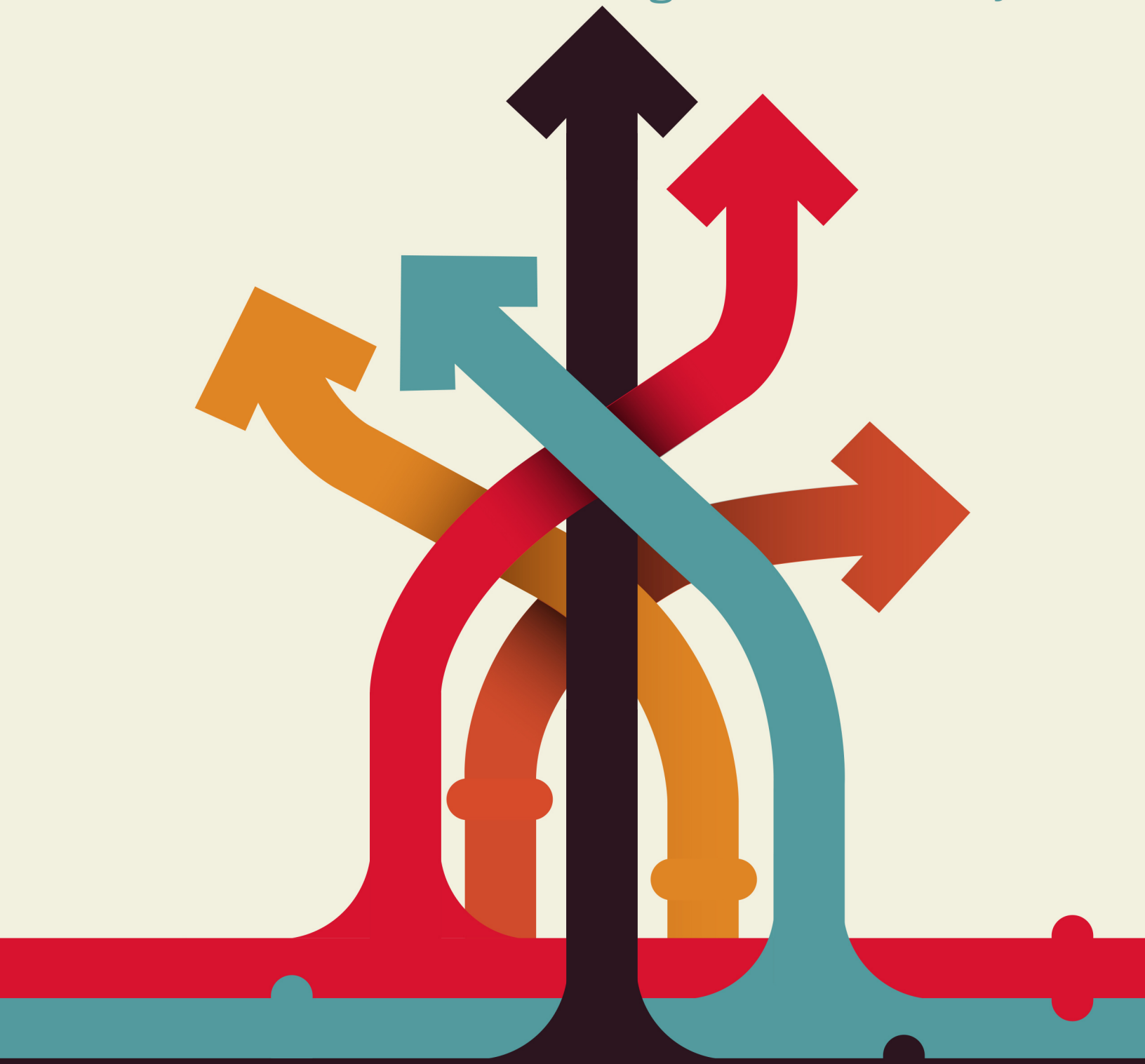
EASY LARAVEL 5

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 2015-02-05

Also By W. Jason Gilmore

Easy Active Record for Rails Developers

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

Contents

Introduction

I've spent the vast majority of the past 15 years immersed in the PHP language. During this time I've written seven PHP-related books, including a bestseller that has been in print for more than ten years. Along the way I've worked on dozens of PHP-driven applications for clients ranging from unknown startups to globally-recognized companies, penned hundreds of articles about PHP and web development for some of the world's most popular print and online publications, and instructed hundreds of developers in the United States and Europe. So you might be surprised to learn that a few years ago I became rather disenchanted with PHP. It felt like there were more exciting developments taking place within other programming communities, and wanting to be part of that buzz, I wandered off. In recent years, I spent the majority of my time working on a variety of projects including among others several ambitious Ruby on Rails applications and even a pretty amazing Linux-powered robotic device.

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

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

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

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

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

What's New in Laravel 5?

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

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

About this Book

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

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

Chapter 1. Introducing Laravel

In this opening chapter you'll learn how to create and configure your Laravel project both using your existing PHP development environment and Laravel Homestead. I'll also show you how to properly configure your environment for effective Laravel debugging, and how to expand Laravel's capabilities by installing several third-party Laravel packages that promise to supercharge your development productivity. We'll conclude the chapter with an introduction to PHPUnit, showing you how to create and execute your first Laravel unit test!

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

In this chapter you'll learn how to create controllers and actions, and define the routes used to access your application endpoints using Laravel 5's new route annotations feature. You'll also learn how to create the pages (views), work with variable data and logic using the Blade templating engine, and reduce redundancy using layouts and view helpers. I'll also introduce Laravel Elixir, a new feature for managing Gulp[10] tasks, and show you how to integrate the popular Bootstrap front-end framework and jQuery JavaScript library. We'll conclude the chapter with several examples demonstrating how to test your controllers and views using PHPUnit.

Chapter 3. Talking to the Database

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

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

Building and navigating table relations is an standard part of the development process even when working on the most unambitious of projects, yet this task is often painful when working with many web frameworks. Fortunately, using Laravel it's easy to define and traverse these relations. In this chapter I'll show you how to define, manage, and interact with one-to-one, one-to-many, many-to-many, has many through, and polymorphic relations. You'll also learn about a great feature known as scopes which encapsulate the logic used for more advanced queries, thereby hiding it from your controllers.

[10] http://gulpjs.com/

Chapter 5. Forms Integration

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

Chapter 6. Integrating Middleware

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

Chapter 7. Authenticating and Managing Your Users

Most modern applications offer user registration and preference management features in order to provide customized, persisted content and settings. In this chapter you'll learn how to integrate user registration, login, and account management capabilities into your Laravel application.

Chapter 8. Deploying, Optimizing and Maintaining Your Application

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

Introducing the TODOParrot Project

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

[11] http://todoparrot.com

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

About the Author

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

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

Over the years Jason has published more than 300 articles within popular publications such as Developer.com, JSMag, and Linux Magazine, and instructed hundreds of students in the United States and Europe. Jason is cofounder of the wildly popular CodeMash Conference[14], the largest multi-day developer event in the Midwest.

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

Jason loves talking to readers and invites you to e-mail him at wj@wjgilmore.com.

Errata and Suggestions

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

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

Chapter 1. Introducing Laravel

Laravel is a web application framework that borrows from the very best features of other popular framework solutions, among them Ruby on Rails and ASP.NET MVC. For this reason, if you have any experience working with other frameworks then I'd imagine you'll make a pretty graceful transition to Laravel-driven development. If this is your first acquaintance with framework-driven development, you're in for quite a treat! Frameworks are so popular precisely because they dramatically decrease the amount of work you'd otherwise have to do by making many of the mundane decisions for you, a concept known as convention over configuration[15].

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

Everything you learn in this book is based upon Laravel 5, which at the time of this writing is under heavy development and not slated for release until January, 2015. I will do my very best to stay on top of the changes and update the book as quickly as possible, however it is entirely likely you'll occasionally encounter a bit of code that is no longer correct. Such is life when living on the cutting edge of technology. So please be patient, have fun, and if you think something is wrong, e-mail me at wj@wjgilmore.com.

Installing Laravel

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

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

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

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

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

Introducing Homestead

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

In recent years the bar was dramatically lowered with the advent of the *virtual machine*. A virtual machine is a software-based implementation of a computer that can be run inside the confines of another computer (such as your laptop), or even inside another virtual machine. This is an incredibly useful bit of technology, because you can use a virtual machine to for instance run Ubuntu Linux inside Windows 7, or vice versa. Further, it's possible to create a customized virtual machine image preloaded with a select set of software. This image can then be distributed to fellow developers, who can run the virtual machine and take advantage of the custom software configuration. This is precisely what the Laravel developers have done with Homestead[22], a Vagrant[23]-based virtual machine which bundles everything you need to get started building Laravel-driven websites.

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

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

Installing Homestead

Homestead requires Vagrant[24] and VirtualBox[25]. User-friendly installers are available for all of the common operating systems, including OS X, Linux and Windows. Take a moment now to install Vagrant and VirtualBox. Once complete, open a terminal window and execute the following command:

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

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

This command installs the Homestead *box*. A box is just a term used to refer to a Vagrant package. Packages are the virtual machine images that contain the operating system and various programs. The Vagrant community maintains a variety of boxes useful for different applications, so check out this list of popular boxes[26] for an idea of what else is available.

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

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

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

Writing lock file
Generating autoload files
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Save these changes, and then run the following command:

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

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

Next you'll want to update your development machine's `hosts` file so you can easily access the server via a hostname rather than the IP address found in the `Homestead.yaml` file. If you're running OSX or Linux, this file is found at `/etc/hosts`. If you're running Windows, you'll find the file at `C:\Windows\System32\drivers\etc\hosts`. Open up this file and add the following ine:

```
192.168.10.10  homestead.app
```

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

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

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

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

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

```
$ homestead destroy
```

SSH'ing Into Your Virtual Machine

Because Homestead is a virtual machine running Ubuntu, you can SSH into it just as you would any other server. For instance you might wish to configure nginx or MySQL, install additional software, or make other adjustments to the virtual machine environment. You can SSH into the virtual machine using the `ssh` command if you're running Linux or OS X, or using a variety of SSH clients if you're running Windows (My favorite Windows SSH client is PuTTY[29].:

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

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

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

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

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

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

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

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

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

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

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

Creating the TODOParrot Application

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

```
$ composer create-project laravel/laravel dev.todoparrot.com --prefer-dist
Installing laravel/laravel (v5.0.0)
```

- Installing laravel/laravel (v5.0.0) ... Compiling views Application key [9UCBk7IDjvAGrkLOUBXw43yYKlyml
  set successfully.

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

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

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

This command tells Composer to create a new project using the `laravel/laravel` package, placing the project contents in the directory `dev.todoparrot.com`. These contents are a combination of files and directories, each of which plays an important role in the functionality of your application so it's important for you to understand their purpose. Let's quickly review the role of each:

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

- `.env`: Laravel 5 uses the PHP dotenv[32] to conveniently manage environment-specific settings. You'll use `.env` file as the basis for configuring these settings. A file named `.env.example` is also included in the project root directory. This file should be used as the setting template, which fellow developers will subsequently copy over to `.env` and change to suit their own needs. I'll talk about this file and Laravel 5's new approach to managing environment settings in the later section, "Configuring Your Laravel Application".
- `.gitattributes`: This file is used by Git[33] to ensure consistent settings across machines, which is particularly useful when multiple developers using a variety of operating systems are working on the same project. The lone setting found in your project's `.gitattributes` file (`text=auto`) ensures file line endings are normalized to `LF` whenever the files are checked into the repository. Plenty of other attributes are however available; Scott Chacon's book, "Pro Git"[34] includes a section ("Customizing Git - Git Attributes"[35]) with further coverage on this topic.
- `.gitignore`: This file tells Git what files and folders should not be included in the repository. You'll see the usual suspects in here, including the annoying OS X `.DS_Store` file, Windows' equally annoying `Thumbs.db` file, and the `vendor` directory, which includes the Laravel source code and various other third-party packages.
- `app`: This directory contains much of the custom code used to power your application, including the models, controllers, and middleware. We'll spend quite a bit of time inside this directory as the application development progresses.
- `artisan`: `artisan` is a command-line interface we'll use to rapidly develop new parts of your applications such as controllers, manage your database's evolution through a great feature known as *migrations*, and clear the application cache. You'll also regularly use `artisan` to interactively debug your application, and even easily view your application within the browser using the native PHP development server. We'll return to `artisan` repeatedly throughout the book as it is such an integral part of Laravel development.
- `bootstrap`: This directory contains the various files used to initialize a Laravel application, loading the configuration files, various application models and other classes, and define the locations of key directories such as `app` and `public`. Normally you won't have to modify any of the files found in the `bootstrap` directory, although I encourage you to have a look as each is heavily commented.
- `composer.json`: Composer[36] is the name of PHP's popular package manager, used by thousands of developers around the globe to quickly integrate popular third-party solutions such as Swift Mailer[37] and Doctrine[38] into a PHP application. Laravel supports Composer, and you'll use the `composer.json` file to identify the packages you'll like to integrate into your Laravel application. If you're not familiar with Composer you'll quickly come to wonder how you

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

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

- `composer.lock`: This file contains information about the state of the installed Composer packages at the time these packages were last installed and/or updated. Like the `bootstrap` directory, you will rarely if ever directly interact with this file.
- `config`: This directory contains more than a dozen files used to configure various aspects of your Laravel application, such as the database credentials, and the cache, e-mail delivery and session settings.
- `database`: This directory contains the directories used to house your project's database migrations and seed data (migrations and database seeding are both introduced in Chapter 3).
- `gulpfile.js`: Laravel 5 introduces a new feature called *Laravel Elixir*. `Gulpfile.js` is used by Elixir to define various Gulp.js[39] tasks used by Elixir to automate various build-related processes associated with your project's CSS, JavaScript, tests, and other assets. I'll introduce Elixir in Chapter 2.
- `package.json`: This file is used by the aforementioned Elixir to install Elixir and its various dependencies. I'll talk about this file in Chapter 2.
- `phpspec.yml`: This file is used to configure the behavior driven development tool phpspec[40]. In this book I'll discuss Laravel testing solely in the context of PHPUnit but hope to include coverage of phpspec in a forthcoming update.
- `phpunit.xml`: Even relatively trivial web applications should be accompanied by an automated test suite. Laravel leaves little room for excuse to shirk this best practice by configuring your application to use the popular PHPUnit[41] test framework. The `phpunit.xml` is PHPUnit's application configuration file, defining characteristics such as the location of the application tests. We'll return to this topic repeatedly throughout the book, so stay tuned.
- `public`: The `public` directory serves as your application's root directory, housing the `.htaccess`, `robots.txt`, and `favicon.ico` files, in addition to a file named `index.php` that is the *first* file to execute when a user accesses your application. This file is known as the *front controller*, and it is responsible for loading and executing the application.
- `readme.md`: The `readme.md` file contains some boilerplate information about Laravel of the sort that you'll typically find in an open source project. Feel free to replace this text with information about your specific project. See the TODOParrot[42] README file for an example.
- `resources`: The `resources` directory contains your project's views and localized language files. You'll also store your project's raw assets (CoffeeScript, SCSS, etc.).
- `storage`: The `storage` directory contains your project's cache, session, and log data.
- `tests`: The `tests` directory contains your project's PHPUnit tests. Testing is a recurring theme throughout this book, complete with numerous examples.

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

- `vendor`: The `vendor` directory is where the Laravel framework code itself is stored, in addition to any other third-party code. You won't typically directly interact with anything found in this directory, instead doing so through the `artisan` utility and Composer interface.

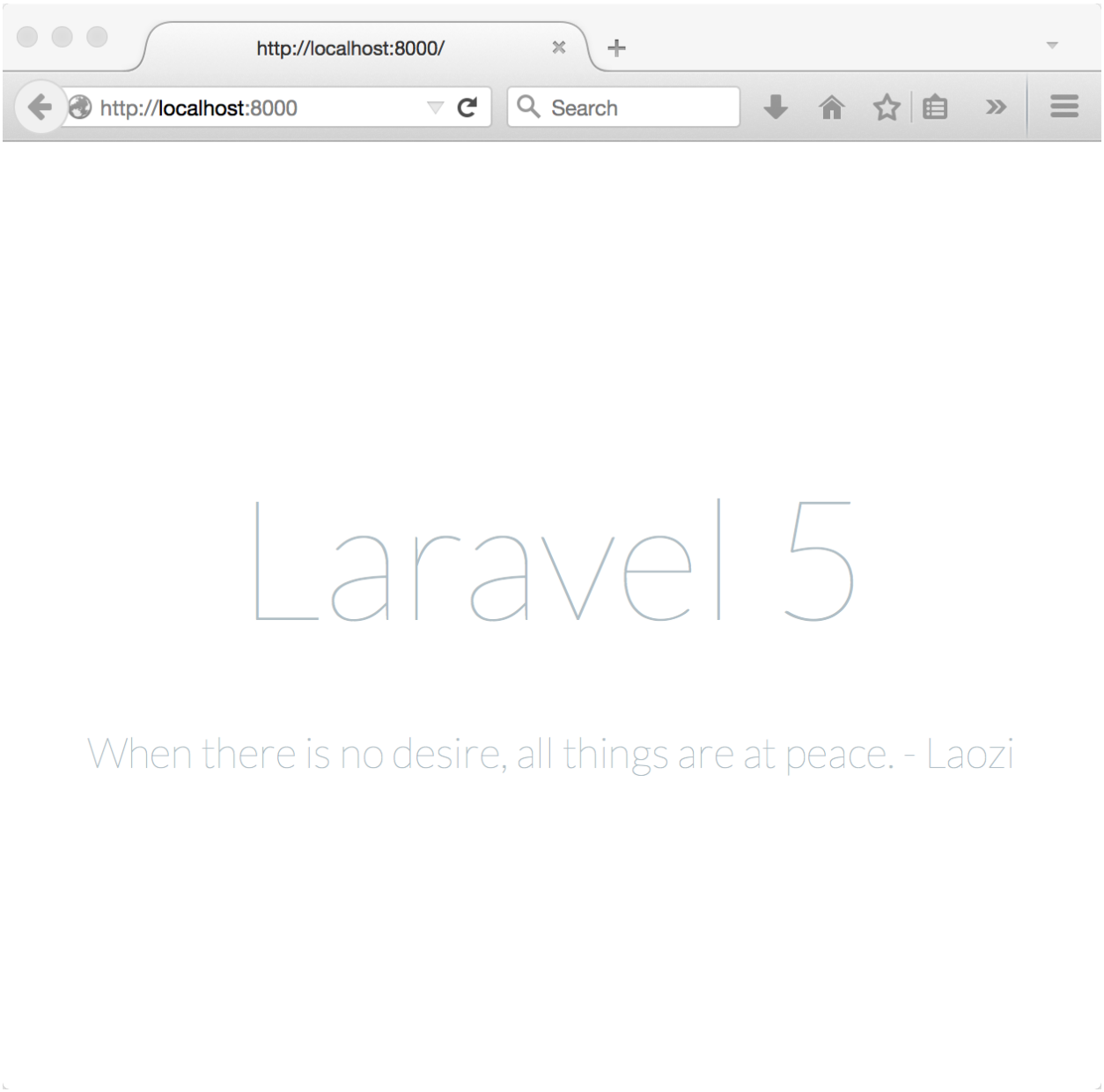
Now that you have a rudimentary understanding of the various directories and files comprising a Laravel skeleton application let's see what happens when we load the default application into a browser. If you're using Homestead then navigate to `http://homestead.app`, otherwise if you plan on using PHP's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

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

Alternatively you could use the built-in PHP server:

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

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



The Laravel splash page

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

Setting the Application Namespace

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

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

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

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

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

Configuring Your Laravel Application

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

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

- `app.php`: The `app.php` file contains settings that have application-wide impact, including whether debug mode is enabled (more on this in a moment), the application URL, timezone, locale, and autoloaded service providers.
- `auth.php`: The `auth.php` file contains settings specific to user authentication, including what model manages your application users, the database table containing the user information, and how password reminders are managed. I'll talk about Laravel's user authentication features in Chapter 7.
- `cache.php`: Laravel supports several caching drivers, including filesystem, database, memcached, redis, and others. You'll use the `cache.php` configuration file to manage various settings specific to these drivers.
- `compile.php`: Laravel can improve application performance by generating a series of files that allow for faster package autoloading. The `compile.php` configuration file allows you to define additional class files that should be included in the optimization step.
- `database.php`: The `database.php` configuration file defines a variety of database settings, including which of the supported databases the project will use, and the database authorization credentials.
- `filesystems.php`: The `filesystems.php` configuration file defines the file system your project will use to manage assets such as file uploads. Currently the local disk, Amazon S3, and Rackspace are supported.

- `mail.php`: As you'll learn in Chapter 5 it's pretty easy to send an e-mail from your Laravel application. The `mail.php` configuration file defines various settings used to send those e-mails, including the desired driver (SMTP, Sendmail, PHP's `mail()` function, Mailgun, and the Mandrill API are supported). You can also direct mails to the log file, a technique that is useful for development purposes.
- `queue.php`: Queues can improve application performance by allowing Laravel to offload time- and resource-intensive tasks to a queueing solution such as Beanstalk[44] or Amazon Simple Queue Service[45]. The `queue.php` configuration file defines the desired queue driver and other relevant settings.
- `services.php`: If your application uses a third-party service such as Stripe for payment processing or Mandrill for e-mail delivery you'll use the `services.php` configuration file to define any third-party service-specific settings.
- `session.php`: It's entirely likely your application will use sessions to aid in the management of user preferences and other customized content. Laravel supports a number of different session drivers used to facilitate the management of session data, including the file system, cookies, a database, the Alternative PHP Cache[46], Memcached, and Redis. You'll use the `session.php` configuration file to identify the desired driver, and manage other aspects of Laravel's session management capabilities.
- `view.php`: The `view.php` configuration file defines the default location of your project's view files and the renderer used for pagination.

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

Programming Terminology Alert

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

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

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

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

Configuring Your Environment

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

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

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

CACHE_DRIVER=file
SESSION_DRIVER=file
```

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

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

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

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

Useful Development and Debugging Tools

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

The debugging and development utilities discussed in this section are specific to Laravel, and do not take into account the many other tools available to PHP in general. Be sure to check out Xdebug[48], FirePHP[49], and the many tools integrated into PHP IDEs such as Zend Studio[50] and PHPStorm[51].

The dd Function

Ensuring the `debug` option is enabled is the easiest way to proactively view information about any application errors however it isn't a panacea for all debugging tasks. For instance, sometimes you'll want to peer into the contents of an object or array even if the data structure isn't causing any particular problem or error. You can do this using Laravel's `dd()`[52] helper function, which will dump a variable's contents to the browser and halt further script execution. To demonstrate the `dd()` function open the file `app/Http/Controllers/HomeController.php` and you'll find single class method that looks like this:

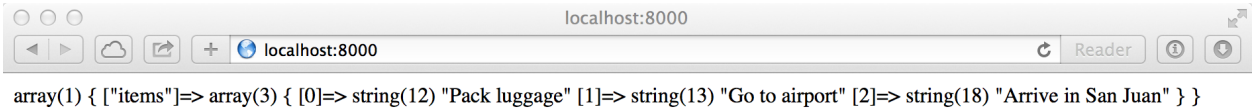
```
1  public function index()
2  {
3      return view('hello');
4  }
```

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

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

```
public function index()
{

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

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

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



dd() function output

The Laravel Logger

While the `dd()` helper function is useful for quick evaluation of a variable's contents, taking advantage of Laravel's logging facilities is a more effective approach if you plan on repeatedly monitoring one or several data structures or events without necessarily interrupting script execution. Laravel will by default log error-related messages to the application log, located at `storage/logs/laravel.log`. Because Laravel's logging features are managed by Monolog[53], you have a wide array of additional logging options at your disposal, including the ability to write log messages to this log file, set logging levels, send log output to the Firebug console[54] via FirePHP[55], to the Chrome console[56] using Chrome Logger[57], or even trigger alerts via e-mail, HipChat[58] or Slack[59]. Further, if you're using the Laravel 4 Debugbar (introduced later in this chapter) you can easily peruse these messages from the Debugbar's `Messages` tab.

Generating a custom log message is easy, done by embedding one of several available logging methods into the application, passing along the string or variable you'd like to log. Open the `app/Http/Controllers/HomeController.php` file and modify the `index` method to look like this:

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

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

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

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

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

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

Integrating the Logger and FirePHP

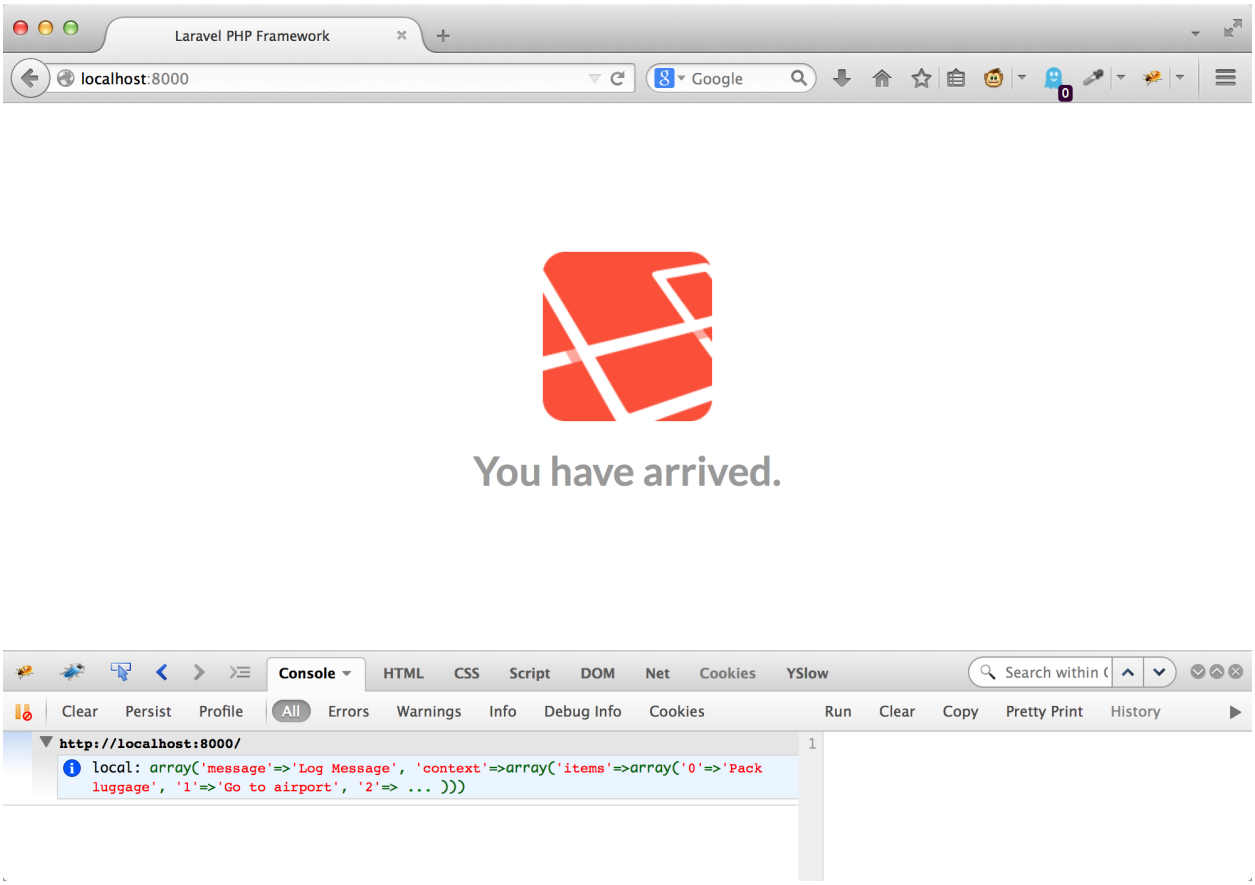
When monitoring the log file it's common practice to use the `tail -f` command (available on Linux and OS X) to view any log file changes in real time. You can however avoid the additional step of maintaining an additional terminal window for such purposes by instead sending the log messages to the Firebug[60]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

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

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

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

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

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

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

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



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 eliminate the additional overhead by instead using the tinker console, a command line-based window into your Laravel application. Open tinker by executing the following command from your application's root directory:

```
$ php artisan tinker --env=local
Psy Shell v0.3.3 (PHP 5.5.18 â€" cli) by Justin Hileman
>>>
```

Notice tinker uses PsySH[63], a great interactive PHP console and debugger. PsySH is new to Laravel 5, and is a huge improvement over the previous console. Be sure to take some time perusing the feature list on the PsySH website[64] to learn more about what this great utility can do. In the meantime, let's get used to the interface:

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

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

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

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

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

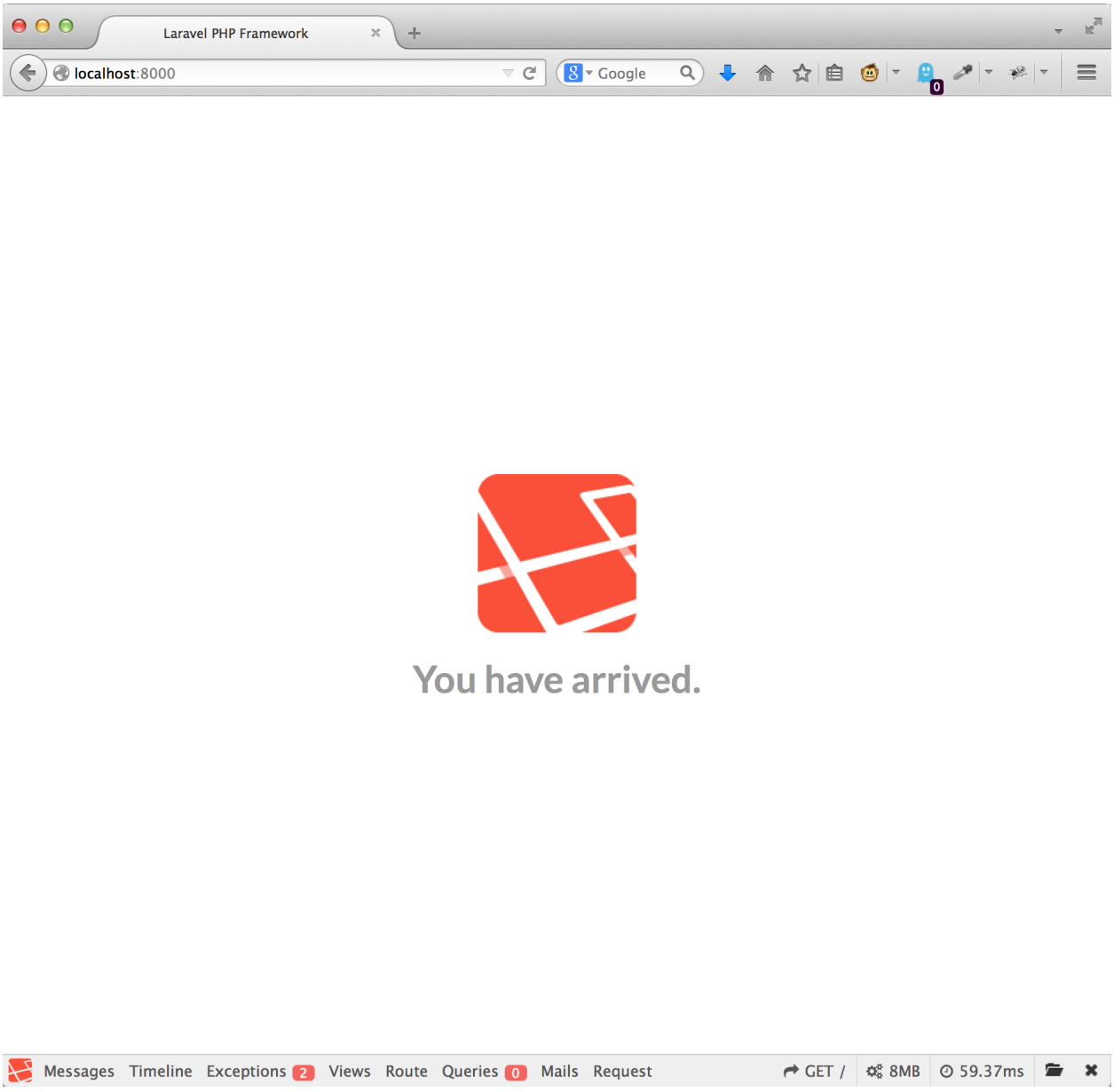
```
>>> exit
Exit:  Goodbye.
$
```

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

Introducing the Laravel Debugbar

It can quickly become difficult to keep tabs on the many different events that are collectively responsible for assembling the application response. You'll regularly want to monitor the status of database requests, routing definitions, view rendering, e-mail transmission and other activities. Fortunately, there exists a great utility called Laravel Debugbar[65] that provides easy access to the status of these events and much more by straddling the bottom of your browser window (see below screenshot).

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



The 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.

[66] http://getfirebug.com

- Views: This tab provides information about the various views used to render the page, including the layout.
- Route: This tab presents information about the requested route, including the corresponding controller and action.
- Queries: This tab lists the SQL queries executed in the process of serving the request.
- Mails: This tab presents information about any e-mails delivered while processing the request.
- Request: This tab lists information pertinent to the request, including the status code, request headers, response headers, and session attributes.

To install the Laravel Debugbar, execute the following command:

```
$ composer require barryvdh/laravel-debugbar
Using version ~1.8 for barryvdh/laravel-debugbar
./composer.json has been updated
...
Writing lock file
Generating autoload files
$
```

Next, add the following lines to the `providers` and `aliases` arrays, respectively:

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

...

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

Save the changes and reload 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`):

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

Save the changes and reload the home page within the browser. Check the Debugbar's `Messages` panel and you'll see the logged message! Like the Laravel logger, the Laravel Debugbar supports the log levels defined in PSR-3[67], meaning methods for debug, info, notice, warning, error, critical, alert and emergency are available.

Testing Your Laravel Application with PHPUnit

Automated testing is a critical part of today's web development workflow, and should not be ignored even for the most trivial of projects. Fortunately, the Laravel developers agree with this mindset and automatically include reference the PHPUnit package within every new Laravel project's `composer.json` file:

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

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

Running Your First Test

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

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

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

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

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

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

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

```
<?php

class ExampleTest extends TestCase {

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

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

}
```

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

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

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

.

Time: 94 ms, Memory: 10.25Mb

OK (1 test, 1 assertion)
```

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

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

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

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

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

F

Time: 234 ms, Memory: 10.25Mb

There was 1 failure:

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

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

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

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

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

Conclusion

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

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

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

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

Creating Your First View

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

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

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

```
13          color: #999;
14        }
15
16      </style>
17    </head>
18    <body>
19    <h1>Welcome to TODOParrot</h1>
20    </body>
21  </html>
```

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

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

```
1  <?php namespace todoparrot\Http\Controllers;
2
3  class WelcomeController extends Controller {
4
5    public function __construct()
6    {
7      $this->middleware('guest');
8    }
9
10   public function index()
11   {
12     return view('welcome');
13   }
14
15 }
```

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

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

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

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

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

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

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

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

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

Managing Your Application Routes

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

Introducing the routes.php File

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

```
<?php

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

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

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

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

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

Simultaneously Defining Multiple Routes

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

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

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

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

class ListsController extends Controller {

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

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

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

}
```

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

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

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

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

Defining Custom Routes

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

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

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

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

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

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

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

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

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

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

Creating Route Aliases

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

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

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

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

Listing Routes

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

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

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

Caching Routes

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

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

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

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

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

Introducing Route Annotations

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

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

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

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

Defining URL Parameters Using Annotations

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

[72] https://github.com/adamgoose/laravel-annotations/

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

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

Defining Route Aliases Using Annotations

You can define route aliases like so:

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

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

Creating Your First Controller

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

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

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

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

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

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

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

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

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

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

Introducing the Blade Template Engine

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

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

Displaying Variables

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

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

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

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

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

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

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

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

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

Escaping Dangerous Input

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

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

It would be rendered within like this, meaning the tags would be rendered to the browser as text rather than allowed to be interpreted by the browser:

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

If you wanted to output raw data, you would use triple brace syntax:

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

When rendered to the view, the triple brace syntax would result in the JavaScript-triggered alert box being displayed to the user! Therefore you should only output raw data when you're absolutely certain it does not originate from a potentially dangerous source. In any case, Laravel 5 uses a different escape syntax for this purpose:

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

As of Laravel 5, the triple brace syntax works identically to the double brace approach, escaping HTML entities by default.

Displaying Multiple Variables

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

```
public function index()
{

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

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

}
```

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

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

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

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

Determining Variable Existence

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

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

Looping Over an Array

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

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

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

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

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

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

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

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

If Conditional

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

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

Blade also supports the if-elseif-else construct:

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

Managing Your Application Layout

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

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

  @yield('content')

</body>
</html>
```

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

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

@section('content')

<h1>Welcome to TODOParrot</h1>

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

@stop
```

The `@extends` directive tells Laravel which layout should be used. Note how dot notation is used to represent the path, so for instance `layouts.master` translates to `layouts/master`. You specify the

layout because it's possible your application will employ multiple layouts, for instance one sporting a sidebar and another without.

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

Defining Multiple Layout Sections

A layout can identify multiple sections. For instance many web applications employ a main content area and a sidebar. In addition to the usual header and footer the layout might include some globally available sidebar elements, but you probably want the flexibility of appending view-specific sidebar content. This can be done using multiple `@section` directives in conjunction with `@show` and `@parent`. For reasons of space I'll just include the example layout's `<body>`:

```
<body>

  <div class="container">

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

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

</body>
</html>
```

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

```
@stop
@yield('advertisement')
```

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

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

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

@section('advertisement')
  @parent
    <p>
    Buy the TODOParrot Productivity guide for $10!
    </p>
@stop
```

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

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

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

Integrating Images, CSS and JavaScript

Your project images, CSS and JavaScript should be placed in the project's `public` directory. While you could throw everything into `public`, for organizational reasons I prefer to create `images`, `css`, and `javascript` directories. Regardless of where in the `public` directory you places these files, you're free to reference them using standard HTML or can optionally take advantage of a few helpers via the Laravel HTML component[73]. For instance, the following two statements are identical:

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

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

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

[73] https://github.com/illuminate/html

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

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

If you want to take advantage of the HTML helpers, you'll need to install the `Illuminate/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 `Illuminate/HTML` package to your `composer.json` file:

```
"require": {
  ...
  "illuminate/html": "5.0.*@dev"
},
```

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

```
'Illuminate\Html\HtmlServiceProvider',
```

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

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

With these changes in place, you can begin using the `Illuminate/HTML` package. Because at the time of this writing this feature was only recently separated from the native Laravel distribution, the documentation is still found on the Laravel website. Check out this section[74] of the Laravel documentation for a list of available helpers.

Integrating the Bootstrap Framework

Bootstrap[75] 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,

[74] http://laravel.com/api/class-Illuminate.Html.HtmlBuilder.html
[75] http://getbootstrap.com/

often customizing it with a Bootswatch theme[76]. TODOParrot is no exception, taking advantage of both Bootstrap and the Bootswatch Flatly[77] theme.

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

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

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

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

Once added you're free to begin taking advantage of the various CSS widgets and JavaScript plugins found in the Bootstrap documentation[78]. For instance try adding a stylized hyperlink to the `welcome.blade.php` view just to confirm everything is working as expected:

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

Integrating the Bootstrapper Package

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

[76] http://bootswatch.com/

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

[78] http://getbootstrap.com/

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[79], a great package created by Patrick Tallmadge[80]. Once installed you can use a variety of helpers to integrate Bootstrap widgets into your views. For instance the following helper will create a hyperlinked button:

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

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

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

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

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

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

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

[79] http://bootstrapper.aws.af.cm/

[80] https://github.com/patricktalmadge

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

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

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

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

Introducing Laravel Elixir

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

The Laravel 5 developers hope to reduce some of the time and hassle associated with these sort of tasks by providing a new API called Laravel Elixir[81]. The Elixir API integrates with Gulp, providing an easy solution for compiling your Laravel project's Less[82], Sass[83] and CoffeeScript[84], and perform any other such administrative task. In this section I'll show you how to create and execute several Elixir tasks in conjunction with your Laravel application. But first because many readers are likely not familiar with Gulp I'd like to offer a quick introduction, including instructions for installing Gulp and it's dependencies.

Introducing Gulp

Gulp[85] 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[86]. In this section I'll show you how to install and configure Gulp for subsequent use within Elixir.

Installing Gulp

Because Gulp is built atop Node.js[87], you'll first need to install Node. No matter your operating system this is easily done by downloading one of the installers via the Node.js website[88]. If you'd prefer to build Node from source you can download the source code via this link. If like me you're a Mac user, you can install Node via Homebrew. Linux users additionally likely have access to Node via their distribution's package manager.

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

[81] https://github.com/laravel/elixir
[82] http://lesscss.org/
[83] http://sass-lang.com/
[84] http://coffeescript.org/
[85] http://gulpjs.com/
[86] http://gulpjs.com/plugins/
[87] http://nodejs.org
[88] http://nodejs.org/download/

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

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

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

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

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

With Gulp installed it's time to install Elixir!

Installing Elixir

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

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

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

```
$ npm install
```

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

Creating Your First Elixir Task

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

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

The `mix.less` task compiles a Less[89] file, in this case the file named `app.less`. This file resides in `resources/assets/less`, and looks like this:

```
@import "bootstrap/bootstrap";

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

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

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

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

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

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

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

```
$ gulp --production
```

[89] http://lesscss.org/

Compiling Your JavaScript Assets

You'll likely also want to manage your JavaScript assets. For instance if you use CoffeeScript[90], you'll place your CoffeeScript files in `resources/assets/coffee` (you'll need to create this directory). Here's a simple CoffeeScript statement which will display one of those annoying alert boxes in the browser:

alert "Hi I am annoying"

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

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

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

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

Save the changes and run `gulp` again:

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

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

(function() { alert("Hello world");

}).call(this);

[90] http://coffeescript.org/

Watching for Changes

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

```
$ gulp watch
```

Other Elixir Tasks

Less and CoffeeScript compilation are but two of several Elixir features you can begin taking advantage of right now. Be sure to check out the Elixir README[91] for an extended list of capabilities.

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

Testing Your Views

Earlier versions of Laravel automatically included the BrowserKit[92] and DomCrawler[93] 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[94] package. Goutte is a web crawling library that can mimic a user's behavior in many important ways, and we an use it in conjunction with PHPUnit to functionally test the application. Open your project's `composer.json` file and add the following line:

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

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

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

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

```
<?php

use Goutte\Client;

class WelcomeTest extends TestCase {

}
```

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

```
public function testUserSeesWelcomeMessage()
{

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

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

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

}
```

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

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

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

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

```
public function testUserClicksContactLinkAndIsTakenToContactPage()
{

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

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

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

  $crawler = $client->click($link);

  $this->assertCount(1,
    $crawler->filter('h1:contains("Contact Us")'));

}
```

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

Conclusion

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

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 informs Laravel as to 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
<?php

return [

  'fetch' => PDO::FETCH_CLASS,

  'default' => 'mysql',

  'connections' => [

    'sqlite' => [
      'driver'   => 'sqlite',
      'database' => storage_path().'/database.sqlite',
      'prefix'   => '',
    ],

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

    'pgsql' => [
      'driver'   => 'pgsql',
      'host'     => env('DB_HOST', 'localhost'),
      'database' => env('DB_DATABASE', 'forge'),
      'username' => env('DB_USERNAME', 'forge'),
      'password' => env('DB_PASSWORD', ''),
      'charset'  => 'utf8',
      'prefix'   => '',
      'schema'   => 'public',
    ],

    'sqlsrv' => [
      'driver'   => 'sqlsrv',
      '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[95] 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[96] 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

Thanks to the Laravel 4 Generators package installed at the conclusion of Chapter 1, creating a model is easy. Let's kick this chapter 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:

[95] http://redis.io/

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

```
$ php artisan make:model Todolist
```

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

<?php namespace todoparrot;

```
use Illuminate\Database\Eloquent\Model;

class Todolist extends Model {

    //

}
```

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. This table is however not created automatically when the model is generated, so let's go ahead and do so now using Laravel's powerful *migrations* feature, introduced 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 create the `Todolist` model's corresponding database table. Typically the table names takes the plural form of the model name, therefore we'll create a table named `todolists`, doing so using the `artisan` CLI's `make:migration` command:

```
$ php artisan make:migration --create=todolists create_todolists_table
Created Migration: 2015_01_14_215705_create_todolists_table
```

Because we're creating a table, you can optionally pass along the table's name using the `--create` argument, otherwise you'd have to hand code the table creation statement within the migration after generation the migration file. The migration name `create_todolists_table` forms part of the migration file name, which is always prepended with the timestamp indicating the date and time in which the migration file was generated. This is important because Laravel will use these timestamps to definitively know the order in which migrations should be executed.

Once complete, a new migration file named something like `2015_01_14_215705_create_todolists_table.php` will be created and placed in the `database/migrations` directory. Open up this file and you'll see the following contents:

```
<?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');
```

```
30    }
31
32  }
```

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`[97] 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:

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

```
  Schema::create('todolists', function(Blueprint $table)
  {
    $table->increments('id');
    $table->string('name');
    $table->text('description');
    $table->timestamps();
  });
```

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:

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

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:

```
mysql> select * from migrations;
+-------------------------------------------------+-------+
| migration                                       | batch |
+-------------------------------------------------+-------+
| 2015_01_14_215705_create_todolists_table        |     1 |
+-------------------------------------------------+-------+
```

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.

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:

```
mysql> show tables;
+------------------------------+
| Tables_in_dev_todoparrot_com |
+------------------------------+
| todolists                    |
| migrations                   |
+------------------------------+
2 rows in set (0.03 sec)
```

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

```
mysql> describe todolists;
+-------------+------------------+------+-----+---------------------+-----------\
-----+
| Field       | Type             | Null | Key | Default             | Extra     \
     |
+-------------+------------------+------+-----+---------------------+-----------\
-----+
| id          | int(10) unsigned | NO   | PRI | NULL                | auto_incre\
ment |
| name        | varchar(255)     | NO   |     | NULL                |           \
     |
| description | text             | NO   |     | NULL                |           \
     |
| created_at  | timestamp        | NO   |     | 0000-00-00 00:00:00 |           \
     |
```

```
| updated_at  | timestamp        | NO   |     | 0000-00-00 00:00:00 |           \
    |
+-------------+------------------+------+-----+---------------------+-----------\
-----+
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:

```
$ php artisan migrate:rollback
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.

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:

```
$ php artisan migrate:status
+------+-----------------------------------------+
| Ran? | Migration                               |
+------+-----------------------------------------+
| □    | 2015_01_14_215705_create_todolists_table   |
| □    | 2014_01_15_155245_create_users_table       |
| □    | 2014_01_16_030306_add_user_id_to_todolists |
| □    | 2014_01_16_175008_create_tasks_table       |
+------+-----------------------------------------+
```

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:

```
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:

```
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`:

```
class User extends Model {

  public function setPasswordAttribute($password)
  {
      $this->attributes['password'] = \Hash::make($password);
  }

}
```

This example uses Laravel's `Hash` class to generate a hash (learn more about this class here[98]), 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:

```
$user = new User;
$user->password = 'blah';
echo $user->password;
$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:

```
use Carbon\Carbon;

...

class Todolist extends Model {

    public function isDueToday()
    {
      $now = \Carbon::now();
      if ($this->created_at->diff($now)->days == 0) {
        return true;
      } else {
        return false;
      }
    }

}
```

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

This example uses the fantastic Carbon[99] library, a PHP 5.3+ DateTime extension. The Carbon library was included within earlier versions of Laravel by default, however it has since been removed meaning you'll need to install it via Composer:

```
"require": {
  ...
  "nesbot/Carbon": "1.*"
},
```

After running `composer update` you'll have all of Carbon's capabilities at your disposal. With the `isDueToday` method in place, you can easily determine whether a list's due date has arrived:

```
$list = Todolist::find(12);

if ($list->isDueToday()) {
  echo "This list is due today!";
}
```

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:

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

```
class Todolist extends Model {

  private $rules = [
    'name' => 'required',
    'description' => 'required'
  ];

}
```

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:

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

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[100].

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:

```
class Todolist extends Model {

  ...

  public function validate()
  {

    $v = \Validator::make($this->attributes, $this->rules);
    if ($v->passes()) return true;
    $this->errors = $v->messages();
    return false;

  }
```

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

```
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:

```
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:

```
class Todolist extends Model {

  protected $guarded = ['some_important_attr'];

}
```

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:

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:

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

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

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

use todoparrot\Http\Requests;
use todoparrot\Http\Controllers\Controller;

class ListsController extends Controller {

    public function index()
    {
    }

    public function create()
    {
    }

    public function store()
    {
    }

    public function show($id)
    {
    }

    public function edit($id)
    {
    }

```

```
28      public function update($id)
29      {
30      }
31
32      public function destroy($id)
33      {
34      }
35
36  }
```

A controller is just a typical PHP class that extends Laravel's `BaseController` class. Because Laravel (and the Laravel 4 Generators package) 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.

If you're using the `routes.php` file to manage your route definitions, then all you need to do to register all of the `ListsController`'s routes is add the following line:

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

If you'd like to use the route annotations feature introduced in Chapter 2, all you need to do is define controller as RESTful in the class comment:

```
1  /**
2   * @Resource("lists")
3   */
4  class ListsController extends Controller {
5
6  ...
```

If you're using annotations, after saving the changes run the following command so Laravel knows to record any routing changes:

```
1  $ php artisan route:scan
2  Routes scanned!
```

Regardless of which route management solution you choose, if you try to access `http://homestead.app/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:

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

@section('content')

<h1>Lists</h1>

@stop
```

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

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

Save the file and navigate to `http://homestead.app/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:

```
$ php artisan tinker
Psy Shell v0.3.3 (PHP 5.5.18 — cli) by Justin Hileman
>>>
```

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

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

Next, assign a list name and description:

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

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

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

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

```
>>> $list->save();
=> 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:

```
>>> echo $list->id;
=> 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:

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

```
mysql> select * from todolists;
+----+----------+----------------------+---------------------+----------------\
----+
| id | name     | description          | created_at          | updated_at      \
   |
+----+----------+----------------------+---------------------+----------------\
----+
|  1 | San Juan | Pre-vacation planning | 2014-09-25 23:38:34 | 2014-09-26 16:05\
:24 |
+----+----------+----------------------+---------------------+----------------\
----+
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:

```
>>> 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:

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

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:

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

use Illuminate\Routing\Controller;
use todoparrot\Todolist;

class ListsController extends Controller {

...

}
```

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

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

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\Coll` 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:

```
<h1>Lists</h1>

@if ($lists->count() > 0)
  <ul>
    @foreach ($lists as $list)
      <li>{{ $list->name }}</li>
    @endforeach
  </ul>
@else
  <p>
    No lists found!
  </p>
@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 `http://homestead.app/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:

```
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      Model::unguard();
16
17      // $this->call('UserTableSeeder');
18    }
19
20  }
```

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:

```
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:

```
1  <?php namespace todoparrot;
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 line:

```
1  $this->call('\todoparrot\TodolistTableSeeder');
```

After saving the changes run the seeder anew:

```
1  $ php artisan db:seed
2  Seeded: \todoparrot\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[101] to easily create large amounts of sample data. Let's modify `TodolistTableSeeder.php` to generate 50 lists using Faker.

Begin by installing the Faker library using Composer. Modify your project's `composer.json` file, adding the Faker library to `require-dev` as demonstrated below:

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

Save the changes and run `composer update` to add Faker to your project. Next, we'll use Faker to create fifty lists. Modify the `TodolistTableSeeder`'s `run()` method to look like this:

[101] https://github.com/fzaninotto/Faker

```
class TodolistTableSeeder extends Seeder {

  public function run()
  {

    $faker = \Faker\Factory::create();

    Todolist::truncate();

    foreach(range(1,50) as $index)
    {

        Todolist::create([
          'name' => $faker->sentence(2),
          'description' => $faker->sentence(4),
        ]);

    }

  }

}
```

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[102]-style sentences consisting of two and four words, respectively.

After saving the changes, run `php artisan db:seed` again. After the command completes, check out your database and you should see fifty records that look like this:

[102] http://en.wikipedia.org/wiki/Lorem_ipsum

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

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[103] 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 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:

```
$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:

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

```
>> namespace todoparrot;
>>> $lists = Todolist::all();
=> <Illuminate\Database\Eloquent\Collection...
>>> foreach ($lists as $list) {
... printf("%s\n", $list->name);
... }
San Juan Vacation
Home Winterization
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:

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

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] > $list = todoparrot\Todolist::find(3);
[2] echo $list
{ "id":3,
  "name":"Rental Maintenance",
  "description":"Cleanup and improvements for new tenants",
  "created_at":"2014-10-02 17:38:55",
  "updated_at":"2014-10-02 17:38:55"
}
```

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:

```
public function show($id)
{
}
```

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:

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

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:

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

@section('content')

<h1>{{ $list->name }}</h1>

<p>
Created on: {{ $list->created_at }} Last modified: {{ $list->updated_at }}<br />
</p>
```

```
10
11  <p>
12  {{ $list->description }}
13  </p>
14
15  @stop
```

After saving the changes, navigate to `http://homestead.app/lists/1` 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: 2014-10-02 17:38:55 Last modified: 2014-10-02 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 `http://homestead.app/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/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:

```
protected function renderModelNotFoundException(ModelNotFoundException $e)
{

  if (view()->exists('errors.404'))
  {
    return response()->view('errors.404', [], 404);
  }
  else
  {
    return (new SymfonyDisplayer(config('app.debug')))
      ->createResponse($e);
  }
}
```

This method borrows heavily from the native `renderHttpException` method found in `vendor/laravel/framework/s` 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:

```
use Illuminate\Database\Eloquent\ModelNotFoundException;

...

public function render($request, Exception $e)
{
  if ($this->isHttpException($e))
  {
    return $this->renderHttpException($e);
  }
  elseif ($e instanceof ModelNotFoundException)
  {
    return $this->renderModelNotFoundException($e);
  }
  else
  {
    return parent::render($request, $e);
  }
}
```

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 `http://homestead.app/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:

```
> php artisan tinker
>>> $lists = todoparrot\Todolist::select('name', 'description')->first();
>>> echo $lists->name;
Sed voluptates.
>>> echo $lists;
{"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:

```
[4] > echo todoparrot\Todolist::count();
50
```

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

```
$lists = Todolist::all();
...
{{ $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:

```
$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`:

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

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

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

This is equivalent to executing the following SQL statement:

```
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:

```
$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:

```
$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:

```
$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:

```
mysql> select year(created_at) as `year`, count(name) as `count` from todolists
    -> group by `year` order by `count` desc;
+--------------+
| year | count |
+------+-------+
| 2014 |   247 |
| 2013 |   112 |
| 2012 |   92  |
| 2011 |   14  |
+---------------+
14 rows in set (0.00 sec)
```

This query can be reproduced in Laravel like so:

```
$lists = Todolist::select(DB::raw('year(created_at) as year'),
         DB::raw('count(name) as count'))
         ->groupBy('year')
         ->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:

```
@if ($lists->count() > 0)
  <ul>
    @foreach ($lists as $list)
      <li>{{ $list->count }} lists created in {{ $list->year }}</li>
    @endforeach
  </ul>
@else
  <p>
    No lists found!
  </p>
@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`:

```
$lists = Todolist::select(
          DB::raw('year(created_at) as year'),
          DB::raw('count(name) as count'))
          ->groupBy('year')
          ->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`:

```
$lists = Todolist::select(
          DB::raw('year(created_at) as year'),
          DB::raw('count(name) as count'))
          ->groupBy('year')
          ->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:

```
$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:

```
$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:

```
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:

```
$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:

```
$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:

```
$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:

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

This is equivalent to executing the following SQL:

```
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:

```
$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:

```
select count(*) as aggregate from `todolists`
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):

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

```
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!

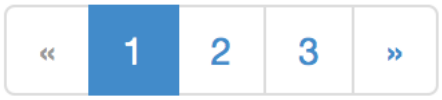
To paginate results, you'll use the `paginate` method:

```
$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:

```
{!! $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:

```
$list = new Todolist;
$list->name = 'San Juan Vacation';
$list->description = 'Pre-vacation planning';
$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:

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

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:

```
$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:

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

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

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:

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

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:

```
$ php artisan make:migration add_soft_delete_to_todolists --table=todolists
Created Migration: 2014_10_07_203253_add_soft_delete_to_todolists
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:

```
public function up()
{
  Schema::table('todolists', function(Blueprint $table)
  {
    $table->softDeletes();
  });
}

public function down()
{
  Schema::table('todolists', function(Blueprint $table)
  {
    $table->dropColumn('deleted_at');
  });
}
```

Save the changes and run the migration:

```
$ 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 namespace todoparrot;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletingTrait;

class Todolist extends Model {

    use SoftDeletingTrait;

    protected $dates = ['deleted_at'];

}
```

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:

```
$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:

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

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

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

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

```
$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:

```
$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`:

```
$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:

```
DB::insert('insert into todolists (name, description) values (?, ?)',
  array('San Juan Vacation', 'Things to do before vacation');

DB::update('update todolists set completed = 1 where id = ?', array(52));

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`:

```
$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[104] for a much more comprehensive summary of what's available.

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

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 whether your model accessors and mutators are properly configured, whether your custom methods are behaving as expected, that 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 that said, 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:

```
<php>
    ...
    <env name="DB_DATABASE" value="testing_todoparrot_com"/>
</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`.

Save these changes and then open the `tests/TestCase.php` file. This file is executed before any tests, bootstrapping the resources necessary to run your project in the test environment (including setting the environment to `testing`). We need to modify this file to populate the test database with the same tables used in the application, done by running the migration files found in `database/migrations`. To run these migrations we'll override the `setUp` method, ensuring it additionally executes `Artisan::call('migrate')` prior to each test. Additionally, we'll override the `tearDown` method, ensuring the tables are all removed following each test. Here's the modified `TestCase.php` file with the new `setUp()` and `prepareDatabase()` methods::

```
<?php

class TestCase extends Illuminate\Foundation\Testing\TestCase {

  public function setUp()
  {
      parent::setUp();
      Artisan::call('migrate');

  }

  public function tearDown()
  {
      parent::setUp();
      Artisan::call('migrate:reset');

  }

  public function createApplication()
  {
    $app = require __DIR__.'/../bootstrap/app.php';

    $app->make('Illuminate\Contracts\Console\Kernel')->bootstrap();

    return $app;
  }

}
```

After saving these changes we're ready to run our first model-oriented 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 `Todolist.php`, placing it in the `models` directory. Add the following contents to this file:

```
use todoparrot\todolist;

class TodolistTest extends TestCase
{

  public function testCanInstantiateTodolist()
  {

    $list = new Todolist;

    $this->assertEquals(get_class($list), 'todoparrot\Todolist');

  }

}
```

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:

```
vendor/bin/phpunit tests/models/TodolistTest.php
PHPUnit 4.5-dev by Sebastian Bergmann.

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

.

Time: 147 ms, Memory: 14.50Mb

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:

public function testNotValidWhenNameMissing() {

```
$t = new Todolist;

$this->assertFalse($t->validate());
```

}

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*[105], 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:

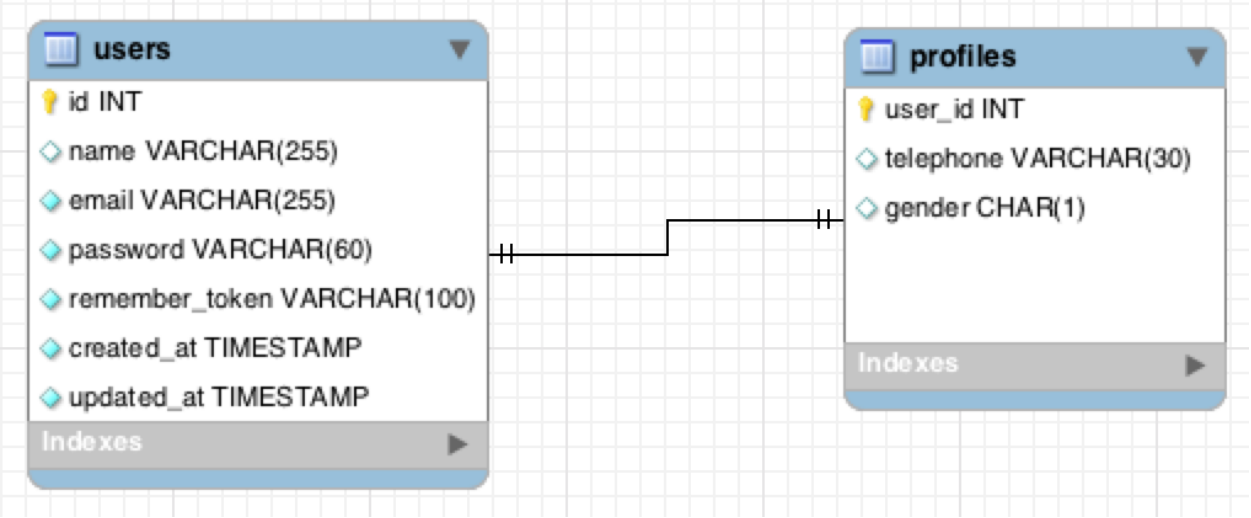
- The One-to-One Relation: One-to-One relations are used when one entity can only belong with one other entity. For instance for organizational reasons you might choose to separate user authentication information (e-mail address and password) from profile-related characteristics such as their name, bio 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.

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

- 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:

```
$ php artisan make:model Profile
```

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

<?php namespace todoparrot;

```
use Illuminate\Database\Eloquent\Model;

class Profile extends Model {

    //

}
```

Next, create the `Profile` model's underlying `profiles` table. We'll use Artisan's `make:migration` command to create the migration:

```
$ php artisan make:migration --create=profiles create_profiles_table
Created Migration: 2015_01_20_201647_create_profiles_table
```

Next, open up the newly created migration and modify the `up` method to look like this:

```
public function up()
{
  Schema::create('profiles', function(Blueprint $table)
  {
    $table->integer('user_id')->unsigned();
    $table->foreign('user_id')->references('id')->on('users');
    $table->string('name');
    $table->string('telephone');
    $table->timestamps();
  });
}
```

The bolded lines are the only two 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:

```
$ php artisan migrate
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:

```
class User extends Model {

  public function profile()
  {
    return $this->hasOne('todoparrot\Profile');
  }

}
```

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:

```
$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:

```
$profile = new Profile;
$profile->telephone = '614-867-5309';

$user = User::find(212);
$user->profile()->save($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:

```
$user = User::find(212);
$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:

```
$table->integer('user_id')->unsigned();
$table->foreign('user_id')->references('id')
      ->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

While the One-to-One relation's applicability is fairly limited, you'll repeatedly turn to the One-to-Many relation when building Laravel applications. Indeed the One-to-Many relation is used throughout the 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:

```
$ php artisan make:model Task
```

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

<?php namespace todoparrot;

```
use Illuminate\Database\Eloquent\Model;

class Task extends Model {

    //

}
```

Next, create the `Task` model's underlying `tasks` table. We'll use Artisan's `make:migration` command

```
$ php artisan make:migration --create=tasks create_tasks_table
Created Migration: 2015_01_20_210010_create_tasks_table
```

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

```
public function up()
{
  Schema::create('tasks', function(Blueprint $table)
  {
    $table->increments('id');
    $table->integer('todolist_id')->unsigned();
    $table->foreign('todolist_id')
      ->references('id')->on('todolists')
      ->onDelete('cascade');
    $table->string('name');
    $table->text('description');
    $table->boolean('done')->default(false);
    $table->timestamps();
  });
}

public function down()
{
  Schema::drop('tasks');
}
```

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:

```
$ php artisan migrate
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:

```
class Todolist extends Model {

  ...

  public function tasks()
  {
    return $this->hasMany('todoparrot\Task');
  }

}
```

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

```
class Task extends Model {

  ...

  public function todolist()
  {
    return $this->belongsTo('todoparrot\Todolist');
  }

}
```

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:

```
$list = Todolist::find(245);

$task = new Task;
$task->name = 'Walk the dog';
$task->description = 'Walk Barky the Mutt around the block';

$list->tasks()->save($task);

$task = new Task;
$task->name = 'Make tacos for dinner';
$task->description = 'Mexican sounds really yummy!';

$list->tasks()->save($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:

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

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:

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

@section('content')

<h1>{{ $list->name }}</h1>

<p>
Created on: {{ $list->created_at }}
Last modified: {{ $list->updated_at }}<br />
</p>

<p>
{{ $list->description }}
</p>

<h2>Tasks</h2>

@if ($list->tasks->count() > 0)
  <ul>
  @foreach ($list->tasks as $task)

    <li>{{ $task->name }}</li>

  @endforeach
  </ul>
@else
 <p>
  You haven't created any tasks.
  <a href="{{ URL::route('lists.tasks.create', [$list->id]) }}"
     class='btn btn-primary'>Create a task</a>
</p>
@endif

@stop
```

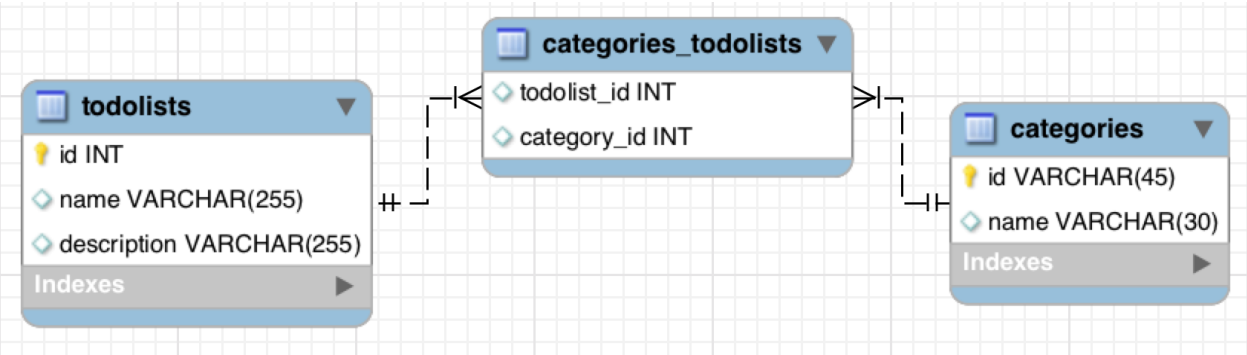
Filtering Related Records

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:

```
$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
```

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

<?php namespace todoparrot;

```
use Illuminate\Database\Eloquent\Model;

class Category extends Model {

  protected $fillable = [];

}
```

Next, let's create the `categories` table using Artisan's `make:migration` command:

```
$ php artisan make:migration --create=categories create_categories_table
Created Migration: 2015_01_20_214155_create_categories_table
```

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

```
public function up()
{
  Schema::create('categories', function(Blueprint $table)
  {
    $table->increments('id');
    $table->string('name');
    $table->timestamps();
  });
}
```

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

```
$ php artisan migrate
```

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

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

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

```
public function up()
{
  Schema::create('category_todolist', function(Blueprint $table)
  {
      $table->integer('category_id')->unsigned()->nullable();
      $table->foreign('category_id')->references('id')
            ->on('categories')->onDelete('cascade');

      $table->integer('todolist_id')->unsigned()->nullable();
      $table->foreign('todolist_id')->references('id')
            ->on('todolists')->onDelete('cascade');

      $table->timestamps();
  });
}
```

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

```
$ 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:

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

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:

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

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:

```
$tl = Todolist::find(1);

$category = new Category(array('name' => 'Vacation'));

$tl->categories()->save($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:

```
mysql> select * from categories;
+----+----------+---------------------+---------------------+
| id | name     | created_at          | updated_at          |
+----+----------+---------------------+---------------------+
|  1 | Vacation | 2014-11-03 20:44:11 | 2014-11-03 20:44:11 |
+----+----------+---------------------+---------------------+

mysql> select * from category_todolist;
+-------------+------------+---------------------+---------------------+
| category_id | todolist_id | created_at          | updated_at          |
+-------------+------------+---------------------+---------------------+
|           1 |          1 | 2014-11-04 20:50:54 | 2014-11-04 20:50:54 |
+-------------+------------+---------------------+---------------------+
```

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

```
$list = Todolist::find(2);

$category = Category::find(1)

$list->categories()->save($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:

```
$list = Todolist::find(2);

$category = Category::find(1)

// In this example we're passing in a Category object
$list->categories()->attach($category);

// The number 5 is the primary key of another category
$list->categories()->attach(5);
```

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

```
$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:

```
// Pass the Category object into the detach method
$list->categories()->detach(Category::find(3));

// Pass a category's ID
$list->categories()->detach(3);

// Pass along an array of category IDs
$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:

```
$list = Todolist::find(2);

$category = Category::find(1)

$list->categories()->save($category);
$list->categories()->save($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:

```
mysql> select * from category_todolist;
+-------------+-------------+---------------------+---------------------+
| category_id | todolist_id | created_at          | updated_at          |
+-------------+-------------+---------------------+---------------------+
|           1 |           2 | 2014-11-04 20:50:54 | 2014-11-04 20:50:54 |
|           1 |           2 | 2014-11-04 20:50:55 | 2014-11-04 20:50:55 |
+-------------+-------------+---------------------+---------------------+
```

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

```
$list = Todolist::find(2);

$category = Category::find(1)

if ($list->categories->contains($category))
{

  return Redirect::route('lists.show', array($list->id))
    ->with('message', 'Category could not be assigned. Duplicate entry!');

} else {

  $list->categories()->save($category);

  return Redirect::route('lists.show', array($list->id))
    ->with('message', 'The category has been assigned!');

}
```

Saving Multiple Relations Simultaneously

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

```
$list = Todolist::find(1);

$categories = [
  new Category(array('name' => 'Vacation')),
  new Category(array('name' => 'Tropical')),
  new Category(array('name' => 'Leisure')),
];

$list->categories()->saveMany($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:

```
$list = Todolist::find(2);

...

@if ($list->categories->count() > 0)

  <ul>

  @foreach($list->categories as $category)

    <li>{{ $category->name }}</li>

  @endforeach

  </ul>

@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:

```
$category = Category::find(2);

...

@if ($category->lists->count() > 0)

  <ul>

  @foreach($category->lists as $list)

    <li>{{ $list->name }}</li>

  @endforeach

  </ul>

@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:

```
$categories = [7, 12, 52, 77];

$tl = Todolist::find(2);

$tl->categories()->sync($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:

```
$ php artisan make:migration add_description_to_category_todolist_table
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:

```
public function up()
{
  Schema::table('category_todolist', function($table)
  {
      $table->string('description');
  });
}

public function down()
{
  Schema::table('category_todolist', function($table)
  {
      $table->dropColumn('description');
  });
}
```

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

```
$ php artisan migrate
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:

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

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:

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

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:

```
class User extends Model {

  public function country()
  {
    return $this->belongsTo('\todoparrot\Country');
  }

}

class Country extends Model {

  public function users()
  {
      return $this->hasMany('\todoparrot\User');
  }

}
```

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:

```
SELECT count(todolists.id), countries.name FROM todolists
  LEFT JOIN users on users.id = todolists.user_id
  LEFT JOIN countries ON countries.id = users.country_id
  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:

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

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:

```
$country = Country::where('name', 'Italy')->get()->first();

...

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

```
$ php artisan make:model Comment
```

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

```
<?php namespace todoparrot;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model {

    //

}
```

Next, generate the associated table:

```
$ php artisan make:migration --create=comments create_comments_table
Created Migration: 2015_01_20_223902_create_comments_table
```

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

```
Schema::create('comments', function(Blueprint $table)
{
  $table->increments('id');
  $table->text('body');
  $table->integer('commentable_id');
  $table->string('commentable_type');
  $table->timestamps();
});
```

Finally, save the changes and run the migration:

```
$ php artisan migrate
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:

```
class Comment extends Model {

    public function commentable()
    {
      return $this->morphTo();
    }

}
```

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:

```
class Todolist extends Model {

  public function comments()
  {
    return $this->morphMany('\todoparrot\Comment', 'commentable');
  }

}
```

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:

```
$list = Todolist::find(1);

$c = new Comment();

$c->body = 'Great work!';

$list->comments()->save($c);
```

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

```
mysql> select * from comments;
+----+-------------+---------------+--------------------+-----------+--------\
----+
| id | body        | commentable_id | commentable_type    | created_at | updated\
_at |
+----+-------------+---------------+--------------------+-----------+--------\
----+
|  1 | Great work! |             1 | todoparrot\Todolist | 2015-...   | 2015-..\
.   |
+----+-------------+---------------+--------------------+-----------+--------\
----+
```

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:

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

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

```
  @foreach ($list->comments as $comment)
    <p>
      {{ $comment->body }}
    </p>
  @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:

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

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

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

```
$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 way for you to cleanly package 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:

```
$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:

```
class Task extends Model
{

    public function scopeDone($query)
    {
        return $query->where('done', 1);
    }

}
```

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

```
$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:

```
class Task extends Model {

    public function scopeDone($query, $flag)
    {
        return $query->where('done', $flag);
    }

}
```

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

```
// Get completed tasks
$completedTasks = Task::done(true)->get();

// Get incomplete tasks
$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:

```
$list = Todolist::find(34);
$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. Forms Integration

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[106]. 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 is nearly identical to the form found on the Contact TODOParrot[107] page. It incorporates HTML5-specific markup[108], PHP-specific security features, and Bootstrap-specific markup[109] to produce a flexible, secure, and responsive form. Take a moment to examine the form markup before moving on to the summary found below.

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

<input name="_token" type="hidden" value="YLMaxbvKETQ4Tz6zVuWhd6XblhatvPtVVboSJv\
Hh">

<div class="form-group">
  <label for="name">Your Name</label>
  <input class="form-control" placeholder="Your name"
    name="name" type="text" required="required">
</div>
```

[106] http://todoparrot.com/contact
[107] http://todoparrot.com/contact
[108] http://diveintohtml5.info/forms.html
[109] http://getbootstrap.com/

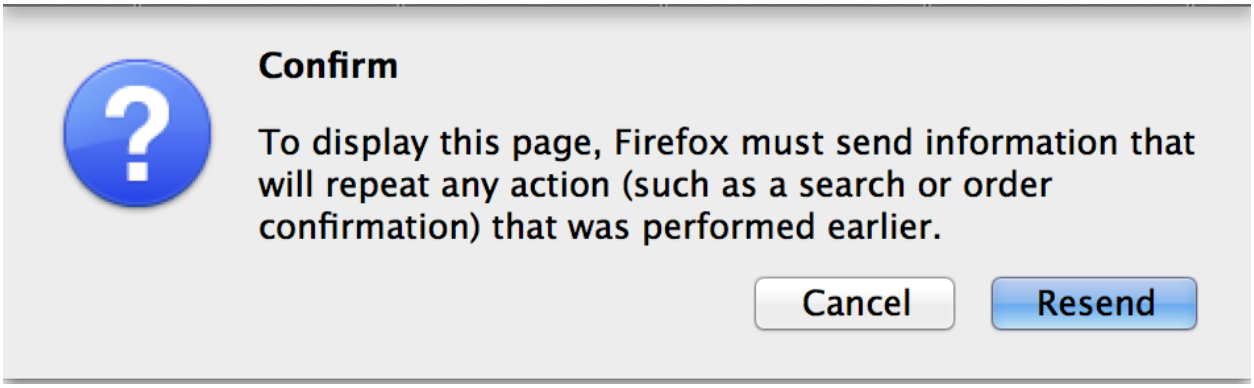
```
12
13  <div class="form-group">
14    <label for="email">Your E-mail Address</label>
15    <input class="form-control" placeholder="Your e-mail address"
16      name="email" type="text" required="required">
17  </div>
18
19  <div class="form-group">
20    <label for="message">Your Message</label>
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 projects `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)[110] 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:

[110] http://en.wikipedia.org/wiki/Cross-site_request_forgery

```
<label for="name">Your Name</label>
<input class="form-control" placeholder="Your name"
  name="name" type="text" required="required">

<label for="email">Your E-mail Address</label>
<input class="form-control" placeholder="Your e-mail address"
  name="email" type="text" required="required">

<label for="message">Your Message</label>
<textarea class="form-control" name="message"
  required="required"></textarea>
```

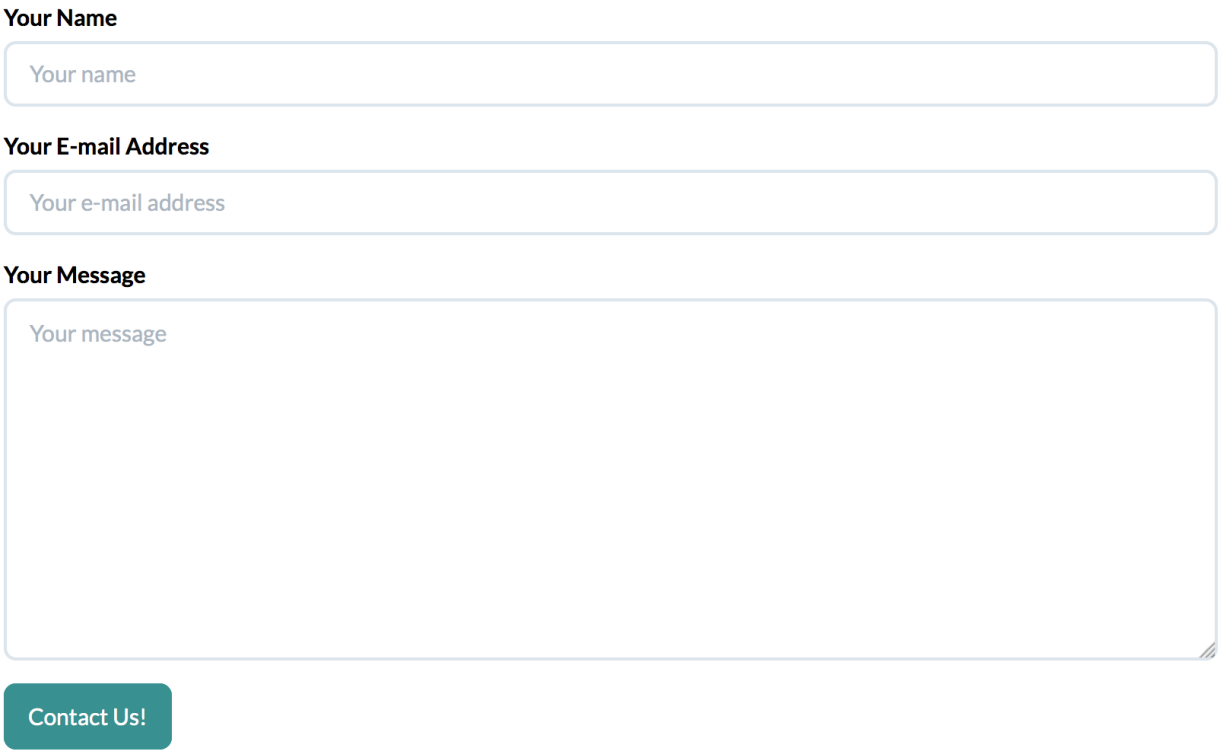
There's not much to discuss regarding these fields other than to mention all three fields are identified as `required`. This is an HTML5 feature that when implemented by browsers will automatically provide client-side validation without additional work on the part of the developer. I recommend you use 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

Because I'd imagine a fair number of readers have no prior experience creating a proper Laravel-powered form, let's create a simple contact form. The form we're going to build in this section also happens to be the same used for the contact form, located at http://todoparrot.com/contact[111] and consists of three fields, including the user's name, email address, and message (see below figure).

[111] http://todoparrot.com/contact



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[112]) 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:

[112] http://todoparrot.com/about

```
Route::get('contact',
  ['as' => 'contact', 'uses' => 'AboutController@create']);
Route::post('contact',
  ['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 namespace todoparrot\Http\Controllers;

class AboutController extends Controller {

    public function create()
    {
        return view('about.contact');
    }

    public function store()
    {
    }

}
```

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:

```
<h1>Contact TODOParrot</h1>

<ul>
    @foreach($errors->all() as $error)
        <li>{{ $error }}</li>
    @endforeach
</ul>

{!! Form::open(array('route' => 'contact_store', 'class' => 'form')) !!}

<div class="form-group">
    {!! Form::label('Your Name') !!}
    {!! Form::text('name', null,
        array('required',
              'class'=>'form-control',
              'placeholder'=>'Your name')) !!}
</div>

<div class="form-group">
    {!! Form::label('Your E-mail Address') !!}
    {!! Form::text('email', null,
        array('required',
              'class'=>'form-control',
              'placeholder'=>'Your e-mail address')) !!}
</div>

<div class="form-group">
    {!! Form::label('Your Message') !!}
    {!! Form::textarea('message', null,
        array('required',
              'class'=>'form-control',
              'placeholder'=>'Your message')) !!}
</div>

<div class="form-group">
    {!! Form::submit('Contact Us!',
      array('class'=>'btn btn-primary')) !!}
</div>
{!! Form::close() !!}
```

This form uses the form builder made available through Laravel's HTML component[113]. I explained how to install this component 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:

```
'Form'=> 'Illuminate\Html\FormFacade'
```

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:

```
{!! Form::open(array('route' => 'contact_store', 'class' => 'form')) !!}
...
{!! 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' => 'post'` 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:

```
<ul>
    @foreach($errors->all() as $error)
        <li>{{ $error }}</li>
    @endforeach
</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). 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.

[113] https://github.com/illuminate/html

Once you add this code to your project's `resources/views/about/contact.html.erb` file, navigate to `http://homestead.app/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:

```
$ php artisan make:request ContactFormRequest
Request created successfully.
```

This created a file named `ContactFormRequest.php` that resides in the directory `app/Http/Requests/ContactFormRe`
The class skeleton looks like this (comments removed):

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

use todoparrot\Http\Requests\Request;

class ContactFormRequest extends Request {

  public function authorize()
  {
    return false;
  }

  public function rules()
  {
    return [
      //
    ];
  }

}
```

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`:

```
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:

```
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 (`app/Http/Controllers/AboutCont` and modify the `store` method to look like this:

```
...

use todoparrot\Http\Requests\ContactFormRequest;

class AboutController extends Controller {

  public function store(ContactFormRequest $request)
  {

    return \Redirect::route('contact')
      ->with('message', 'Thanks for contacting us!');

  }

}
```

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:

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:

```
<ul>
  @foreach($errors->all() as $error)
    <li>{{ $error }}</li>
  @endforeach
</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:

```
@if(Session::has('message'))
    <div class="alert alert-info">
      {{Session::get('message')}}
    </div>
@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[114] 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[115] and Mandrill[116] 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.

Because you likely possess a Google Gmail account, I'll show you how to configure `config/mail.php` to send e-mail through a Gmail account. Change the following settings as directed:

- Change the `driver` setting to `smtp`. This is the default value.
- Change the `host` setting to `smtp.gmail.com`.
- Change the `port` setting to `465`.
- Change the `encryption` setting to `ssl`.
- Change the `username` setting to the Gmail e-mail address you'd like to use as the sender.

[114] http://swiftmailer.org/
[115] http://www.mailgun.com/
[116] https://mandrill.com/

- Change the `password` setting to your Gmail e-mail password. Keep in mind in a production environment you'll probably want to store this and other sensitive information in your project's `.env` file or as a server environment variable.

Save these changes, and then modify the `About` controller's `store` method to look like this:

```
public function store(ContactFormRequest $request)
{

    \Mail::send('emails.contact',
        array(
            'name' => $request->get('name'),
            'email' => $request->get('email'),
            'user_message' => $request->get('message')
        ), function($message)
    {
        $message->from('wj@wjgilmore.com');
        $message->to('wj@wjgilmore.com', 'Admin')->subject('TODOParrot Feedback'\
);
    });

  return \Redirect::route('contact')->with('message', 'Thanks for contacting us!\
');

}
```

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[117] for a complete explanation of the `Mail::send` method's features.

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:

[117] http://laravel.com/docs/master/mail

```
You received a message from TODOParrot.com:

<p>
Name: {{ $name }}
</p>

<p>
{{ $email }}
</p>

<p>
{{ $user_message }}
</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[118] 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 experience issues with Gmail, 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[119] to resolve the issue. Keep in mind however that you definitely do not want to 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:

```
$ php artisan make:controller ListsController
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:

[118] http://laravel.com/docs/master/mail

[119] https://www.google.com/settings/security/lesssecureapps

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

Save the changes to `routes.php` and then open the newly created controller (`app/Http/Controllers/ListsControll`). 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:

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

class ListsController extends Controller {

  public function create()
  {
  }

  public function store()
  {
  }

}
```

As a reminder, the `create` action is responsible for serving the form, and `store` is responsible for processing the submitted form data.

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

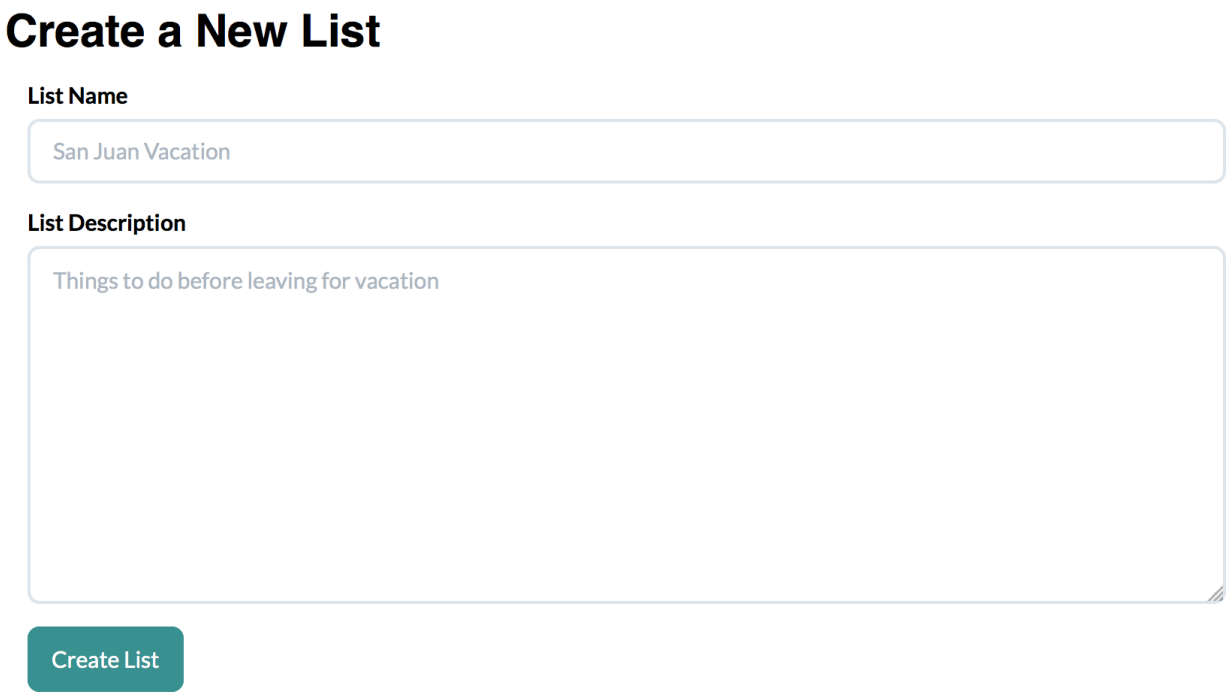
Before creating the form you'll first need to create the `List` controller's `create` view. Begin by creating a directory named `lists`, placing it in the directory `resources/views`. Inside this directory create a file named `create.blade.php` and add the following contents to it:

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

@section('content')

<h1>Create a New List</h1>

<ul>
    @foreach($errors->all() as $error)
        <li>{{ $error }}</li>
    @endforeach
</ul>

{!! Form::open(array('route' => 'lists.store', '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('Create List', array('class'=>'btn btn-primary')) !!}
</div>
{!! Form::close() !!}

@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:

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

After saving the changes to the `List` controller, navigate to `http://homestead.app/lists/create` and you should see the form presented in the below screenshot!



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:

```
$ php artisan make:request ListFormRequest
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 {

```
public function authorize()
{
    return false;
}

public function rules()
{
    return [
      //
    ];
}
```

}

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:

```
  public function rules()
  {
        return [
            'name' => 'required',
            'description' => 'required'
        ];
  }
```

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):

```
public function authorize()
{
  return true;
}
```

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:

```
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:

```
  public function edit($id)
  {
  }

  public function update($id)
  {
  }
```

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 tht used to create a new record, with one very important difference; instead of `Form::open` you'll use `Form::model`:

```
{!! Form::model($list, array('route' => ['lists.update', $list->id], 'class' => \
'form')) !!}
...
{!! 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:

```
public function edit($id)
{

   $list = Todolist::find($id);

   return view('lists.edit')->with('list', $list);

}
```

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:

```
  public function update($id, ListFormRequest $request)
  {

      $list = Todolist::find($id);

      $list->update([
        'name' => $request->get('name'),
        'description' => $request->get('description')
        ]);

      return \Redirect::route('lists.edit',
        array($list->id))->with('message', 'Your list has been updated!');

  }
```

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.

```
$ php artisan route:list
+--------+----------------------+--------------+----------------------------+--\
----------+
| Domain | URI                  | Name         | Action                     | M\
iddleware |
+--------+----------------------+--------------+----------------------------+--\
----------+
| ...    | ...                  | ...          | ...                        |  \
          |
|        | DELETE lists/{lists} | lists.destroy | ...ListsController@destroy |  \
          |
+--------+----------------------+--------------+----------------------------+--\
----------+
```

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:

```
{!! Form::open(array('route' => array('lists.destroy', $list->id), 'method' => '\
delete')) !!}
    <button type="submit">Delete List</button>
{!! 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:

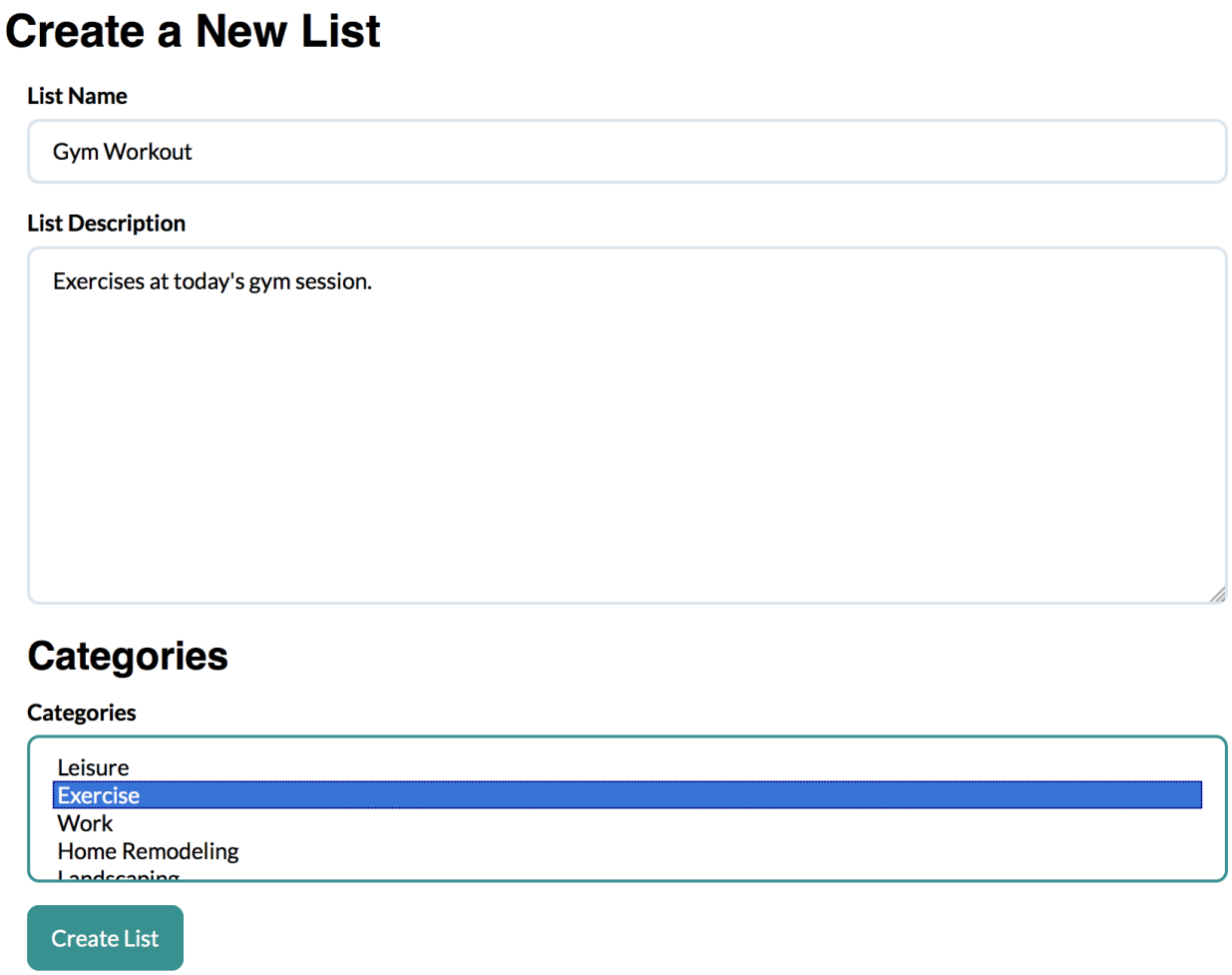
```
public function destroy($id)
{

  Todolist::destroy($id);

  return \Redirect::route('lists.index')
    ->with('message', 'The list has been deleted!');

}
```

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:

```
$list = Todolist::find(1);

$task = new Task;

$task->name = 'Walk the dog';
$task->description = 'Walk Barky the Mutt around the block';

$list->save();

// Associate categories 3 and 4 with this list
$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.



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:

```
{!! Form::open(array('route' => 'lists.store', '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>

<h3>Categories</h3>

<div class="form-group">
    {!! Form::label('Categories') !!}
    {!! Form::select('categories', $categories, null,
      array('multiple'=>'multiple','name'=>'categories[]')) !!}
</div>

<div class="form-group">
    {!! Form::submit('Create List', array('class'=>'btn btn-primary')) !!}
</div>
{!! Form::close() !!}
```

This newly added bit of code creates a multiple select box containing a list of categories:

```
{!! Form::select('categories[]', $categories, null,
  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:

```
<select name="categories[]" multiple="multiple">
  <option value="1">Leisure</option>
  <option value="2">Exercise</option>
  <option value="3">Work</option>
  <option value="4">Home Remodeling</option>
  <option value="5">Landscaping</option>
</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] > $c = \todoparrot\Category::lists('name', 'id');
// array(
//   1 => 'Leisure',
//   2 => 'Exercise',
//   3 => 'Work',
//   4 => 'Home Remodeling',
//   5 => 'Landscaping'
// )
```

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:

```
public function create()
{
  $categories = Category::lists('name', 'id');
  return view('lists.create')->with('categories', $categories);
}
```

Finally, you'll update the `Lists` controller's `store` action to ensure any desired categories are attached to the newly created list:

```
public function store(ListFormRequest $request)
{

  $list = new Todolist(array(
      'name' => $request->get('name'),
      'description' => $request->get('description')
  ));

  $list->save();

  if (count($request->get('categories')) > 0) {
    $list->categories()->attach($request->get('categories'));
  }

  return \Redirect::route('lists.create')
    ->with('message', 'Your list has been created!');

}
```

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.

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 two 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.

These are both known as route-level middlewares, because you can selectively apply them according to the specific route request. You'll find the two aforementioned 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' => 'App\Http\Middleware\Authenticate',
3    'auth.basic' => 'Illuminate\Auth\Middleware\AuthenticateWithBasicAuth',
4    'guest' => 'App\Http\Middleware\RedirectIfAuthenticated',
5  ];
```

The `Authenticated`, `AuthenticatedWithBasicAuth`, and `RedirectIfAuthenticated` middlewares are identified as route-level because you'll likely only want to use these in conjunction with specific routes. 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:

```
protected $middleware = [
  'Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode',
  'Illuminate\Cookie\Middleware\EncryptCookies',
  'Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse',
  'Illuminate\Session\Middleware\StartSession',
  'Illuminate\View\Middleware\ShareErrorsFromSession',
  'Illuminate\Foundation\Http\Middleware\VerifyCsrfToken',
];
```

The `EncryptCookies`, `AddQueuedCookiesToResponse`, `StartSession`, and `ShareErrorsFromSession` 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` (`vendor/laravel/framework/src/Illumina` class looks like this:

```
<?php namespace Illuminate\Foundation\Http\Middleware;

use Closure;
use Illuminate\Contracts\Routing\Middleware;
use Symfony\Component\HttpFoundation\Cookie;
use Illuminate\Contracts\Encryption\Encrypter;
use Illuminate\Session\TokenMismatchException;
use Symfony\Component\Security\Core\Util\StringUtils;

class VerifyCsrfToken implements Middleware {

  protected $encrypter;

  public function __construct(Encrypter $encrypter)
```

```
15    {
16      $this->encrypter = $encrypter;
17    }
18
19    public function handle($request, Closure $next)
20    {
21      if ($this->isReading($request) || $this->tokensMatch($request))
22      {
23        return $this->addCookieToResponse($request, $next($request));
24      }
25
26      throw new TokenMismatchException;
27    }
28
29    protected function tokensMatch($request)
30    {
31      $token = $request->session()->token();
32
33      $header = $request->header('X-XSRF-TOKEN');
34
35      return StringUtils::equals($token, $request->input('_token')) ||
36             ($header && StringUtils::equals
37             ($token, $this->encrypter->decrypt($header)));
38    }
39
40    protected function addCookieToResponse($request, $response)
41    {
42      $response->headers->setCookie(
43        new Cookie('XSRF-TOKEN', $request->session()->token(),
44        time() + 60 * 120, '/', null, false, false)
45      );
46
47      return $response;
48    }
49
50    protected function isReading($request)
51    {
52      return in_array($request->method(), ['HEAD', 'GET', 'OPTIONS']);
53    }
54
55  }
```

Every middleware solution implements the `Middleware` contract (found in your project's `vendor/laravel/framework` file). This contract contains a single 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/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(Guard $auth, Registrar $registrar)
2  {
3      $this->auth = $auth;
4      $this->registrar = $registrar;
5
6      $this->middleware('guest', ['except' => 'getLogout']);
7  }
```

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). If so, the user is redirected to the home page, otherwise the request is allowed to pass on without further interference:

```
public function handle($request, Closure $next)
{
  if ($this->auth->check())
  {
    return new RedirectResponse(url('/'));
  }

  return $next($request);
}
```

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:

```
$ php artisan make:middleware RequestLogger
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 namespace todoparrot\Http\Middleware;

use Closure;

class RequestLogger implements Middleware {

  public function handle($request, Closure $next)
  {
    //
  }

}
```

Modify the `handle` method to log the visitor's IP address to the Laravel log, and then pass on the request:

```
public function handle($request, Closure $next)
{
  \Log::info($request->ip());
  return $next($request);
}
```

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:

```
public function handle($request, Closure $next)
{
  $response = $next($request);
  \Log::info($request->ip());
  return $response;
}
```

Next, open up `app/Http/Kernel.php` and register the new `RequestLogger` middleware:

```
protected $routeMiddleware = [
  'auth' => 'todoparrot\Http\Middleware\Authenticate',
  'auth.basic' => 'Illuminate\Auth\Middleware\AuthenticateWithBasicAuth',
  'guest' => 'todoparrot\Http\Middleware\RedirectIfAuthenticated',
  'iplogger' => 'todoparrot\Http\Middleware\RequestLogger',
];
```

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:

```
class HomeController extends Controller {

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

  ...

}
```

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:

[2015-01-26 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`.

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.

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 `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 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;

  protected $table = 'users';

  protected $fillable = ['name', 'email', 'password'];

  protected $hidden = ['password', 'remember_token'];

}
```

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` (`vendor/laravel/framework` 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[120] 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:

mysql> describe users; +————————+————————+——+——+————+————————+ | Field | Type | Null | Key | Default | Extra | +————————+————————+——+——+————+————————+ | id | int(10) unsigned | NO | PRI | NULL | auto_increment | | name | varchar(255) | NO | | NULL | | | email | varchar(255) | NO | UNI | NULL | | | password | varchar(60) | NO | | NULL | | | remember_token | varchar(100) | YES | | NULL | | | created_at | timestamp | NO | | ... | | | updated_at | timestamp | NO | | ... | | +————————+————————+——+——+————+————————+ 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.

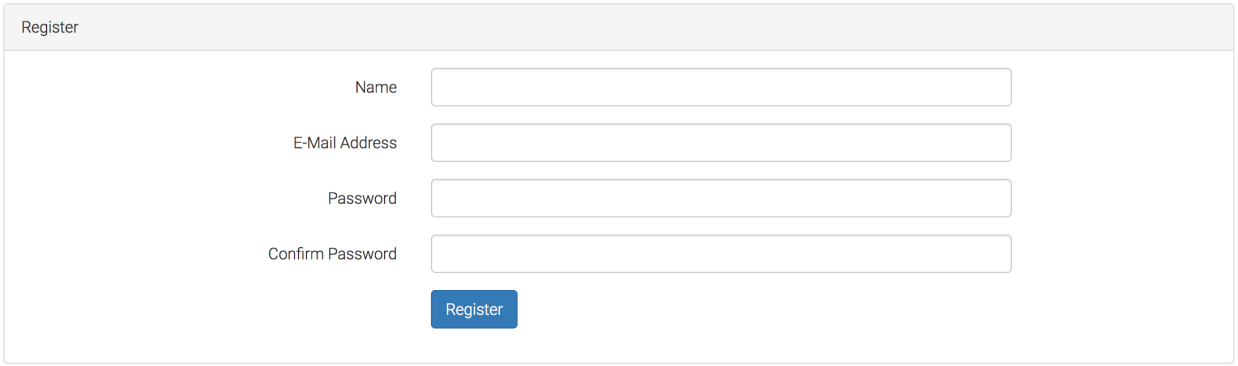
Introducing the Account Registration Feature

If you open `app/Http/routes.php` you'll find the following route definitions:

[120] http://culttt.com/2014/06/25/php-traits/

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

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`. Via this controller you'll be able to access endpoints such as `http://homestead.app/auth/register`, which renders the registration form found in the below screenshot.



The default registration view

As you can probably recognize, this form takes advantage of Bootstrap's stylistic form elements. However, you can easily modify the form to suit your requirements by updating the view found in `resources/views/auth/register.blade.php`.

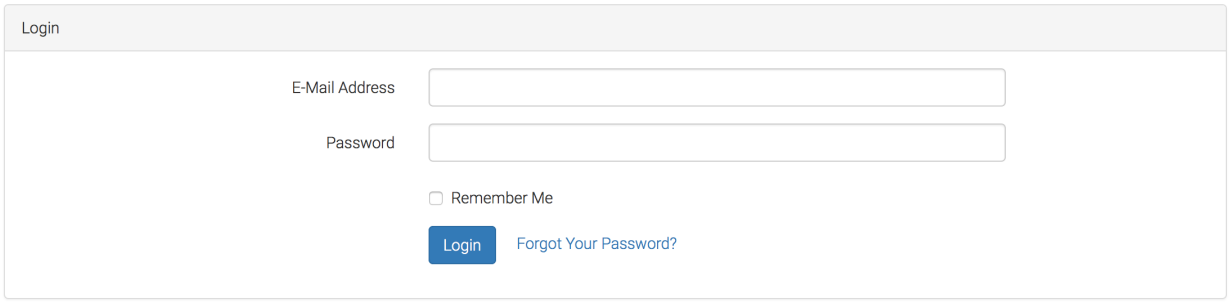
Go ahead and register, and you'll see that a new record will be inserted into the `users` table, and you'll automatically be logged into the site and redirected to the `Home` controller (`app/Http/Controllers/HomeContro`). If you open this `Home` controller you'll see the constructor references the `auth` middleware:

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

It's this middleware that allows you to so easily restrict access to certain controllers exclusively to authenticated users! To confirm this statement, go ahead and sign out by clicking on your name located at the top left of the page, and then selecting the `Logout` menu option. After doing so, try returning to `http://homestead.app/home` and you'll be immediately redirected to the login form.

Introducing the Account Sign In Feature

Users who have previously registered are able to sign into their account by navigating to `http://homestead.app/aut`. They'll be greeted with the login form presented in the below screenshot.



The default sign in view

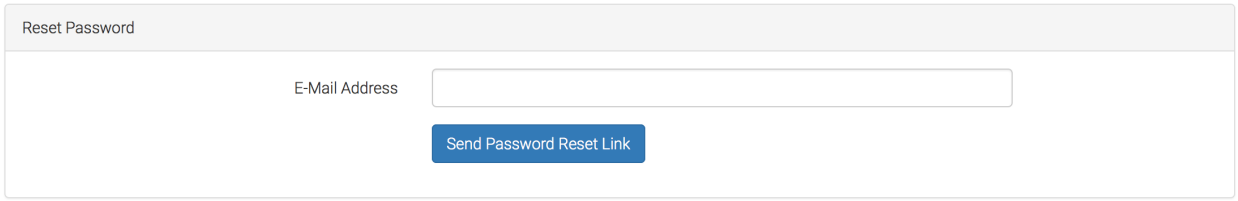
Go ahead and sign in using the account you created in the last section, and as before you'll be redirected to the `Home` controller. Like the registration form, you can easily modify the sign in view to suit your needs by editing the view found at `resources/views/auth/login.blade.php`.

Password Recovery

Recall that the `app/Http/routes.php` file defines a second authentication-related route:

```
1  Route::controllers([
2    ...
3    'password' => 'Auth\PasswordController',
4  ]);
```

The `Password` controller, defined in `app/Http/Controllers/PasswordController` is responsible for resetting a lost password. Users can initiate password recovery via the `http://homestead.app/password/email` route, which is accessible from the sign in form via the `Forgot Your Password?` link. The default password recovery form is presented in the below screenshot.



The password recovery view

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:

```
mysql> select * from password_resets;
+------------------+-----------------------------------------+----------------\
----+
| email            | token                                   | created_at      \
    |
+------------------+-----------------------------------------+----------------\
----+
| wj@wjgilmore.com | 1fed98e763aa83baeb572acf727a4ad16f71a310 | 2015-02-03 19:26\
:02 |
+------------------+-----------------------------------------+----------------\
----+
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:

```
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()`:

```
@if (Auth::check())
  Welcome back, {{ Auth::user()->name }}!
@else
  Hello, stranger! <a href="/auth/login">Login</a> or <a href="/auth/register">R\
egister</a>.
@endif
```

Conversely, you can flip the conditional around, instead `Auth::guest()` to determine if the user is a guest:

```
@if (Auth::guest())
  <li><a href="/auth/login">Login</a></li>
  <li><a href="/auth/register">Register</a></li>
@else
  Welcome back, {{ Auth::user()->name }}!
@endif
```

Restricting Access to Authenticated Users

As I briefly discussed in the earlier section, "Introducing the Account Registration Feature", you can easily restrict access to a controller by referencing the `auth` middleware in the appropriate controller constructor:

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

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

Recall from Chapter 5 the `authorize` method found in Laravel 5's generated Form Request classes. It is used to determine whether the form is available to all users or to some restricted subset:

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

use todoparrot\Http\Requests\Request;

class ContactFormRequest extends Request {

  public function authorize()
  {
    return false;
  }

  ...

}
```

If you'd like to restrict a form request to authenticated users, you can modify the `authorize()` method to look like this:

```
public function authorize()
{
  return Auth::check();
}
```

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:

```
get('/signup', array('as' => 'signup', 'uses' => 'Auth\AuthController@getRegiste\
r'));
post('/signup', array('as' => 'signup', 'uses' => 'Auth\AuthController@postRegis\
ter'));
```

After saving the changes you should be able to navigate to `http://homestead.app/signup` and see the registration form.

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[121]. 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[122]. To implement such a solution you would typically write a script using a package such as ApaiIO[123], and then schedule the script's execution using your server's Cron[124] 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:

[121] https://affiliate-program.amazon.com/

[122] https://affiliate-program.amazon.com/gp/advertising/api/detail/main.html

[123] https://github.com/Exeu/apai-io

[124] http://en.wikipedia.org/wiki/Cron

```
<?php namespace todoparrot\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel {

    protected $commands = [
        'todoparrot\Console\Commands\Inspire',
    ];

    protected function schedule(Schedule $schedule)
    {
        $schedule->command('inspire')->hourly();
    }

}
```

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:

```
$ php artisan inspire
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:

```
$ php artisan make:console UpdateCatalog --command=amazon:update
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 namespace todoparrot\Console\Commands;

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;

class UpdateCatalog extends Command {

  protected $name = 'amazon:update';

  protected $description = 'Command description.';

  public function __construct()
  {
    parent::__construct();
  }

  public function fire()
  {
    //
  }

  protected function getArguments()
  {
    return [
      ['example', InputArgument::REQUIRED,
        'An example argument.'],
    ];
  }

  protected function getOptions()
  {
    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:

```
protected function getArguments()
{
return [
  // ['example', InputArgument::REQUIRED, 'An example argument.'],
];
}
```

The Laravel documentation discusses Artisan command arguments, options, and other features. See this page[125] for more information.

Next, update the `fire` method to look like this:

```
  public function fire()
  {
    $this->info("Amazon catalog updated!");
  }
```

Save your changes and then register the command within `app/Console/Kernel.php`:

[125] http://laravel.com/docs/master/commands

```
protected $commands = [
    'todoparrot\Console\Commands\Inspire',
    'todoparrot\Console\Commands\UpdateCatalog'
];
```

After saving the changes you should see the custom command in the Artisan `list` output:

```
$ php artisan list
...
Available commands:
...
amazon
 amazon:update        Updates the TODOParrot book catalog.
...
```

You can now execute the `amazon:update` command from the terminal:

```
$ php artisan amazon:update
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:

```
protected function schedule(Schedule $schedule)
{
    $schedule->command('amazon:update')->hourly();
}
```

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`:

```
$schedule->command('amazon:update')->daily();
```

To run it at a specific time, use the `dailyAt` method:

```
$schedule->command('amazon:update')->dailyAt('18:00');
```

If you need to run a command very frequently, you can use an `every` method:

```
$schedule->command('amazon:update')->everyFiveMinutes();
$schedule->command('amazon:update')->everyTwentyMinutes();
```

See the Laravel documentation[126] 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:

```
* * * * * 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:

```
$schedule->call(function()
{
    // Send some e-mail

})->daily();
```

You can also schedule terminal commands for execution like so:

```
$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.

[126] 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[127]. 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 `storage/framework/services` 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
```

[127] 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 `storage/framework/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[128] 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:

[128] http://lesscss.org/

```
@import "bootstrap/bootstrap";

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

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

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`:

```
$ 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:

```
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[129].

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

[129] http://encosia.com/3-reasons-why-you-should-let-google-host-jquery-for-you/

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[130], and introduce you to Laravel Forge[131]. 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.

Deploying to Heroku

Heroku[132] 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"[133]). 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[134] 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[135]). 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/[136]). 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

[130] http://heroku.com
[131] https://forge.laravel.com
[132] http://heroku.com
[133] http://www.wjgilmore.com/books/easy-heroku-rails.html
[134] https://www.heroku.com/pricing
[135] https://signup.heroku.com
[136] https://toolbelt.heroku.com/

https://toolbelt.heroku.com/[137], 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`:

```
$ heroku
Usage: heroku COMMAND [--app APP] [command-specific-options]

Primary help topics, type "heroku help TOPIC" for more details:

addons    #  manage addon resources
apps      #  manage apps (create, destroy)
...
update       #  update the heroku client
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:

```
$ composer create-project laravel/laravel dev.herokutest.com dev-develop
Installing laravel/laravel (dev-develop 083db95...dac46617)
  - Installing laravel/laravel (dev-develop develop)
  Cloning develop
...
Compiling views
Do you want to remove the existing VCS (.git, .svn..) history? [Y,n]? Y
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:

[137] https://toolbelt.heroku.com/

```
web: vendor/bin/heroku-php-apache2 public
```

Incidentally, other options are available; see the Heroku PHP documentation[138] 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:

```
$ git init
Initialized empty Git repository in /Users/wjgilmore/Software/dev.herokutest.com\
/.git/
$ git add .
$ 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"[139] (free to read online) and checking out the interactive Git tutorial at https://try.github.io/[140].

With your repository created, it's time to deploy! Use the Heroku Toolbelt to initialize a new Heroku project:

```
$ heroku create
Creating lit-retreat-6653... done, stack is cedar-14
https://lit-retreat-6653.herokuapp.com/ | https://git.heroku.com/lit-retreat-665\
3.git
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:

[138] https://devcenter.heroku.com/articles/custom-php-settings#setting-the-document-root
[139] https://progit.org/
[140] https://try.github.io/

```
$ heroku config:add \
> BUILDPACK_URL=https://github.com/heroku/heroku-buildpack-php
Setting config vars and restarting lit-retreat-6653... done, v5
BUILDPACK_URL: https://github.com/heroku/heroku-buildpack-php
```

Finally, it's time to deploy! You can push your local changes to this remote by executing the following command:

```
$ git push heroku master
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 416 bytes | 0 bytes/s, done.
Total 5 (delta 4), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Fetching custom git buildpack... done
remote: -----> PHP app detected
remote: -----> No runtime required in composer.json, defaulting to PHP 5.6.5.
remote: -----> Installing system packages...
...
remote: -----> Launching... done, v6
remote:        https://lit-retreat-6653.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/lit-retreat-6653.git
   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:

```
$ heroku addons:add heroku-postgresql:hobby-dev
Adding heroku-postgresql:hobby-dev on lit-retreat-6653... done, v8 (free)
Attached as HEROKU_POSTGRESQL_NAVY_URL
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pgbackups:restore.
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[141].

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[142] 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:

```
$ heroku config --app lit-retreat-6653 | grep DATABASE_URL
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/[143] for installation instructions.

[141] https://addons.heroku.com/heroku-postgresql

[142] https://devcenter.heroku.com/articles/cleardb

[143] http://www.postgresql.org/

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 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`:

```
'default' => 'pgsql',

...

'pgsql' => [
    'driver'   => 'pgsql',
    'host'     => parse_url(getenv("DATABASE_URL"))["host"],
    'database' => substr(parse_url(getenv("DATABASE_URL"))["path"], 1),
    'username' => parse_url(getenv("DATABASE_URL"))["user"],
    'password' => parse_url(getenv("DATABASE_URL"))["pass"],
    'charset'  => 'utf8',
    'prefix'   => '',
    'schema'   => 'public',
],
```

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:

```
$ git add .
$ git commit -m "Updated database configuration"
$ git push heroku master
```

Next, you'll want to migrate the database. You can easily do this using the Heroku Toolbelt's `run` command:

```
$ heroku run php artisan migrate --app lit-retreat-6653
Running `php artisan migrate` attached to terminal... up, run.6981
*************************************
*     Application In Production!     *
*************************************

Do you really wish to run this command? [y/N] y
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
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/[144]) 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[145] and Papertrail[146] monitoring agents.
- Simple Scheduling: If you're using Laravel Queues[147] (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[148] or DigitalOcean[149] (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[150] 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[151].

[144] https://forge.laravel.com/
[145] http://newrelic.com/
[146] https://papertrailapp.com/
[147] http://laravel.com/docs/master/queues
[148] https://www.linode.com/
[149] https://www.digitalocean.com/
[150] https://www.digitalocean.com/pricing/
[151] http://mattstauffer.co/blog/getting-your-first-site-up-and-running-in-laravel-forge

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 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:

```
$ php artisan down
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:

```
$ php artisan up
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:

```
$ 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.