrot\Http\Middleware;
2
3  use Closure;
4  use Illuminate\Contracts\Auth\Guard;
5  use Illuminate\Http\RedirectResponse;
6
7  class AdminAuthentication {
8
9          protected $auth;
10
11         public function __construct(Guard $auth)
12         {
13                 $this->auth = $auth;
14         }
15
16         public function handle($request, Closure $next)
17         {
18                 if ($this->auth->check())
19                 {
20                         if ($this->auth->user()->is_admin == true)
```

```
21                              {
22                                      return $next($request);
23                              }
24                      }
25
26                      return new RedirectResponse(url('/'));
27
28              }
29
30 }
```

Most of this should be familiar by now given coverage of authentication and middleware in chapters 5 and 6, respectively, so I won't belabor the changes. Save this file and then open `app/Http/Kernel.php` and register the middleware inside the `$routeMiddleware` array:

```
1 protected $routeMiddleware = [
2         ...
3         'admin' => 'Todoparrot\Http\Middleware\AdminAuthentication',
4 ];
```

With the middleware registered, all that remains is to associate the middleware with the route group:

```
1 Route::group(
2         [
3             'prefix' => 'admin',
4             'namespace' => 'admin',
5             'middleware' => 'admin'
6         ], function()
7         {
8             Route::resource('user', 'UserController');
9         });
```

Once you've saved this change to the `routes.php` file, your administrative controllers will be restricted to administrators!

Summary

Hopefully this brief chapter adequately demonstrated just how easy it is to create a restricted administrative console for your Laravel applications. Of course, if you require more sophisticated role-based features then definitely check out a package such as Entrust[159] however for more simplistic requirements I certainly suggest embracing this straightforward approach!

---

[159]https://github.com/Zizaco/entrust

Chapter 10. Introducing Lumen

I've spent the past nine chapters and more than 200 pages praising Laravel's virtues. The framework offers countless useful features, is endlessly configurable, and benefits from an enormous ecosystem of third-party packages.

But what if you didn't *need* all of these capabilities? What if your primary priority was performance above everything else? This role has historically been filled by *microframeworks*. Microframeworks are effectively stripped down variants of the so-called enterprise frameworks such as Laravel, Symfony, and Zend Framework. Although lacking many of the features packaged into these larger solutions, microframeworks counterbalance their relatively few capabilities by offering superior performance in addition to generally being easier to learn.

Fortunately, Laravel developers can now take advantage of an incredibly fast microframework that is nonetheless decidedly Laravelish in nature. It's called Lumen[160], and it was first released by Laravel creator Taylor Otwell in April, 2015. Lumen offers many of the very same features also available to Laravel, and in fact a quick glance of the code powering a Lumen application might not be suffice to tell the difference!

So for what sort of applications might you choose to use Lumen instead of its larger sibling? Again, generally speaking it might come into consideration anytime performance becomes a priority. For instance if a particular area of your web application is used with much greater frequency than other areas and it's beginning to degrade overall performance, then you might consider rebuilding that popular feature using Lumen and integrating it alongside the Laravel application. Alternatively, if you were building a REST API then you might consider using Lumen since the service likely won't require features such as Blade templating or Eloquent, two capabilities not enabled in Lumen by default (but are nonetheless available should you need them, as will be discussed later in this chapter).

In this chapter we'll build a simple REST API for the TODOParrot application. Specifically, we'll create a small microservice[161] which is responsible for retrieving a few simple system statistics, returning them in JSON format. You could then consume this service within multiple venues, such as the TODOParrot website and maybe an iPhone application used by an administrator for monitoring the system. This sort of solution is precisely where Lumen's strengths lie given the framework's speed and bare bones feature set.

> You'll find all of the source code used to power the sample Lumen application discussed in this chapter over on GitHub at https://github.com/wjgilmore/status.todoparrot.com[162].

---

[160]http://lumen.laravel.com/

[161]http://martinfowler.com/articles/microservices.html

[162]https://github.com/wjgilmore/status.todoparrot.com

Creating Your First Lumen Application

You'll generate Lumen applications in much the same way you learned how to generate Laravel applications back in Chapter 1. Begin by installing the Lumen package:

```
1  $ composer global require "laravel/lumen-installer=~1.0"
```

Once installed you're ready to create your first Lumen application:

```
1  $ lumen new status.todoparrot.com
2  Crafting application...
3  Application ready! Build something amazing.
```

Next, you'll want to copy `.env.example`, creating `.env`. For the purposes of this example I've updated the `.env` `DB_*` parameters to use the same settings as those used for my local instance of the TODOParrot web application, thus giving the Lumen application access to the same database. With the `.env` file created, you'll next need to enable the `phpdotenv` library by opening the `bootstrap/app.php` file and uncommenting the following line:

```
1  Dotenv::load(__DIR__.'/../');
```

After saving these changes, return to the terminal and launch the application just to ensure everything is working properly. For sake of demonstration I'll launch it using the built-in PHP server:

```
1  $ php artisan serve
```

Open your browser, navigate to `http://localhost:8000`, and you should see the splash page presented in the following screenshot.
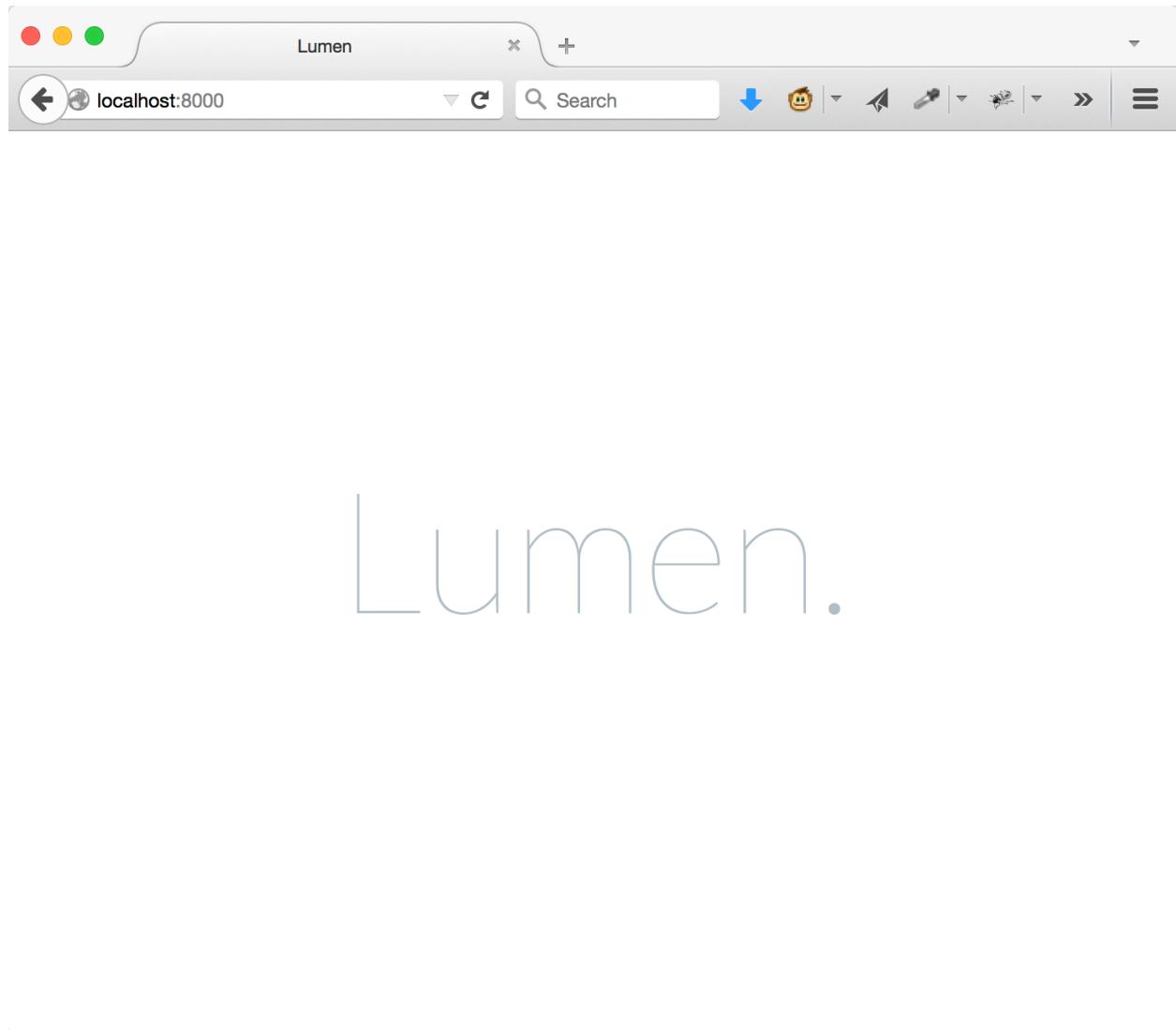
The default Lumen application homepage

Once you've confirmed the application is up and running, return to the terminal and execute php artisan list to view the list of available Artisan commands:

```
1  $ php artisan list
2  ...
3  cache
4   cache:clear        Flush the application cache
5   cache:table        Create a migration for the cache database table
6  db
7   db:seed            Seed the database with records
8  make
9   make:migration     Create a new migration file
```

```
10   migrate
11    migrate:install       Create the migration repository
12    migrate:refresh       Reset and re-run all migrations
13    migrate:reset         Rollback all database migrations
14    migrate:rollback      Rollback the last database migration
15    migrate:status        Show the status of each migration
16                          queue
17    queue:failed          List all of the failed queue jobs
18    queue:failed-table    Create a migration for the failed queue jobs
19                          database table
20    queue:flush           Flush all of the failed queue jobs
21    queue:forget          Delete a failed queue job
22    queue:listen          Listen to a given queue
23    queue:restart         Restart queue worker daemons after their current job
24    queue:retry           Retry a failed queue job
25    queue:subscribe       Subscribe a URL to an Iron.io push queue
26    queue:table           Create a migration for the queue jobs database table
27    queue:work            Process the next job on a queue
28   schedule
29    schedule:run          Run the scheduled commands
```

Notice this list is significantly smaller than that available to the typical Laravel application. Notably, almost all of the make commands are missing, meaning it's not even possible at this time to automate the generation of a controller. Frankly I'm not quite sure why the generators aren't available by default to Lumen application, however manually generating skeletons for controllers or models isn't really a big deal, as you'll see in a moment.

Creating a Status API

We'll create a RESTful API for retrieving various usage statistics. Begin by creating a new file named StatusController.php inside app/Http/Controllers, and adding the following contents to it:

```php
1   <?php namespace App\Http\Controllers;
2
3   use App\Http\Requests;
4   use App\Http\Controllers\Controller;
5   use Illuminate\Http\Request;
6
7   class StatusController extends Controller {
8
9       public function index()
```

```
10              {
11                      return response()->json(['status' => 'Polly wants a cracker!']);
12              }
13
14      }
```

Laravel's response helper is very useful when you'd like to return JSON because it will automatically set the Content-Type header to application/json. Properly setting this header is crucial in order to ensure clients can properly retrieve and parse the returned JSON, and therefore taking advantage of response in order to ensure this important (but easy to forget) matter is handled.

Next, open the app/Http/routes.php file and add the following route declaration:

```
1   $app->get('/status', 'App\Http\Controllers\StatusController@index');
```

As you can see this route declaration differs quite significantly from the declarations found in a typical Laravel application. Most notably, you need to define the path to the route's associated controller and action, because Lumen is not going to autoload these controllers in order to avoid the performance cost of doing so.

After saving these changes, return to the browser and navigate to /status, and you should see the following JSON output to the browser window:

```
1   {"status":"Polly wants a cracker!"}
```

Incidentally, I tend to avoid interacting with JSON-based APIs in this manner during the development phase, instead using a Chrome extension called Postman[163]. Postman was created expressly for this purpose, allowing you to easily view JSON data and headers, set URL parameters, manage authentication settings, and conveniently save endpoints for later reference. Here's a screenshot demonstrating Postman being used to access the /status endpoint:

---

[163]https://www.getpostman.com
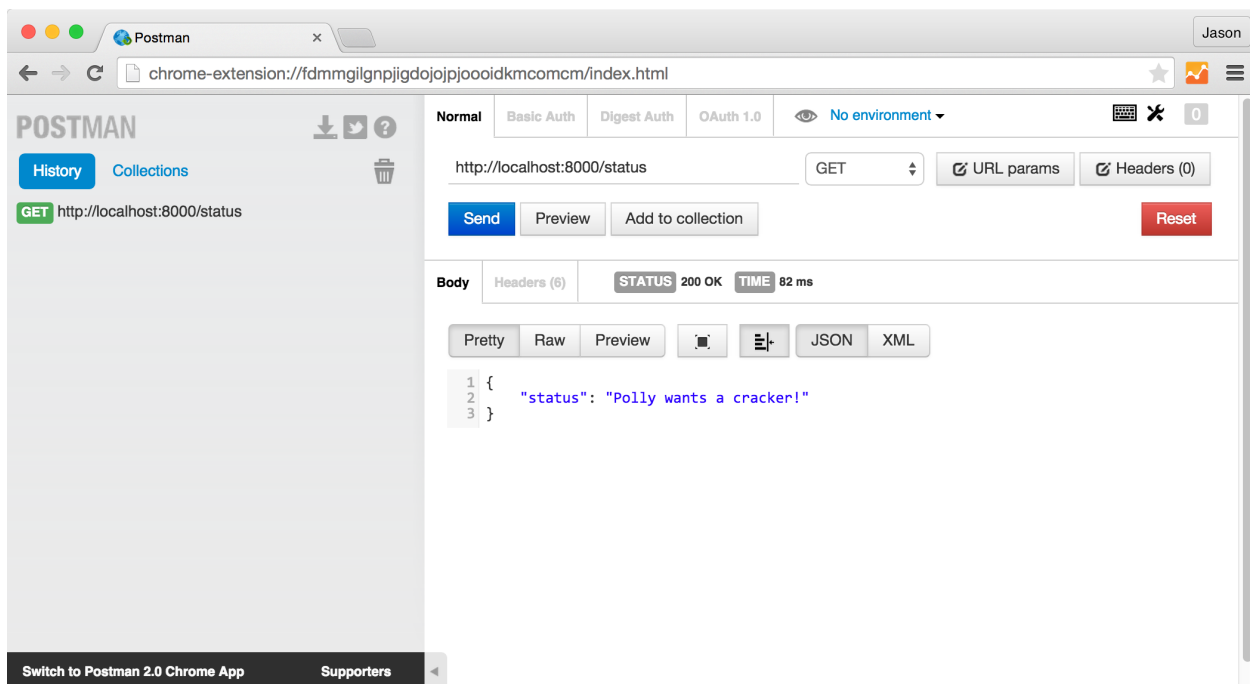
Viewing a JSON endpoint using Postman

Talking to the Database

The previous `/status` endpoint was intended to highlight a few differences between Lumen and Laravel such as controller/action creation and route declarations; otherwise it is obviously pretty useless. So let's add a new endpoint which will return the total number of TODO lists created to date. Although you are already well aware how to talk to the database in a Laravel application, and have the option of doing so in the very same fashion within a Lumen application, there are a few additional configuration-related steps you'll need to take in order to do so.

Let's begin by making the `DB_*` configuration settings found in the `.env` file available to your application. Although these settings are already available in `.env`, a Lumen application doesn't yet know what to do with them! And here you'll encounter the first notable departure from the standard Laravel framework; neither the `database.php` configuration file nor for that matter the `config` directory exists by default in a Lumen application! Lumen does however *support* these files, so let's bring `database.php` into the picture by first creating the `config` directory inside your project's root directory:

```
1  $ mkdir config
```

Next, copy the `database.php` directory from `vendor/laravel/lumen-framework/config/`, placing the copy inside the newly created `config` directory:

```
1  $ cp vendor/laravel/lumen-framework/config/database.php config/
```

With the `config` directory and `app.php` file in place, Lumen will automatically begin incorporating the `.env` file's `DB_*` settings into your application!

Next you'll want to open open `bootstrap/app.php` and uncomment the following line:

```
1  $app->withFacades();
```

Uncommenting this line will allow you to use the `DB` facade and therefore the Laravel Query Builder syntax. With these configuration changes in place we can create a new action and retrieve the list count. Add the following action to the `Status` controller:

```
1  public function lists()
2  {
3
4      $result = \DB::select('select count(id) as `count` from todolists');
5
6      return response()->json(['count' => $result[0]->count]);
7  }
```

Save these changes and then open the `app/Http/routes.php` file and add the following route definition:

```
1  $app->get('/lists/count', 'App\Http\Controllers\StatusController@lists');
```

Keep in mind you are certainly free to add multiple controllers to a Lumen application! Because this microservice example is intended to be very focused, I'm just being a bit naughty here and creating a convenience route (`/lists/count`) that obviously doesn't abide by standard route naming conventions. After saving these changes, return to your browser (or preferably Postman or a similar tool), and navigate to `/lists/count`. You should output similar to the following (formatted for readability):

```
1  {
2      "count": 643
3  }
```

Incidentally, it's also possible to use Eloquent, although you'll need to first uncomment the following line from `bootstrap/app.php`:

```
1  $app->withEloquent()
```

Additionally you'll have to create companion models within your Lumen application. I suggest sticking to the DB facade whenever possible as not only will the application perform slightly faster but you additionally won't have to deal with potentially redundant code (similar or identical to that found in a companion Laravel application).

Integrating the Lumen Application Into TODOParrot.com

After having created and deployed the microservice, it's time to incorporate it into the TODOParrot website. If you head over to the TODOParrot.com homepage, you'll see the homepage includes a reference to the total number of lists created (directly below the button found in the middle of the page). This information is retrieved in real-time from the microservice! I use a simple jQuery-driven AJAX call to talk to the Lumen /lists/count endpoint. Because this JavaScript should only execute on the home page, I added a @yield statement to the layouts/master.blade.php file:

```
1  @yield('footer_js')
```

Next, in welcome.blade.php, I added the following:

```
1  @section('footer_js')
2  <script>
3    $.ajax({
4      url: "http://status.todoparrot.com/lists/count",
5      dataType: 'jsonp',
6      crossDomain: true,
7      success: function(data) {
8        $('#list_count').html(data.count + " lists created!");
9      },
10     error: function(data) {
11       $('#list_count').html('Squawk!');
12     }
13   });
14  </script>
15  @endsection
```

Keep in mind for the purposes of this demonstration I've created a *cross-domain* Ajax request, and therefore the Lumen response varies slightly from that presented earlier in this chapter in order to accommodate the callback wrapper required by JSONP-oriented responses. This is trivial to add

within Laravel/Lumen applications, and you'll see how it's done in the example microservice source code found on GitHub[164]

If your microservice exposes potentially sensitive information, then obviously an approach such as the above isn't going to be secure. You'll need to research solutions for properly securing an API and how credentials are managed when working with JavaScript-based clients. Perhaps I'll expand upon these matters in a future iteration of this chapter but for now I just wanted to make some Lumen-related information available to readers!

Summary

Hopefully this brief introduction to the Lumen microframework served to get your mind racing regarding new ways to increase application performance, not to mention new microservice possibilities. The Laravel team has per usual put together a pretty comprehensive set of documentation[165] so I suggest having a thorough look at it in order to gain an even deeper understanding of what's available via this amazingly fast new solution!

---

[164]https://github.com/wjgilmore/status.todoparrot.com

[165]http://lumen.laravel.com/docs

Chapter 11. Introducing Events

Your project's success can be quantified by usage metrics such as new registrations, user activity, and conversions (purchases of upgrades or other products). In order to stay abreast of these metrics you might wish to receive an e-mail, update an analytics dashboard, or send a message to a group chat application such as Slack[166].

How might you be notified of these milestones? Adding custom logic to the associated controller actions seems like the obvious solution, however such an approach violates the convention of limiting an action's responsibility to one specific task. To illustrate the sort of problems which might arise from such haphazard expansion of a controller action's responsibilities, suppose you wanted to be notified every time a particular milestone is met (such as a new user registration), and so modify the associated action to send an e-mail every time the action is executed. Over time your list of desired notification metrics grows to include active users and conversions, and with it the number of tweaks to other actions. Eventually, your continued success forces a frenetic refactoring of these various notifications to eliminate the constant inbox flooding and ensure more efficient analysis of these milestones.

Fortunately Laravel 5 offers a much more attractive alternative to this common dilemma. Known as *events*, you can create easily maintainable bits of logic which can be triggered to execute in conjunction with a specific occurrence. In this chapter I'll walk you through the steps necessary to incorporate a custom event into your application. You'll also learn about a new Laravel 5.1 feature that allows you to *broadcast* events to all of the users currently interacting with the web application.

Creating an Event

Laravel events are actually comprised of two parts:

- **Event Handler**: The event handler contains the information associated with the event. For instance, if your goal is to trigger an event associated with the creation of a list, then that `List` object would be made available (along with any other information you require) to the event handler. This information will subsequently be made available to the event listener, introduced next.
- **Event listener**: The event listener "listens" for the event instance, and responds to it accordingly. It is here where you will implement the event implementation logic.

So in a nutshell, the event handler will be attached to the application logic you'd like to monitor, and the event listener will execute once the event handler is triggered (or *fired*, in Laravel nomenclature).

---

[166]http://slack.com

Both the event listener and event handler are defined within standard PHP classes, and are subsequently associated with one another using a simple declaration found in a configuration file. In this section I'll guide you through the process of creating a listener and handler, binding the two together, and then integrating the event into the TODOParrot application so that a log message is generated every time a new list is created. I'll keep the actual event logic simple because that's not really the point of this exercise; as you'll soon see it will be trivial to substitute this logic for a more practical outcome such as sending an e-mail or updating the database.

Defining the Event Handler

The event handler is responsible for initializing the event execution process, firing in conjunction with the application logic you desire to monitor. Let's create an event handler which will execute in conjunction with the creation of a new list. You can do so using Artisan's `event:generate` command:

```
1  $ php artisan make:event ListWasCreated
2  Event created successfully.
```

This command generated the event handler skeleton, which you'll find in `app/Events/ListWasCreated.php`. The file looks like this:

```php
1  <?php
2
3  namespace Todoparrot\Events;
4
5  use Todoparrot\Events\Event;
6  use Illuminate\Queue\SerializesModels;
7  use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
8
9  class ListWasCreated extends Event
10 {
11     use SerializesModels;
12
13     /**
14      * Create a new event instance.
15      *
16      * @return void
17      */
18     public function __construct()
19     {
20         //
21     }
22
```

```
23      /**
24       * Get the channels the event should be broadcast on.
25       *
26       * @return array
27       */
28      public function broadcastOn()
29      {
30          return [];
31      }
32  }
```

As you can see, this is just a standard PHP class that extends Laravel's `Event` class. The `Serial-izesModels` trait will help to more effectively manage any Eloquent models as they pass through the handler, and is likely not something you'll need to worry about when getting started using events. The `broadcastOn` method is new to Laravel 5.1, and we'll return to the purpose of this method later in the chapter.

The constructor is by default empty although at a minimum you'll want to pass along the `List` object created within the logic you plan on monitoring:

```
1  use Todoparrot\Todolist;
2
3  class ListWasCreated extends Event
4  {
5
6      ...
7
8      public $list;
9
10     public function __construct(Todolist $list)
11     {
12       $this->list = $list;
13     }
14
15     ...
16
17 }
```

This `$list` attribute will in turn be exposed to the event listener. Let's create that listener next.

Defining the Event Listener

The event listener contains the implementation logic which should be executed when the event is triggered. You can use Artisan to generate the listener skeleton, as demonstrated here:

```
1  $ php artisan make:listener LogMessageWhenListCreated \
2  > --event="ListWasCreated"
```

Note I've identified the name of the event handler created previously (`ListWasCreated`). This command generates the event listener skeleton, which you'll find in `app/Listeners/LogMessage-WhenListCreated.php`. The file looks like this:

```php
1  <?php
2
3  namespace Todoparrot\Listeners;
4
5  use Todoparrot\Events\ListWasCreated;
6  use Illuminate\Queue\InteractsWithQueue;
7  use Illuminate\Contracts\Queue\ShouldQueue;
8
9  class LogMessageWhenListCreated
10 {
11
12     public function __construct()
13     {
14         //
15     }
16
17     public function handle(ListWasCreated $event)
18     {
19         //
20     }
21 }
```

The `handle` method will contain the logic you'd like to execute when the event is triggered. Notice how the `ListWasCreated` event object is passed into the method by default. You'll use this object to access the `List` object assigned in the handler's constructor. Update the constructor to look like this:

```php
1  public function handle(ListWasCreated $event)
2  {
3      \Log::info("LIST CREATED {$event->list->name}");
4  }
```

Again, we're just executing a trivial log message to demonstrate that indeed the `List` object is made available to the listener. You could instead send an e-mail, notify a web service, or do whatever else you please here.

Associating the Events

With the event and event listener created, it's time to associate them so Laravel knows to trigger the event logic when the listener detects the desired action. You'll do so within app/Providers/EventServiceProvider.php. Open this file and you'll find the following protected $listen attribute:

```
1  protected $listen = [
2      'event.name' => [
3          'EventListener',
4      ]
5  ];
```

The lone entry in this array is just an example, so you can go ahead and replace it with our new event binding:

```
1  protected $listen = [
2      'Todoparrot\Events\ListWasCreated' => [
3          'Todoparrot\Listeners\LogMessageWhenListCreated'
4      ]
5  ];
```

After saving these changes you're ready to integrate the event into your application.

Integrating the Event Into Your Application Flow

Triggering an event is incredibly easy, done using the Event::fire method. You'll pass a new instance of the event handler you'd like to execute into this method, and inside it pass the object you'd like its constructor to receive. Here's an example in which the ListWasCreated handler is executed within the Lists controller's store action:

```
1   use Event;
2   use Todoparrot\Events\ListWasCreated;
3
4   ...
5
6   public function store(ListCreateFormRequest $request)
7   {
8
9       ...
10
11      Event::fire(new ListWasCreated($list));
12
13  }
```

With the event handler in place, the next time a new list is created you'll find a log message similar to the following has been added to the log file:

```
1  [2015-06-29 18:59:31] local.INFO: LIST CREATED Groceries for Dinner
```

Binding Multiple Listeners to an Event Handler

It's certainly possible that you might wish to bind multiple listeners to an event handler. For instance, you might wish to both receive and e-mail and write a log message whenever a new list is created. Doing so is trivial; just create all of the necessary event listeners and then bind them like so inside `EventServiceProvider.php`:

```
1  protected $listen = [
2      'Todoparrot\Events\ListWasCreated' => [
3        'Todoparrot\Listeners\LogMessageWhenListCreated',
4        'Todoparrot\Listeners\LogAnotherMessageWhenListCreated'
5      ]
6  ];
```

Binding Events to the Model Lifecycle

If your sole intent is to trigger an event in conjunction with the creation, update, or deletion of a model, there's an even more streamlined approach you might consider. Whenever Laravel creates, updates, saves, deletes, or restores a model, it fires an event. You can piggyback atop these events, triggering your own custom events in kind. This is done within the `app/Providers/AppService-Provider.php`'s `boot` method:

```
1  class AppServiceProvider extends ServiceProvider {
2
3      public function boot()
4      {
5          //
6      }
7
8      ...
9
10 }
```

For instance, if you'd like to trigger the `ListWasCreated` event handler when a new list is created, you can update the `boot` method like so:

```
1   use Event;
2   use Todoparrot\Events\ListWasCreated;
3   use Todoparrot\Todolist;
4
5   ...
6
7   public function boot()
8   {
9       Todolist::created(function ($list) {
10          Event::fire(new ListWasCreated($list));
11      });
12  }
```

After saving the changes, create a new list and you'll see that indeed another message is appended to the log.

Broadcasting Events

Laravel 5.1 introduced the ability to *broadcast* events to all users currently interacting with the application. This is in my opinion one of the most interesting Laravel features as it opens up countless new ways to communicate with your users in a really compelling fashion. For instance, the below screenshot presents an example of a broadcast notification sent out to all users after a user created a new list.
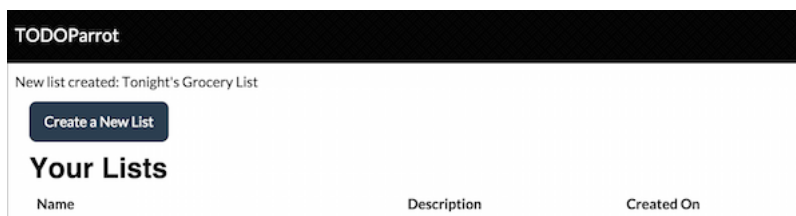
Broadcasting events

In this section I'll show you how to refactor the above list creation event to notify all users whenever a new list is created.

Configuring Broadcasting

To begin, we'll need to make a few configuration-related changes. All Laravel 5.1+ applications include a new configuration file named `config/broadcasting.php`. The default file looks like this:

```php
1   <?php
2
3   return [
4
5       'default' => env('BROADCAST_DRIVER', 'pusher'),
6
7       'connections' => [
8
9           'pusher' => [
10              'driver' => 'pusher',
11              'key' => env('PUSHER_KEY'),
12              'secret' => env('PUSHER_SECRET'),
13              'app_id' => env('PUSHER_APP_ID'),
14          ],
15
16          'redis' => [
17              'driver' => 'redis',
18              'connection' => 'default',
19          ],
20
21          'log' => [
22              'driver' => 'log',
23          ],
24
25      ],
26
27  ];
```

Laravel currently supports three broadcasting drivers, including Pusher[167], Redis[168] (which allows you to use Redis' pubsub capabilities to interact with a solution such as Socket.io[169]), and a logging driver for testing purposes. Because this is for demonstration purposes I'll be using Pusher's free Sandbox plan. If you'd like to experiment with broadcasting, I suggest doing the same as it is the easiest to configure. To do so, head over to https://pusher.com/[170] and create a free account. After doing so, sign into your account and create a new app via the dashboard. You'll be provided with an application ID, public token, and secret token. Next, add the PUSHER_KEY, PUSHER_SECRET, and PUSHER_APP_ID configuration variables to your local .env file, and assign each the appropriate value. Don't forget you'll additionally want to make these configuration settings available to your production environment prior to deployment.

---

[167]https://pusher.com/

[168]http://redis.io/

[169]http://socket.io/

[170]https://pusher.com/

Next you'll need to install two new Composer packages, including `pusher-php-server` and `predis`. Add the following two lines to your `composer.json` file's `require` key:

```
1  "require": {
2      ...
3      "pusher/pusher-php-server": "~2.0",
4      "predis/predis": "~1.0"
5  },
```

Next, you'll need to configure a *queue listener*, because broadcasting depends upon queues for performance reasons. I haven't yet touched upon the topic of queues in the book (this is slated for a forthcoming chapter), so I'm not keen on including an introduction to the matter here and for the time being am going to instead point you to the appropriate Laravel documentation[171].

To enable Laravel's queue support you'll need to update `config/queue.php`, identifying the queue driver you'd like to use. By default it is set to `sync`, which means Laravel will just bypass the queue altogether and execute the event normally. Several drivers are supported, although for the purposes of this example I'm going to use the `database` driver since it is arguably the easiest to configure. After setting the `queue.php` file's `default` key to `database`, open the terminal and execute the following Artisan command from within your project' root directory:

```
1  $ php artisan queue:table
2  Migration created successfully!
```

This creates a new database table which will be used to manage the queues. Next you'll want to run the newly created migration:

```
1  $ php artisan migrate
2  Migrated: 2015_06_30_161852_create_jobs_table
```

Finally, you'll need to start Laravel's queue listener. Open a new terminal tab and execute the following Artisan command from your project directory:

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

With your broadcasting driver configured and listener initiated, we're ready to broadcast events!

Broadcasting Events

Broadcasting events to your application users is a two-step process. First, you'll need to update the appropriate event handler's `broadcastOn` method. Returning to `app/Events/ListWasCreated.php`, you'll find this method at the bottom of the class:

---

[171]http://laravel.com/docs/master/queues

```
1  public function broadcastOn()
2  {
3      return [];
4  }
```

To enable broadcasting, you'll need to make two modifications to this class. First, you'll need to implement the ShouldBroadcast contract which is imported at the top of the class file. Second, you'll update the broadcastOn method to identify the Pusher *channel* which will broadcast the event information. Pusher channels are analogous to television channels; you'll broadcast whatever content you'd like via a channel, and then "tune in" to that channel on the client-side. We'll call our channel list-updates:

```
1   use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
2
3   class ListWasCreated extends Event implements ShouldBroadcast
4   {
5
6       ...
7
8       public function broadcastOn()
9       {
10          return ['list-updates'];
11      }
12  }
```

With the event handler changes in place, you'll need to add the necessary client-side code to the application. If you're using Pusher this is rather straightforward and is described in their documentation[172]. You'll begin by referencing the Pusher JS library in your project layout:

```
1   <script src="//js.pusher.com/2.2/pusher.min.js"></script>
```

Next, you'll create the JavaScript responsible for connecting to Pusher and retrieving the latest broadcasts. Believe it or not all that is required is a few simple lines of code. The following jQuery-enhanced snippet will connect to Pusher using your public key, subscribe to the list-updates channel, bind to the ListWasCreated event, and update a DIV named pusher with information about the newly created list name:

---

[172]https://pusher.com/docs/javascript_quick_start

```
1   $( document ).ready(function() {
2
3       var pusher = new Pusher('YOUR_PUBLIC_PUSHER_KEY');
4       var channel = pusher.subscribe('list-updates');
5
6       channel.bind('Todoparrot\\Events\\ListWasCreated', function(data) {
7         $('#pusher').html('New list created: ' + data.list.name);
8       });
9
10  });
```

Of course, you'll want to additionally add the pusher DIV to an appropriate location within your application layout. With these pieces in place, you're ready to begin broadcasting messages out to your users!

Troubleshooting Pusher

If your broadcasts aren't working as expected, beyond confirming you're using the correct public API key and ensuring the Pusher JS library is loading properly, consider adding the following JS code to your JavaScript client to confirm you are indeed connecting to the Pusher server:

```
1   pusher.connection.bind('connecting', function() {
2     alert('Connecting to Pusher...');
3   });
4
5   pusher.connection.bind('connected', function() {
6     alert('Pusher connection successful');
7   });
8
9   pusher.connection.bind('failed', function() {
10    alert('Pusher connection failed');
11  });
```

Additionally, be sure to monitor the Debug Console within the Pusher dashboard; this is supremely useful for determining whether Pusher is indeed receiving the broadcasts emitted from your event.

Summary

Events offer an elegant solution for executing important tasks alongside your domain logic without risk of polluting the code base. Once you begin using this powerful feature within your Laravel applications you'll wonder how you ever got along without it!

Appendix A. Deploying Your Laravel Application to DreamHost

> ⚠ This appendix was released on August 10, 2015 and should be considered a beta release. I'm concerned Windows users in particular might run into a few issues that are not discussed here, therefore if you run into any issues whatsoever (whether on Windows or not), please do e-mail me at wj@wjgilmore.com so I can update the material accordingly.

Sooner or later, you're going to want to make your Laravel creation available to the world. Fortunately, there are hundreds, if not thousands of web hosting providers perfectly capable of hosting your Laravel application. I've successfully deployed Laravel projects to multiple hosting providers, including Heroku and DreamHost. In Chapter 8 I even explained how to deploy your project to Heroku. However, Heroku isn't for everybody, and it can get pretty expensive quickly if your application requires significant resources. So in this Appendix I thought I'd offer instructions regarding deploying your application to DreamHost.

DreamHost is without a doubt my favorite shared web hosting provider. I've been a customer for almost 10 years now and have rarely experienced a problem with the service, and never anything significant. It's also very inexpensive, costing just $8.95 for an entry-level hosting solution that is perfectly suitable for hosting a Laravel application.

In this chapter I'll guide you through the steps required to deploy your project to DreamHost. We'll use the popular open source server automation and deployment software Capistrano[173], which can be a bit confusing at first but once you understand how it works you'll wonder how you ever lived without it. In addition to configuring Capistrano we'll need to complete some one-time server-side configuration tasks. Incidentally, although these instructions are indeed specific to DreamHost, chances are they'll be easily adapted for other shared hosting providers, although as always keep in mind your mileage may vary. :-)

> 🔑 If you're not already a DreamHost subscriber, and would like to sign up, use the code EASYLARAVELBOOK when signing up and you'll receive $50 off your purchase, meaning your first year monthly hosting fees will be only $5.78!

---

[173]http://capistranorb.com/

Deploying Your Project to DreamHost

Admittedly, configuring DreamHost (or frankly, many other shared hosting providers) to play with Capistrano can be a bit time-consuming the first time around, however many of these tasks only need to be completed once after which you'll be able to deploy subsequent projects in less than five minutes flat. I'll spend the rest of the chapter guiding you through both the one-time and repeating tasks, doing so in what I believe is the most natural order of completion. The tasks will be presented in this order:

1. **Managing Your Project Using Version Control**: Although this task could conceivably be presented later in the chapter, I wanted to discuss it immediately in case you aren't yet managing your project under version control and therefore needed to do some additional footwork (which I'll discuss in the later section).

2. **Configuring Your Domain**: Your application will of course be associated with some domain name. In this section I'll show you how to configure your DreamHost account to host that domain name and it's various files and other resources.

3. **Updating Your Domain Registrar DNS**: When users around the world navigate to your domain name, you'll want the DreamHost servers to respond in kind and serve up your application. In this section you'll learn how to configure your domain registrar's DNS settings to ensure this happens.

4. **Configuring Your DreamHost User Account**: Your default DreamHost user does not have shell access, however access can be easily enabled via the administration panel. In this section I'll show you how to do so.

5. **Updating the DreamHost PHP CLI Version**: When deploying project updates to the DreamHost server it's likely various Artisan commands will execute to carry out tasks such as database migrations. To do so you'll want your DreamHost account to have access to a recent PHP version for use on the command-line. In this section I'll show you how this is (easily) done.

6. **Installing Composer on DreamHost**: Each time your project deploys you'll want to ensure it has access to the latest dependencies. Because these dependencies are managed using Composer, you'll want to install Composer on DreamHost. As with updating the PHP CLI version this is easier than you think, and in this section you'll learn how.

7. **Configuring Passwordless Login**: Although not a requirement, being required to manually authenticate with DreamHost every time you'd like to deploy code updates is going to get tedious fast. By configuring passwordless login using SSH keys you can avoid this inconvenience altogether without compromising your account or server security. In this section I'll discuss the steps necessary to configure and test passwordless login.

8. **Installing and Configuring Capistrano**: With the server configured we'll install Capistrano and update your application's newly created Capistrano configuration files. In this section we'll work through the steps necessary to install and configure Capistrano for deployment purposes.

9. **Deploying Your Application**: The moment of glory has arrived! With the server and Capistrano configured, we'll deploy your application to the DreamHost servers. You'll also learn how to easily and seamlessly rollback a deployment to the previous version.

Managing Your Project Using Version Control

While not strictly required for deployment purposes, placing your project under version control is just an all around very wise idea. In fact, version control is so crucial to the success of your project that if you're not yet using it, I suggest putting off this chapter and instead learning more about the topic and the various version control solutions, including most notably Git[174]. You can find a succinct explanation of the merits of version control on the Tower website[175].

Placing your project under version control has a secondary benefit in regards to Capistrano in that it will help you to deploy your projects much more quickly than other available solutions. When configured accordingly, Capistrano will clone the repository on the production server and then use Git's native capabilities to only retrieve the latest changes which are then incorporated into the server-side repository. Through some subsequent magic that I'll discuss later in the chapter, this updated repository is then exported (or "archived") to a directory which will be subsequently used as the new website version.

Git is the most popular version control solution today, what I use on a daily basis, and is what Capistrano supports natively, so I'll use it for the relevant examples and discussion. I'll presume you've already initialized your project as a Git repository (e.g. `git init`) and have made at least one commit representative of the project version you'd like to deploy.

Additionally, although this isn't strictly necessary, it makes a lot of sense to be managing your project centrally on a project hosting service such as BitBucket[176] or GitHub[177]. If so, Capistrano can be configured to refer to the centrally hosted repository rather than your local repository, which can be very advantageous given multiple team members are probably committing changes to your project. If you would like to manage your project privately but would rather not incur a monthly expense just yet, keep in mind BitBucket offers a free account tier which includes private repositories. I happen to use both BitBucket and GitHub, and find both services to be indispensable. Later in the chapter I'll show you how to configure Capistrano to interact with your GitHub repository, although these instructions could easily be adapted for any other Git project hosting service.

Configuring Your Domain

Your deployed Laravel application will logically be accessible via a domain name. To do so, you'll need to identify this domain within the DreamHost panel, and then update your domain name registrar to point the domain to DreamHost's DNS servers. Let's begin with the former task. If you

---

[174]https://git-scm.com/
[175]http://www.git-tower.com/learn/git/ebook/mac/basics/why-use-version-control
[176]https://bitbucket.org/
[177]http://github.com

wanted to host your domain name `example.com`, you'll login to your DreamHost administration panel http://panel.dreamhost.com[178] and navigate to `Domains` > `Manage Domains`. There you'll find a button titled 'Add Hosting to a Domain / Sub-Domain'. Click that button and you'll be presented with the interface found in the following screenshot:

Hosting a Domain

In order to properly deploy your application in the fashion I outline in this chapter, it is *very important* for you to properly complete this form in the manner I describe here. I'll define the purpose of each field, and explain how you should define the associated value:

- **Domain to host**: Here you'll identify the domain name. Note you'll just specify it as `example.com` and not `www.example.com`.
- **Do you want www in your URL?**: This decision is entirely up to you, however keep in mind that for search engine optimization purposes you'll want to refer to the site exclusively as `www.example.com` or `example.com`, so I would suggest either selecting `Add WWW` or `Remove WWW`, and sticking with whichever convention you choose.

- **Run this domain under the user:**: For security reasons you can create a new owner for each hosted domain. This account would own the hosted files, and you would use that account for SSH'ing and SFTP'ing into the server to manage the domain files. For reasons of convenience I tend to manage my sites using the same user, however keep in mind that in doing so if that account were to be compromised then the attacker would have control over all of your hosted domain files.
- **Web directory**: This field is particularly important so pay very close attention. When you identified the domain in the `Domain to host` field, this web directory path was automatically updated to read like `/home/username/example.com`. Change this to read `/home/username/example.com/current/public`. Triple-check that you typed it exactly as I specify (replacing `example.com` with your domain of course), because if it is not exactly as specified you will run into problems later in this chapter.
- **Logs directory**: This identifies the location of your log files. It can't be changed.
- **PHP mode**: At the time of this writing DreamHost will by default use `PHP 5.4 FastCGI` to run your PHP-powered websites. I suggest changing this to the very latest available FastCGI version, which again at the time of this writing is `PHP 5.6 FastCGI`.
- **Automatically upgrade PHP**: I suggest enabling this option so you can always take advantage of the latest PHP features.
- **Extra web security**: This option enables the Apache mod_security module, which can protect your website from common exploits such as cross-site scripting and remote execution.
- **Page speed optimization**: This option enables the Apache mod_pagespeed module, which automatically applies optimization techniques such as image compression and CSS/JavaScript concatenation and minification. I recommend enabling this feature.
- **PHP XCache Support**: This feature can improve the performance of your PHP applications by caching compiled PHP code. It is however only available to DreamHost's Virtual Private Server customers.
- **Passenger (Ruby/NodeJS/Python apps only)**: This option only applies to Ruby, NodeJS, and Python applications so leave it disabled.

You can safely disregard the options found below the above, although I certainly encourage you to investigate their utility. Once you've completed the aforementioned form fields, press the `Fully host this domain` button to complete the process.

Updating Your Domain Registrar DNS

After pressing the `Fully host this domain` button the ensuing page will instruct you to update your domain registrar's DNS settings to point the domain name to DreamHost's DNS servers. The interface for doing so varies according to the registrar, but they are all pretty straightforward. For instance, the below screenshot presents an example of Namecheap's DNS editor for one of my domains:

Namecheap's DNS Editor

After saving these changes, it will take anywhere from a few minutes to a few hours for the updates to *propagate* to the world's root DNS servers. Once this happens, requests made to your domain will be ultimately handled by DreamHost. Of course, because you haven't yet deployed the application, should these changes propagate before you complete the remaining steps discussed in this chapter then users will be greeted with a default DreamHost splash page.

Incidentally, If you haven't yet purchased a domain name, I've been using Namecheap[179] for recent domain name purchases. They seem much more sane then some of the other domain name registrars out there, and offer a really nice management interface. If you search for "namecheap coupons" you'll find they offer a special coupon good every month which will save you some money on the purchase.

Configuring Your DreamHost User Account

Capistrano will interact with your server by way of a user account and (preferably) passwordless login (discussed in the later section, "Configuring Passwordless Login"). When configuring your domain you were prompted to identify a user when setting the field `Run this domain under the user`. Whether you used the default user account or created a new account, you'll need to update this user to enable shell access. This is done by signing into the DreamHost panel and navigating to

[179]https://www.namecheap.com/

Users > Manage Users, clicking the Edit link associated with the desired user, and then setting the User Type field to Shell user (see below screenshot).

Updating a User Account

Once done, you'll be able to SSH into your DreamHost server by supplying your account username and the domain name you had configured over the course of the previous two sections:

```
1  $ ssh wjgilmore@example.com
2  wjgilmore@example.com's password:
3
4  No mail.
5  Last login: Mon Aug 10 13:05:04 2015 from 70.60.34.46
```

With this step complete, it's time to move on to the next section!

Updating the DreamHost PHP CLI Version

Recall from the earlier section, "Configuring Your Domain", you update the domain's PHP mode option use the latest available PHP version. Indeed this will cause the web server to use that version for running your application, however the PHP CLI (command-line interface) will confusingly continue to refer to the default PHP version (5.3.6 at the time of this writing). You'll want to update the CLI to instead use the same version you specified via the PHP mode setting. This is incredibly easy to do, requiring just a few steps. Begin by SSH'ing into your DreamHost account and creating a directory named bin inside your account home directory:

```
1  $ mkdir -p ~/bin
```

Next you'll create a symbolic link inside this directory which points to the same binary version you referenced in the `PHP mode` setting. I used PHP 5.6 above and so will refer to that version in the below example:

```
1  $ ln -s /usr/local/bin/php-5.6 ~/bin/php
```

Next we want to be able to just execute `php` anywhere within the account directory tree and result in the linked PHP binary being executed. To do so, add this newly created directory to your system path and make sure that updated path is available for both login and non-login shells:

```
1  $ echo "export PATH=~/bin/:\$PATH" >> ~/.bash_profile
2  $ echo "export PATH=~/bin/:\$PATH" >> ~/.bashrc
```

Next you'll want to enable PHP's phar (PHP Archive) extension. Composer is distributed in Phar format, which if you've ever used Java is akin to Java's JAR format. It is just a way to easily distribute PHP-powered applications such as Composer. Don't worry too much about the details here; you just need to carry out a few simple steps to make Phar available to your PHP CLI version. Begin by creating a directory for housing a custom configuration file which will be used by your PHP CLI. Again I'm using a version-specific directory name here, which you might want to change if you're using a PHP version newer than 5.6:

```
1  $ mkdir -p ~/.php/5.6
```

Next, we'll use a command-line convenience to create a file named `phprc` inside this newly created directory, and add the line `extension = phar.so` to it. This tells the PHP CLI to load the `phar.so` extension:

```
1  $ echo "extension = phar.so" >> ~/.php/5.6/phprc
```

Finally, you can confirm Phar is enabled by running the following command:

```
1  $ php -m | grep Phar
2  Phar
```

With Phar installed, it's time to install Composer!

Installing Composer on DreamHost

Just as you use Composer locally to install and manage your project dependencies, so will you need it available on our DreamHost for the same purposes. Fortunately, once Phar has been enabled it is very easy to install and run Composer on your DreamHost server. Begin by downloading the Composer installer:

```
1  $ curl -s https://getcomposer.org/installer | php -- --install-dir=~/bin
```

Once complete, all you need to do is change the permissions of the `composer.phar` file which has been placed in the `bin` directory you created when updating the PHP CLI version:

```
1  $ chmod u+x ~/bin/composer.phar
```

For convenience reasons I prefer to remove the unnecessary `phar` extension from the file:

```
1  $ mv ~/bin/composer.phar ~/bin/composer
```

Once complete you should be able to run Composer in the very same way you do on your development machine:

```
1  $ composer --version
2  Composer version 1.0-dev
```

Configuring Passwordless Login

Next up we'll configure passwordless login. This means you'll be able to SSH into your DreamHost account without supplying a password, like this:

```
1  $ ssh wjgilmore@www.wjgilmore.com
2
3  No mail.
4  Last login: Mon Aug 10 12:56:48 2015 from 1.2.3.4
5  [~]:
```

Notice how I wasn't prompted for a password? This can be incredibly convenient, particularly when you are regularly SSHing into several different servers on a regular basis, each with a distinct account password. Additionally, it's useful for Capistrano deployments because it saves you the hassle of having to enter your DreamHost account username and password every time you deploy. You can do so by providing DreamHost with your public SSH key, and then use key-based authentication when connecting to the server.

If you're not familiar with key-based authentication but seem to recall having generated a pair of public and private SSH key pair at some point, you can easily check by opening a terminal and navigating to your local home directory. List your directory contents, looking for a directory named `.ssh`. If it exists, look inside this directory for files that look like this:

```
1   id_rsa.pub
2   id_rsa
```

If these exist, then indeed you have already created a key pair. If this .ssh directory does not exist, or if the directory is empty, you'll need to generate the keys. If you're on Mac OSX or Linux, this is incredibly easy. Just execute the ssh-keygen command as demonstrated below (replacing the e-mail address placeholder with your actual address), and then follow along with the subsequent prompts:

```
1    $ ssh-keygen -t rsa -b 4096 -C "YOUR_EMAIL_ADDRESS_HERE"
2    Generating public/private rsa key pair.
3    Enter file in which to save the key
4       (/Users/wjgilmore/.ssh/id_rsa):
5    Enter passphrase (empty for no passphrase):
6    Enter same passphrase again:
7    Your identification has been saved in
8       /Users/wjgilmore/.ssh/id_rsa.
9    Your public key has been saved in
10      /Users/wjgilmore/.ssh/id_rsa.pub.
```

Note you'll be prompted to supply what's referred to as a *passphrase*. This is just a fancy word for password. This passphrase can be as long as you desire, and if you plan on using passwordless login to any degree then I suggest choosing a sufficiently long and complicated passphrase. This is because even if a third-party were to obtain your private key (id_rsa in the above example), they will not be able to masquerade as you without knowing that passphrase! This is because every time the keys are used, you'll be required to enter the passphrase.

At first glance it seems like this really isn't any more convenient then manually signing in via SSH in the first place, however you can avoid that by making your keys available to what's referred to as the *SSH agent*. This is a program that will manage and unlock your keys for you, requiring you to only enter your passphrase once per login. These days, operating systems such as OSX even go the extra step of optionally adding the passphrase to your system *keychain*, meaning you won't ever be bothered with having to enter the passphrase again (although you still must keep the passphrase in a safe place should you ever wish to use the SSH keys on another machine). To add your keys to the agent, execute the ssh-add command:

```
1    $ ssh-add ~/.ssh/id_rsa
2    Enter passphrase for ~/.ssh/id_rsa:
```

Once OSX / Linux readers have completed the above step, skip ahead to the section, "Configuring DreamHost for Passwordless Login".

If you're on Windows, per usual things aren't so straightforward but is nonetheless easily achievable. I am however going to take the rare easy way out of further explanation and point you to this

excellent set of instructions on the Joyent website[180] because I haven't used Windows in years and would therefore prefer readers have access to what seems to be a regularly updated set of instructions. Once complete, return to the book and go to the section, "Configuring DreamHost for Passwordless Login".

Configuring DreamHost for Passwordless Login

Once you've configured your SSH key pair, you'll need to provide DreamHost with your public key. Your public key is found in the file having the extension `.pub` in your `.ssh` directory. If you open this file you'll see that it contains an absurdly long random-looking string like this:

```
1  ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQCg4QD5XAvxFy9KgAf815gnpLYW
2  Z13c1eKDM2goER9wC7KgnQuXv7WRPl2gnPzHHMaFSMKFfL0Qyes6v491hBVmXYvQM
3  Yr1cAAEdTdJ2A1pMkrnTGqr5SeNZ0XdI6CFLm3bSBpXF8zedC/ng5Q0fiitcGJ/2oS1
4  Ee8NWvqInWpmVjs0mAfk6L7YlH5sAjAqaLRrQw9E0lXgCPkOCxrC7288Ovmvqn4MDGxSWS
5  8KGf7s6xz/7ZZqd4/yzbdTGScTKLWm2sOMPIkJ1jlMdkTSLniBagAwbKxmhi3CxB4yRg
6  hZCJFqIoLRfs0vIrZdsZ82YcfUmQyX2CJ/S+bcIjS90XIaO1q4nXsrzWCYD9uh7PYk560Wq8
7  TXrFVZ0PuIl4GykB8wSSG6dE74bB9Dr/zxrmh05YadrS96v9q+y/aoEW74DWWXuRgCEi
8  CqVaSrKv5i29hAqIvFtA+/99Jojl9jp9k0X4T/cWPWS+YBQfz6JXImCxLHH6k4wPWKNXlY
9  xktR6FXhrdSoA8n9YppkVx332rq7BkK5/XKS4AeJqPkJIbXGS8i8c/VyVVqKNRMASZf1Vu1r
10 ZwXi1eDb9bLvwNcw/Nv4fFTDghux85j63rKyGUUfIp10ZUxrL+RbviZWxY4x+Ft6jlyGn/9q
11 u6oBy3eKPYcFhnkUBuhxY5O6R/3DoIaw==your_email@example.com
```

Copy these key onto your clipboard, and then SSH into your DreamHost server:

```
1  $ ssh wjgilmore@example.com
```

Next, create a directory named `.ssh` inside your DreamHost account's home directory:

```
1  $ mkdir .ssh
```

Next, enter the `.ssh` directory, and create a file named `authorized_keys`. You can use terminal editors such as vim or nano to create this file. Alternatively you can create the file locally, and then use SFTP to upload this file to the `.ssh` directory. In any case, you'll then want to paste your *public key* (not your private key) into the `authorized_keys` directory. Once complete, exit the DreamHost server, and then try SSHing in anew:

```
1  $ ssh wjgilmore@example.com
```

This time you should immediately enter the server without having to type your password!

---

[180]http://bit.ly/1LhUhPj

Adding Your Public Key to GitHub

In order for the DreamHost server to conveniently talk to GitHub, you'll need to add this newly generated public key to your GitHub account. To do so, sign into your GitHub account, and navigate to `Settings > SSH Keys` and press the `Add SSH Key` button. Paste the `id_rsa.pub` contents into the text area, assign the key some easily recognizable name (such as `my laptop`) and save the changes.
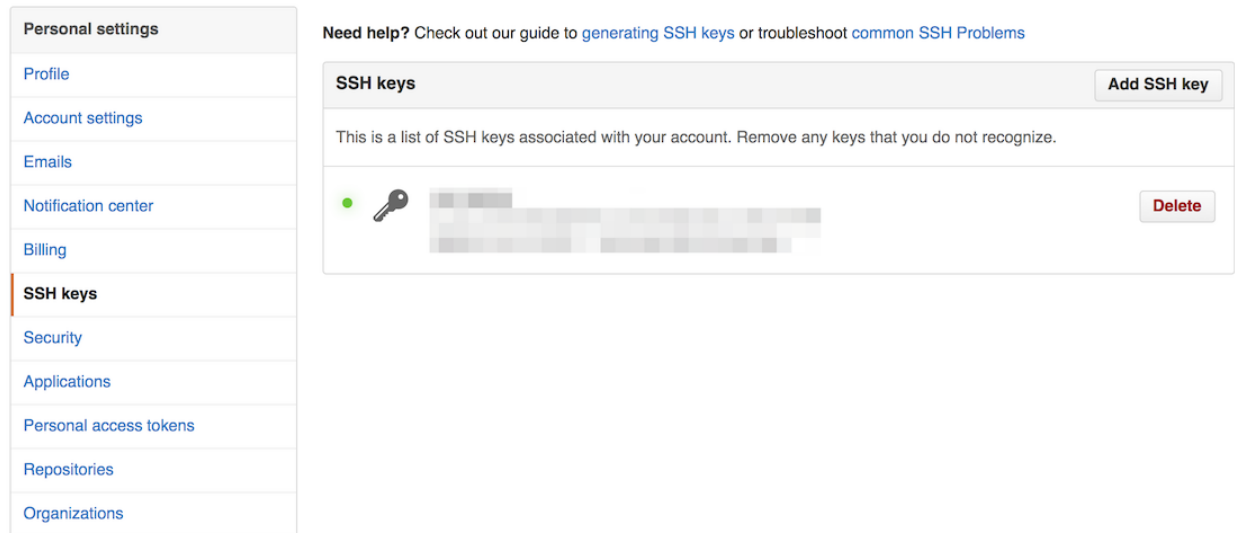
Associating a Public Key with Your GitHub Account

Installing and Configuring Capistrano

With your domain configured and the registrar's DNS updated, it's time to complete the few remaining deployment-specific tasks. Unless you plan on using a different user in conjunction with each domain hosted on DreamHost, the instructions found here are the only ones you'll have to repeat on a per-domain basis when deploying with Capistrano (other than installing Capistrano on your local machine, which you'll only have to do once)!

Let's begin by installing Capistrano. Capistrano is distributed as a Ruby gem, meaning you'll need to install Ruby on your local machine. Keep in mind you will *not* have to learn the Ruby language or anything like that to begin using Capistrano! You just need to install the Ruby interpreter, since Capistrano happens to have been written in Ruby. If you're running OSX or Linux, then chances are Ruby is already installed. You can confirm this by opening a terminal prompt and executing the following:

```
1  $ ruby -v
```

Presuming a version number is installed, then you're ready to go unless you're running a version older than 1.9.3 which is the minimum Capistrano requirement, in which case you'll need to use

your package manager to upgrade. If you're running Windows then you can use the officially recommended Ruby Installer[181].

Once Ruby has been installed, install Capistrano by executing the following command from a terminal:

```
1  $ gem install capistrano
```

Once Capistrano has been installed, enter the root directory of the project you'd like to deploy to DreamHost, and execute the following command:

```
1  $ cap install
```

This command will create several directories and files inside your project root directory, including:

- `Capfile`: This file is used by Capistrano to load the core library files and other optional extensions. You won't need to do anything with this file to deploy a Laravel project as described in this chapter, other than leaving the file in place.
- `config`: This directory will be created for reason of housing a directory named `deploy` and a file named `deploy.rb`. The `deploy` directory will in turn house two files named `production.rb` and `staging.rb`. These files contain *target-specific* deployment instructions. For instance, you could define different sets of deployment instructions for different servers, which would be useful if you wanted to manage a beta and production server. For the purposes of this chapter I'll be discussing solely the `production.rb` file. The `deploy.rb` file contains a general set of deployment settings which will be inherited by `production.rb` and `staging.rb` unless one of these latter files expressly overrides those defaults. We'll return to `deploy.rb` and `production.rb` in just a moment, so don't worry about them too much more now. Of course, Laravel applications already have a `config` directory, so Capistrano won't try to recreate it or anything like that. Instead it will just create the aforementioned `deploy` directory (along with the two files) and `deploy.rb` file inside it
- `lib`: This directory houses any custom Capistrano behavior you create for reason of managing your deployments. You won't need to do anything with this directory, so just leave it in place and don't worry about it for now.

With the necessary directories and files in place, we'll need to modify the `deploy.rb` and `production.rb` files. Let's begin with the `deploy.rb` file. Open it up, marvel at the many options available to you, and then delete everything in the file, replacing it with the following. Please carefully read the comments found throughout this file (prefixed with #) as it is crucial for you to understand the various settings found herein:

---

[181]http://rubyinstaller.org/

```
 1   # This should match whatever Capistrano gem version
 2   # Execute cap -V to find this out.
 3   lock '3.2.1'
 4
 5   # Where does your project GitHub repository reside?
 6   set :repo_url, 'git@github.com:wjgilmore/example.git'
 7
 8   # This is where the files will reside. This should
 9   # match the directory you defined when configuring
10   # your domain directory, but WITHOUT the current/public
11   # path
12   set :deploy_to, '/home/YOUR_USERNAME/example.com'
13
14   # Use this directory for housing the cloned repository
15   # NOTE: you will need to create this directory inside
16   # your DreamHost account home directory.
17   set :tmp_dir, '/home/YOUR_USERNAME/tmp'
18
19   # We're deploying a Git repository
20   set :scm, :git
21
22   # Set the logging to debug so we can see what is going on
23   set :log_level, :debug
24
25   # Maintain three releases on the server
26   set :keep_releases, 3
27
28   namespace :deploy do
29
30     # After deployment is complete, execute the
31     # following commands. Notably, we are going to
32     # run composer, make the artisan script executable,
33     # create a symbolic link to a production-specific
34     # .env file, and then run any outstanding migrations.
35     desc "Build"
36     after :updated, :build do
37       on roles(:app) do
38         within release_path  do
39           execute :composer, "install --no-dev --quiet"
40           execute :chmod, "u+x artisan"
41           execute "ln -nfs #{shared_path}/.env #{release_path}/.env"
42           execute :php, "artisan migrate --force" # run migrations
```

```
43              end
44          end
45      end
46
47  end
```

The comments found in the above code should be suffice to help you understand what's happening, however take special note of the post-deployment step in which we're creating a symbolic link to a production-specific .env file. You're going to want to host an .env file that is almost certainly from that found locally, since logically you're going to want your application to connection to a production-specific database, perhaps use different mail server connection parameters, and so forth. Therefore you'll want to upload a production-specific .env file to DreamHost. It's common convention to place these "shared" files inside a directory named shared, and so go ahead and login to your DreamHost server now and create that directory:

```
1  $ ssh wjgilmore@example.com
```

After logging in, enter the domain directory that was automatically created when you configured the domain:

```
1  $ cd example.com
```

This directory will contain a directory named current, and inside it a directory named public. This is because you identified the document root as such when configuring the domain. However, we're going to let Capistrano manage this path (exactly why we want to do this will become clear in a moment), so delete this current directory (and everything inside it):

```
1  $ rm -rf current
```

Next, remaining inside the domain directory, create a directory named shared:

```
1  $ mkdir shared
```

Copy the contents of your project's .env file, enter this newly created shared directory, and paste the contents into a file also named .env. Update the configuration parameters to reflect those used for your production server (such as the DreamHost MySQL hostname, username, database name, and password). Once this file is in place, Capistrano will create a symbolic link from this file to the newly deployed project root directory by executing the following line within the post-deployment block found in deploy.rb:

```
1   execute "ln -nfs #{shared_path}/.env #{release_path}/.env"
```

Incidentally, there are somewhat more automated ways to handle shared files in Capistrano, however this first time around I thought it would be useful for you to manually create the shared directory and create the .env file, because otherwise the deployment process tends to work like magic, and magic can be dangerous when you're trying to figure out what went wrong at some point down the road.

Next, open config/deploy/production.rb and replace the contents with the following:

```
1   role :app, %w{wjgilmore@example.com}
2
3   set :ssh_options, {
4       forward_agent: true,
5       auth_methods: %w(publickey),
6       user: 'wjgilmore'
7   }
```

All we are doing here is telling Capistrano that when deploying to the production server, we're going to SSH in using the username wjgilmore and the domain address example.com, and as defined by the SSH options we'll be authenticating using the user wjgilmore's public key. The forward_agent option is a convenient solution for allowing your DreamHost account user to furnish your *local* SSH keys to GitHub when it's time to access the hosted repository. In order for this to work, keep in mind you will need to add your public key to your GitHub account as described in the earlier section, "Configuring DreamHost for Passwordless Login".

Deploying Your Application

Whew, that was a pretty long process but we are done. All that remains for you to do is deploy! After having committed your changes and pushed them to GitHub (or Bitbucket or anywhere else you please), you can deploy your project to DreamHost by executing the following command:

```
1   $ capistrano production deploy
```

This tells Capistrano to use the production.rb file settings to carry the deployment process. Combined with the information found in deploy.rb, Capistrano will have everything it needs to complete the process! When executing this command, you'll be treated to a rather exhaustive bit of output pertaining to everything Capistrano is doing to complete the deployment. This is because we set log_level to debug in deploy.rb. You can tone down the verbosity if desired; see the Capistrano documentation for more information.

Once deployed, return to your DreamHost server and enter the domain directory. You'll now see the following directory contents:

- `current`: This is a *symbolic link* pointing to the *most recently deployed* version of your code. Read that twice. For instance this link will point to a directory such as /home/wjgilmore/example.com/releases/20150810233629. Notice how the link points to a directory found in releases'. Capistrano gives you the ability to actually rollback your deployment to a previous version, doing so by simply removing the symbolic link to the latest deployment and then linking to the second-newest deployment (as determined by the timestamp)! I'll show you how to rollback in just a moment.
- `releases`: This directory contains a predetermined number of current and previous deployments, the number of which being determined by the `keep_releases` setting in your `deploy.rb` file. For instance, if `keep_releases` is set to 3, then a total of three deployments will be maintained (the current deployment and the previous two).
- `repo`: This contains the cloned repository.
- `shared`: This is the directory we created prior to deploying; it contains the production `.env` file.

Each time you deploy a new version of your project, Capistrano will retrieve the latest changes, archive them to a timestamped directory found in `releases`, remove the symbolic link pointing to the now "old" deployment, and then create a new symbolic link pointing to the new deployment. It's an incredibly easy yet powerful solution! Among other things this gives you the ability to roll back your changes using the `deploy:rollback` command:

```
1  $ cap production deploy:rollback
```

After deploying for the first time, make a change to your project (add some text to the home page or similar), deploy anew, confirm the changes are on the production site, and then run `deploy:rollback`. Once the command completes, your production site will have reverted to the previous version! Then, undo and commit the local changes, deploy once more, and the production site will no longer include the changed text!

Summary

As you can see, if you require a custom deployment solution using a tool such as Capistrano, it can be a bit of a chore to configure the first time around. Once in place though, it is a solution you'll return to repeatedly for all of your new projects. I've been using Capistrano for several years now and honestly don't know what I'd do without it. And to think we've hardly scratched the surface in terms of what it can do! Be sure to check out the Capistrano documentation for more information.