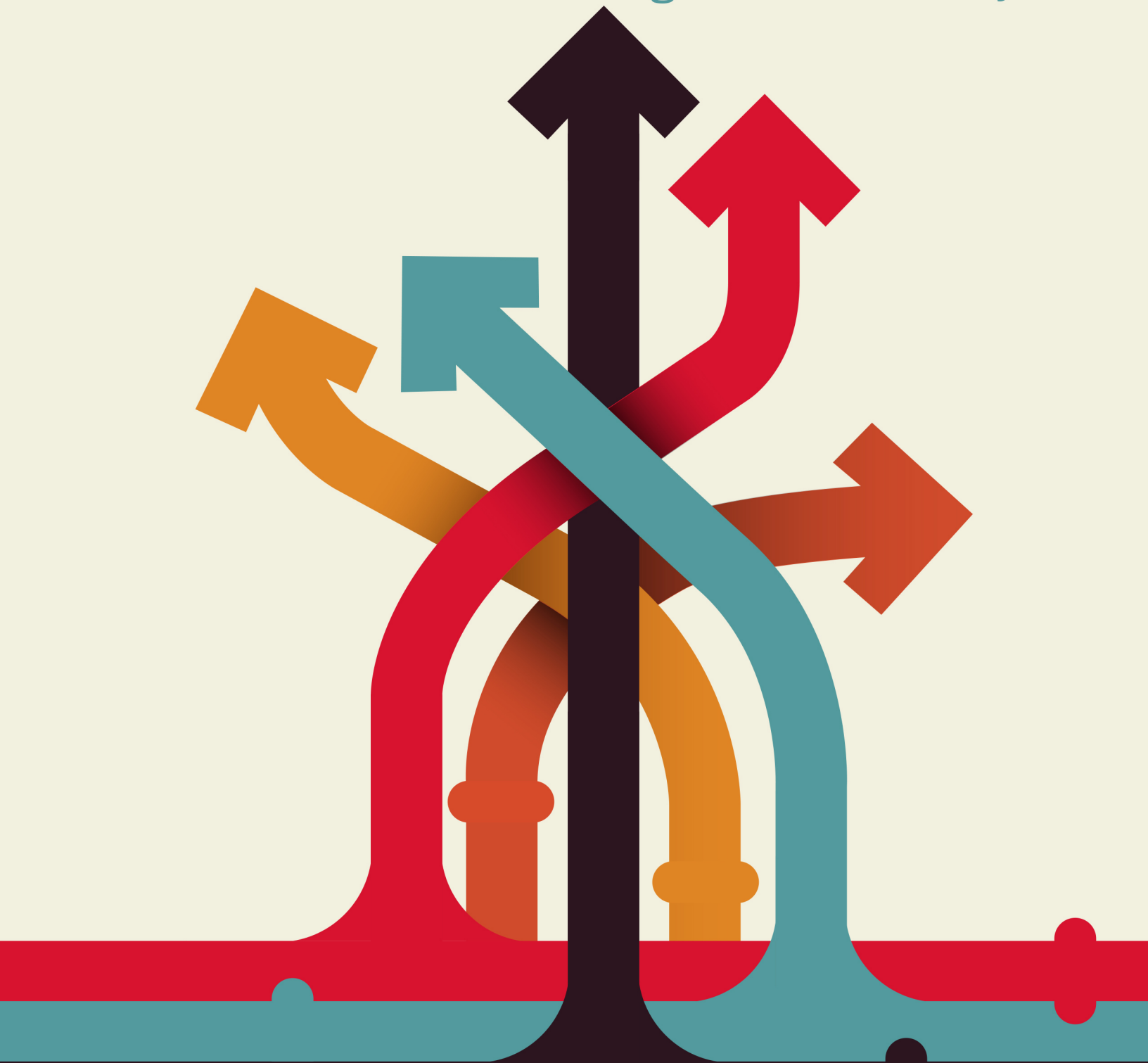


EASY LARAVEL 5

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 2016-10-10



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2014 - 2016 W. Jason Gilmore

Also By **W. Jason Gilmore**

[Easy Active Record for Rails Developers](#)

[Easy E-Commerce Using Laravel and Stripe](#)

[Easy React](#)

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

Contents

Introduction	1
What's New in Laravel 5?	2
About this Book	3
Introducing the TODOParrot Project	6
About the Author	6
Errata and Suggestions	6
Chapter 1. Introducing Laravel	7
Installing Laravel	7
Managing Your Laravel Project Development Environment	8
Creating the TODOParrot Application	21
Configuring Your Laravel Application	25
Useful Development and Debugging Tools	28
Testing Your Laravel Application with PHPUnit	37
Conclusion	39
Chapter 2. Managing Your Project Controllers, Layout, Views, and Other Assets	40
Creating Your First View	40
Creating Your First Controller	41
Managing Your Application Routes	43
Introducing the Blade Template Engine	49
Integrating Images, CSS and JavaScript	57
Introducing Elixir	59
Testing Your Views	66
Conclusion	68
Chapter 3. Introducing Laravel Models	69
Configuring Your Project Database	69
Introducing the Eloquent ORM	72
Creating Your First Model	72
Introducing Migrations	73
Creating a RESTful Controller	80
Finding Data	87
Inserting New Records	98

CONTENTS

Updating Existing Records	101
Deleting Records	101
Introducing Query Builder	104
Defining Accessors, Mutators, and Methods	105
Seeding the Database	108
Creating Sluggable URLs	112
Testing Your Models	115
Summary	120
Chapter 4. Model Relations, Scopes, and Other Advanced Features	121
Introducing Relations	121
Introducing One-to-One Relations	122
Introducing the Belongs To Relation	126
Introducing One-to-Many Relations	127
Introducing Many-to-Many Relations	132
Introducing Has Many Through Relations	142
Introducing Polymorphic Relations	143
Eager Loading	146
Introducing Scopes	149
Summary	151
Chapter 5. Integrating Web Forms	152
Web Form Fundamentals	152
Creating a User Feedback Form	155
Creating New Lists	167
Updating a TODO List	172
Deleting TODO Lists	175
Associating Tasks with Categories	176
Uploading Files	181
Summary	184
Chapter 6. Introducing Middleware	185
Introducing Laravel's Default Middleware	185
Creating Your Own Middleware Solution	190
Using Middleware Parameters	192
Summary	193
Chapter 7. Authenticating and Managing Your Users	194
Registering Users	194
Retrieving the Authenticated User	204
Restricting Access to Authenticated Users	205
Restricting Forms to Authenticated Users	206
Adding Custom Fields to the Registration Form	207
Summary	209

CONTENTS

Chapter 8. Creating a Restricted Administration Console	210
Identifying Administrators	210
Creating the Administration Controllers	211
Restricting Access to the Administration Console	213
Summary	214
Chapter 9. Deploying, Optimizing and Maintaining Your Application	215
Introducing the Laravel 5 Command Scheduler	215
Optimizing Your Application	220
Deploying Your Application	223
Placing Your Application in Maintenance Mode	230
Summary	231
Chapter 10. Introducing Lumen	232
Creating Your First Lumen Application	233
Creating a Status API	235
Integrating the Lumen Application Into TODOParrot.com	239
Summary	240
Chapter 11. Introducing Events	241
Creating an Event	241
Broadcasting Events	247
Summary	251
Chapter 12. Introducing Vue.js	252
Installing Vue.js	252
Refactoring the List Tasks Manager	255
Where to From Here?	267
Summary	267
Appendix A. Deploying Your Laravel Application to DreamHost	268
Deploying Your Project to DreamHost	269
Summary	284
Appendix B. Feature Implementation Cheat Sheets	285
Creating a Model	285
Creating a Migration	286
Seeding the Database	286
Creating a Resource Controller	288
Creating and Validating a Form	289

Introduction



ATTENTION: I'm in the middle of a major book update for Laravel 5.3. The first five chapters have been updated, and I continue working on chapter 6 and beyond. As a reader, you'll always receive free updates so stay tuned for update notifications!

I've spent the vast majority of the past 16 years immersed in the PHP language. During this time I've written eight PHP-related books, including a bestseller that has been in print for more than eleven 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 personally delivered training sessions to hundreds of developers. So you might be surprised to learn a few years ago I became rather disenchanted with the very language that for so very long had consumed the better part of my professional career. 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, old habits are hard to break and so during this time I kept tabs on the PHP community, watching with great interest as several talented developers worked tirelessly to inject that missing enthusiasm back into the language. During my time in the wilderness Nils Adermann and Jordi Boggiano released the [Composer](https://getcomposer.org/)¹ dependency manager. The [Framework Interoperability Group](http://www.php-fig.org/)² was formed. And in 2012 the incredibly talented [Taylor Otwell](http://taylorotwell.com/)³ created the [Laravel framework](http://laravel.com/)⁴ which quickly soared in popularity to become the most followed PHP project on GitHub, quickly surpassing projects and frameworks that had been under active development 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 a relative newcomer to the PHP framework landscape, Laravel is incredibly polished, offering a shallow learning curve, easy test integration, a great object-relational mapping solution called Eloquent, and a wide variety of other great features. 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/>

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](https://github.com/vlucas/phpdotenv)⁵ 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.
- **Elixir:** [Elixir](https://github.com/laravel/elixir)⁶ offers Laravel users a convenient way to automate various development tasks using [Gulp](http://gulpjs.com/)⁷, 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](https://github.com/thephpleague/flysystem)⁸, 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 form data. In chapter 5 I'll introduce you to this great new feature.
- **Middleware:** 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. You'll learn all about Laravel authentication in chapter 6.
- **Event handling:** Laravel 5 event handlers allow you to reduce redundant logic otherwise found in your controllers by packaging bits of logic separately and then executing that logic in conjunction following certain events, such as sending an e-mail following the registration of a new user. In chapter 11 you'll learn how to create an event handler and then integrate a corresponding event listener into your code.
- **The Lumen Microframework:** Although not part of the Laravel framework per se, Lumen is an optimized version of Laravel useful for creating incredibly fast micro-services and REST APIs. I'll introduce you to this great framework in chapter 10.

⁵<https://github.com/vlucas/phpdotenv>

⁶<https://github.com/laravel/elixir>

⁷<http://gulpjs.com/>

⁸<https://github.com/thephpleague/flysystem>

But we're not going to stop with a mere introduction to these new features. I want you to learn how to build *real-world* Laravel applications, and so I additionally devote extensive coverage to about topics such as effective CSS and JavaScript integration, automated testing, and more!

About this Book

This book is broken into twelve chapters and an appendix, each of which is briefly described below.

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](http://gulpjs.com/)⁹ 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.

⁹<http://gulpjs.com/>

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.

Chapter 5. Integrating Web Forms

Your application will almost certainly contain at least a few web forms, which will likely interact with the models, meaning you'll require a solid grasp on Laravel's form generation and processing capabilities. While creating simple forms is fairly straightforward, things can complicated fast when implementing more ambitious solutions such as forms involving multiple models. In this chapter I'll go into extensive detail regarding how you can integrate forms into your Laravel applications, introducing Laravel 5's new form requests feature, covering both Laravel's native form generation solutions as well as several approaches offered by popular packages. You'll also learn how to upload files using a web form and Laravel's fantastic file upload capabilities.

Chapter 6. Integrating Middleware

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

Chapter 7. Authenticating and Managing Your Users

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

Chapter 8. Creating a Restricted Administration Console

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

Chapter 9. Deploying, Optimizing and Maintaining Your Application

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

Chapter 10. Introducing the Lumen Microframework

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

Chapter 11. Introducing Events

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

Chapter 12. Introducing Vue.js

[Vue.js](http://vuejs.org/)¹⁰ has become the Laravel community’s de facto JavaScript library, and for good reason; it shares many of the practical, productive attributes Laravel developers have come to love. Chapter 12 introduces Vue.js’ fundamental features, and shows you how to integrate highly interactive and eye-appealing interfaces into your Laravel application.

Appendix B. Feature Implementation Cheat Sheets

The book provides occasionally exhaustive explanations pertaining to the implementation of key Laravel features such as controllers, migrations, models and views. However, once you understand the fundamentals it isn’t really practical to repeatedly reread parts of the book just to for instance recall how to create a model with a corresponding migration or seed the database. So I thought it might be useful to provide an appendix which offered a succinct overview of the steps necessary to carry out key tasks. This is a work in progress, but already contains several pages of succinct explanations.

¹⁰<http://vuejs.org/>

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](http://todoparrot.com)¹¹, a web-based task list application built atop Laravel.

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

About the Author

[W. Jason Gilmore](http://www.wjgilmore.com)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 SaaS for the interior design and architecture industries, an e-commerce analytics application for a globally recognized publisher, an intranet application for a major South American avocado farm, and a 10,000+ product online store.

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

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](http://www.codemash.org)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.

¹¹<http://todoparrot.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. Newcomers to framework-driven development will have a slightly steeper learning curve due to the introduction of new concepts, however I promise Laravel's practical and user-friendly features will make your journey an enjoyable one.

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

Installing Laravel

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

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

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

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

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

```
1 $ laravel --version
2 Laravel Installer version 1.3.1
```

You'll primary use the `laravel` CLI to generate new Laravel projects, which you can do with the `new` command:

```
1 $ laravel new igniterental
2 Crafting application...
3 Loading composer repositories with package information
4 Installing dependencies (including require-dev) from lock file
5 ...
6 Application ready! Build something amazing.
```

If you peek inside the `igniterental` directory you'll see all of the files and directories which comprise a Laravel application! While I know diving into the code found in this newly generated project is very tantalizing, I *implore* you to be patient and take time to first learn more about how to manage your locally hosted Laravel projects. *Homestead* and *Valet* are Laravel's two standard solutions, and I'll introduce you to both next.

Managing Your Laravel Project Development Environment

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 5.6.4 or newer and one of Laravel's supported databases (MySQL, PostgreSQL, SQLite, and Microsoft SQL Server). While those of you who are seasoned PHP developers likely already have local versions of this software installed on your development laptop, I'd like to recommend two *far more efficient* solutions which completely eliminate the need to manage this software on your own. Fortunately for newcomers these solutions will be equally welcome since it allows you to avoid the often time-consuming and error-prone process of installing and configuring PHP, MySQL, and a web server. These solutions are *Homestead* and *Valet*, and in this section you'll learn about both.

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 [NGINX](http://nginx.org/)¹⁵, a database server such as [MySQL](http://www.mysql.com/)¹⁶ or [PostgreSQL](http://www.postgresql.org/)¹⁷, and often a variety of supplemental technologies such as [Redis](http://redis.io/)¹⁸ and [Grunt](http://gruntjs.com/)¹⁹. 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.

¹⁵<http://nginx.org/>

¹⁶<http://www.mysql.com/>

¹⁷<http://www.postgresql.org/>

¹⁸<http://redis.io/>

¹⁹<http://gruntjs.com/>

In recent years *virtual machines* dramatically lowered this bar. A virtual machine is a software-based implementation of a computer that can be run inside the confines of another computer (such as your laptop), or even inside another virtual machine. This is incredible technology, because for instance you can use a virtual machine to run an Ubuntu Linux server on your Windows 10 laptop, or vice versa. Further, it's possible to create a customized virtual machine image preloaded with a select set of software. This image can then be distributed to fellow developers, who can run the virtual machine and take advantage of the custom software configuration. This is precisely what the Laravel developers have done with [Homestead](#)²⁰, a virtual machine which bundles everything you need to get started building Laravel-driven websites.

Homestead is currently based on Ubuntu 16.04, and includes everything you need to get started building Laravel applications, including PHP 7.0, Nginx, MySQL, PostgreSQL and a variety of other useful utilities such as Redis and Memcached. It runs flawlessly on OS X, Linux and Windows, and the installation process is very straightforward, meaning in most cases you'll be able to begin managing Laravel applications in less than 30 minutes.



Mac users have another option at their disposal when it comes to locally hosting a Laravel application. It's called Valet, and later in this chapter I'll introduce Valet. If you're looking for a no-frills hosting environment during the development process, Valet is way to go although for most readers I still recommend spending the extra time and effort required to install Homestead because of all the additional features it has to offer.

Installing Homestead

Homestead requires [Vagrant](#)²¹ and [VirtualBox](#)²². User-friendly installers are available for all of the common operating systems, including OS X, Linux and Windows. Take a moment now to install Vagrant and VirtualBox. Once complete, open a terminal window and execute the following command:

```
1 $ vagrant box add laravel/homestead
2 ==> box: Loading metadata for box 'laravel/homestead'
3     box: URL: https://atlas.hashicorp.com/laravel/homestead
4 This box can work with multiple providers! The providers that it
5 can work with are listed below. Please review the list and choose
6 the provider you will be working with.
7
8 1) virtualbox
9 2) vmware_desktop
10
```

²⁰<http://laravel.com/docs/homestead>

²¹<http://www.vagrantup.com/>

²²<https://www.virtualbox.org/wiki/Downloads>


```
11 Enter your choice: 1
12 ==> box: Adding box 'laravel/homestead' (v0.4.2) for provider: virtualbox
13     box: Downloading: https://atlas.hashicorp.com/laravel/boxes/homestead/versio\
14 ns/0.4.2/providers/virtualbox.box
15 ==> box: Successfully added box 'laravel/homestead' (v0.4.2) for 'virtualbox'!
```



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

This command installs the Homestead *box*. A box is just a term used to refer to a Vagrant package. Packages are the virtual machine images that contain the operating system and various programs. The Vagrant community maintains hundreds of different boxes useful for building applications using a wide variety of technology stacks, so check out this [list of popular boxes](https://vagrantcloud.com/discover/popular)²³ for an idea of what else is available.

Once the box has been added, you'll next want to install Homestead. To do so, you'll ideally using Git to clone the repository. If you don't already have Git installed you can easily do so by heading over to the [Git website](https://git-scm.com/downloads)²⁴ or using your operating system's package manager.

Next, open a terminal and enter your home directory:

```
1 $ cd ~
```

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

```
1 $ git clone https://github.com/laravel/homestead.git Homestead
2 Cloning into 'Homestead'...
3 remote: Counting objects: 1497, done.
4 remote: Compressing objects: 100% (5/5), done.
5 remote: Total 1497 (delta 0), reused 0 (delta 0), pack-reused 1492
6 Receiving objects: 100% (1497/1497), 241.74 KiB | 95.00 KiB/s, done.
7 Resolving deltas: 100% (879/879), done.
8 Checking connectivity... done.
```

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

²³<https://vagrantcloud.com/discover/popular>

²⁴<https://git-scm.com/downloads>

```
1 $ bash init.sh
```

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

```
1 $ init.bat
```

This will create a directory called `.homestead`, which will also be placed in your home directory. You'll modify the files found in this directory to configure Homestead in a variety of ways, including most notably how to find and serve the web applications which will be hosted on the virtual machine.

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](#)²⁵ offers a succinct set of steps. If you're running Linux or OS X, [nixCraft](#)²⁶ offers a solid tutorial.

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

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

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

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

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

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

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

²⁵http://kb.siteground.com/how_to_generate_an_ssh_key_on_windows_using_putty/

²⁶<http://www.cyberciti.biz/faq/how-to-set-up-ssh-keys-on-linux-unix/>

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

This particular step tends to be the source of great confusion Homestead beginners, so pay close attention to the following description. The `folders` structure'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* identify your Laravel project's root directory (I realize we haven't created the project or directory just yet, so just keep in mind this value must identify that soon-to-be-created directory). The `folders` structure'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. You almost certainly do not have to change the `to` attribute's default value, so don't worry about it for now.



Windows users should keep in mind the tilde (`~`) home directory shortcut is not supported on Windows and so you'll need to specify the absolute path to your chosen directory.

The `sites` structure's `map` attribute defines the domain name you'll use to access the Laravel application via the browser. For instance, you might change this to `dev.todoparrot.com`. Keep in mind this domain name is used purely for internal developmental purposes, so you don't actually have to own the domain name.

Finally, the `sites` structure's `to` attribute defines the Laravel project's root *web directory*, which is `/public` by default. This isn't just some contrived setting; a file named `index.php` resides in your Laravel application's `/public` directory, and it "listens" for incoming requests to your application and kicks off the process which ultimately results in the client's desired resource (web page, JSON data, etc.) being returned.

Despite my best efforts this explanation is likely clear as mud, so let's clarify with an example. Begin by setting the `folders` structure's `map` attribute to somewhere within the directory where you tend to manage your various software projects. For instance, mine is set like this:

```
1 folders:
2   - map: ~/Code/dev.todoparrot.com
3     to: /home/vagrant/Code
```

Next, modify the `sites` structure to look like this:

```

1 sites:
2     - map: dev.todoparrot.com
3       to: /home/vagrant/Code/dev.todoparrot.com/public

```

Save the changes and we'll next create a quick test to confirm you can indeed talk to the Homestead webserver. Create the directory identified by the `map` attribute, and inside it create a directory named `public`. Create a file named `index.php` inside the `public` directory, adding the following contents to it:

```

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

```

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

```

1 $ vagrant up
2 Bringing machine 'default' up with 'virtualbox' provider...
3 ==> default: Importing base box 'laravel/homestead'...
4 ==> default: Matching MAC address for NAT networking...
5 ==> default: Checking if box 'laravel/homestead' is up to date...
6 ==> default: Setting the name of the VM: homestead-7
7 ==> default: Clearing any previously set network interfaces...
8 ==> default: Preparing network interfaces based on configuration...
9 ...
10 ==> default: Forwarding ports...
11     default: 80 => 8000 (adapter 1)
12     default: 443 => 44300 (adapter 1)
13     default: 3306 => 33060 (adapter 1)
14     default: 5432 => 54320 (adapter 1)
15     default: 22 => 2222 (adapter 1)
16 ==> default: Running 'pre-boot' VM customizations...
17 ==> default: Booting VM...
18 $

```

Your Homestead virtual machine is up and running! With that done, we have one remaining step. We'll need to configure your laptop to recognize what it should do when the `dev.todoparrot` URL defined in `Homestead.yaml` is requested in the browser. To do so, you'll need to update your development machine's hosts file so you can easily access the server via a hostname rather than the IP address found in the `Homestead.yaml` file. If you're running OSX or Linux, this file is found at `/etc/hosts`. If you're running Windows, you'll find the file at `C:\Windows\System32\drivers\etc\hosts`. Open up this file and add the following line:

```
1 192.168.10.10 dev.todoparrot.com
```

After saving these changes, we'll want to create the Laravel project that will be served via this URL. However, there still remains plenty to talk about regarding Homestead and virtual machine management so in the sections that follow I discuss several important matters pertaining to this topic. For the moment though I suggest jumping ahead to the section "Creating the TODOParrot Application" and returning to the below sections later.

Managing Your Virtual Machine

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

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

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

```
1 $ vagrant up
```

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

```
1 $ vagrant destroy
```

I stress executing the `destroy` command this *will delete* the virtual machine and all of its data! Executing this command is very different from shutting down the machine using `halt`.

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

```
1 $ vagrant box list
2 laravel/homestead (virtualbox, 0.4.2)
```

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

```
1 $ vagrant --help
```

SSH'ing Into Your Virtual Machine

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

```
1 $ ssh vagrant@127.0.0.1 -p 2222
2 Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 3.19.0-25-generic x86_64)
3
4 * Documentation:  https://help.ubuntu.com/
5 Last login: Thu Mar 10 17:11:55 2016 from 10.0.2.2
```

Windows users will need to first install an SSH client. A popular Windows SSH client is [PuTTY](http://www.putty.org/)²⁷.

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

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

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

Transferring Files Between Homestead and Your Laptop

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

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

²⁷<http://www.putty.org/>

```

1  $ sftp -P 2222 vagrant@127.0.0.1
2  Connected to 127.0.0.1.
3  sftp> cd dev.farm.com
4  sftp> get hello.txt
5  Fetching /home/vagrant/dev.todoparrot.com/db.sql.gz to db.sql.gz
6  /home/vagrant/dev.farm.com/db.sql.gz 0% 0 0.0KB/s --:-- ETA
7  sftp>

```

Connecting to Your Database

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

```

1  databases:
2      - homestead

```

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

```

1  $ ssh vagrant@127.0.0.1 -p 2222
2  Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 3.19.0-25-generic x86_64)
3
4  * Documentation:  https://help.ubuntu.com/
5  Last login: Wed Mar 23 00:56:23 2016 from 10.0.2.2

```

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

```

1  vagrant@homestead:~$ mysql -u homestead homestead -p
2  Enter password:
3  Reading table information for completion of table and column names
4  You can turn off this feature to get a quicker startup with -A
5
6  Welcome to the MySQL monitor.  Commands end with ; or \g.
7  Your MySQL connection id is 2330
8  Server version: 5.7.11 MySQL Community Server (GPL)
9

```

```
10 Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.
11
12 Oracle is a registered trademark of Oracle Corporation and/or its
13 affiliates. Other names may be trademarks of their respective
14 owners.
15
16 Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
17
18 mysql>
```

There are of course no tables in the database (we'll do this in chapter 3), but feel free to have a look anyway:

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

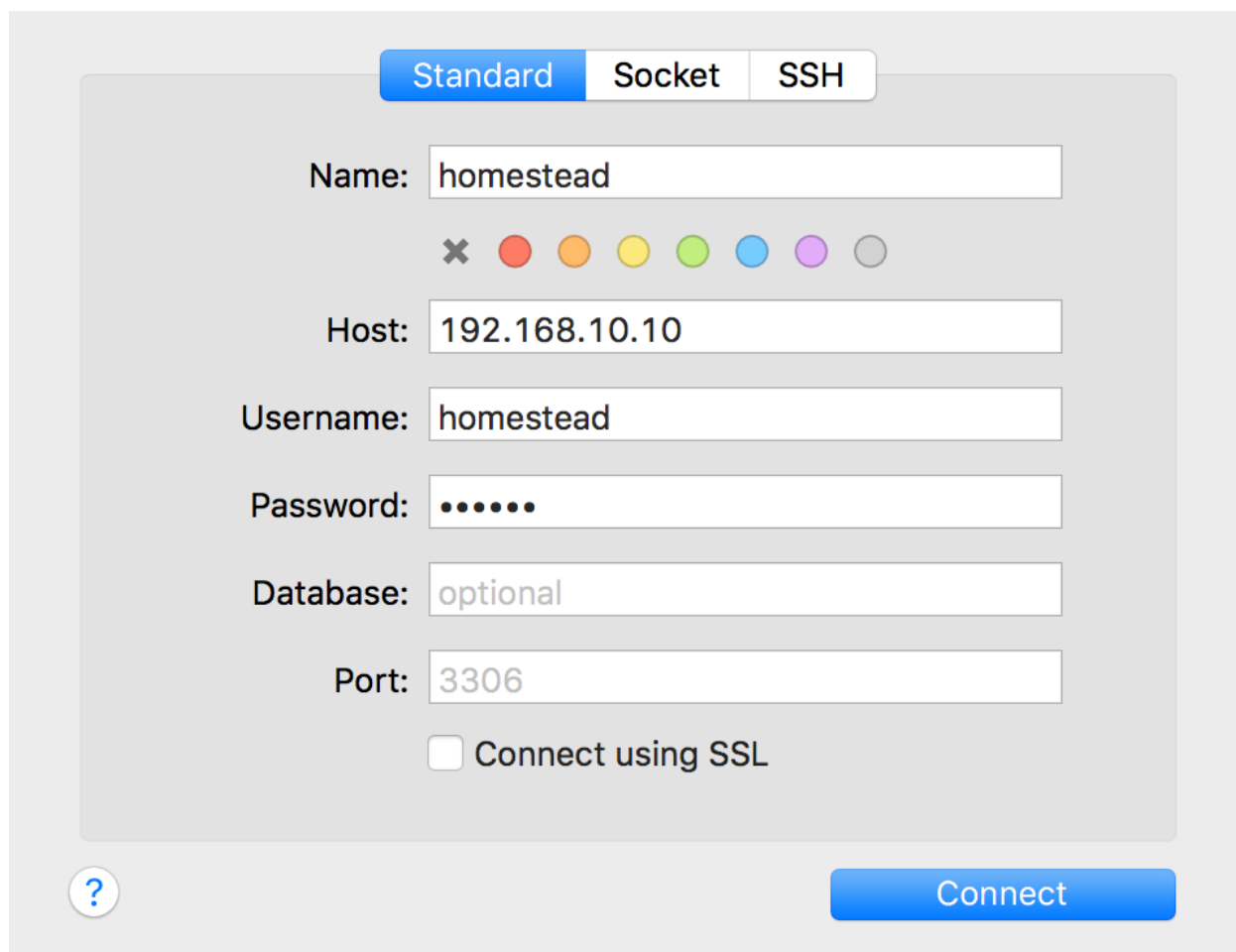
To exit the mysql client, just execute exit:

```
1 mysql> exit;
2 Bye
3 $
```

Chances are you prefer to interact with your database using a GUI-based application such as [Sequel Pro](http://www.sequelpro.com/)²⁸ or [phpMyAdmin](https://www.phpmyadmin.net/)²⁹. You'll connect to the homestead database like you would any other, by supplying the username (homestead), password (secret), and the host, which is 192.168.10.10. For instance, the following screenshot depicts my Sequel Pro connection window:

²⁸<http://www.sequelpro.com/>

²⁹<https://www.phpmyadmin.net/>



The Sequel Pro connection window

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

Defining Multiple Homestead Sites and Databases

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

```

1  folders:
2      - map: ~/Code/dev.larabrain.com
3        to: /home/vagrant/dev.larabrain.com
4      - map: ~/Code/dev.todoparrot.com
5        to: /home/vagrant/dev.todoparrot.com
6
7  sites:
8      - map: dev.larabrain.com
9        to: /home/vagrant/dev.larabrain.com/public
10     - map: dev.todoparrot.com
11       to: /home/vagrant/dev.todoparrot.com/public
12
13 databases:
14     - dev_larabrain_com
15     - dev_todoparrot_com

```

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

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

```

1  $ vagrant global-status
2  id      name      provider  state  directory
3  -----
4  6f13a59  default  virtualbox running /Users/wjgilmore/Homestead

```

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

```

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

```

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

Introducing Valet

Virtual machines and ready-made environments such as Homestead are great, and have become indispensable tools I use on a daily basis. However Homestead can admittedly be rather intimidating for newcomers, and can frankly be overkill for many developers. If you use a Mac and are interested in a no-frills development environment, Laravel offers a streamlined solution called *Valet* which can be configured in mere minutes.

Installing Valet

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

```
1 $ brew install homebrew/php/php70
```

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

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

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

```
1 $ valet install
```

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

```
1 $ brew install mysql
```

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

Serving Sites with Valet

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

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

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

- 1 `$ laravel new todoparrot`

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



The Laravel splash page

It doesn't get any easier than that!

Creating the TODOParrot Application

With Laravel (and presumably Homestead or Valet) 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. Create a new Laravel project using the `laravel` command:

- 1 `$ laravel new todoparrot`

Of course, you can call the project directory anything you want. If you're using Valet, then you're free to place the project directory anywhere you please. However, if you're using Homestead, recall Homestead expects the application to reside in the directory you specified within the `Homestead.yaml` file's `folders` structure's `map` property. As a reminder here is what mine looks like:

```
1 folders:
2   - map: ~/Code/todoparrot
3   - to: /home/vagrant/Code
```

These contents are a combination of files and directories, each of which plays an important role in the functionality of your application so it's important for you to understand their purpose. Let's quickly review the role of each:

- `.env`: Laravel 5 uses the [PHP dotenv](https://github.com/vlucas/phpdotenv)³⁰ library to conveniently manage your application's configuration variables. 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, which should be used as a template from which fellow developers will copy over to `.env` and modify to suit their own needs. I'll talk more about these files and practical approaches for managing your environment settings in the later section, "Configuring Your Laravel Application".
- `.gitattributes`: This file is used by [Git](http://git-scm.com/)³¹ to ensure consistent settings across machines, which is useful when multiple developers using a variety of operating systems are working on the same project. You'll find a few default settings in the file, however these are pretty standard and you in all likelihood won't have to modify them. Plenty of other attributes are however available; Scott Chacon's online book, "[Pro Git](http://git-scm.com/book)"³² includes a section ("[Customizing Git - Git Attributes](http://git-scm.com/book/en/Customizing-Git-Git-Attributes)"³³) with further coverage on this topic.
- `.gitignore`: This file tells Git what files and folders should not be included in the repository. You'll see a few default settings in here, including the `vendor` directory which houses the Laravel source code and other third-party packages, and the `.env` file, which should never be managed in version control since it presumably contains sensitive settings such as database passwords.
- `app`: This directory contains much of the custom code used to power your application, including the models, controllers, and middleware. We'll spend quite a bit of time inside this directory as the book progresses.
- `artisan`: `artisan` is a command-line tool we'll use to rapidly create new parts of your applications such as controllers and models, manage your database's evolution through a great feature known as *migrations*, and interactively debug your application. We'll return to `artisan` repeatedly throughout the book because it is such an integral part of Laravel development.
- `bootstrap`: This directory contains the various files used to initialize a Laravel application, loading the configuration files, various application models and other classes, and define the locations of key directories such as `app` and `public`. Normally you won't have to modify any of the files found in this directory.

³⁰<https://github.com/vlucas/phpdotenv>

³¹<http://git-scm.com/>

³²<http://git-scm.com/book>

³³<http://git-scm.com/book/en/Customizing-Git-Git-Attributes>

- `composer.json`: [Composer](https://getcomposer.org)³⁴ is PHP's de facto package manager, used by thousands of developers around the globe to quickly integrate popular third-party solutions such as [Swift Mailer](http://swiftmailer.org/)³⁵ and [Doctrine](http://www.doctrine-project.org/)³⁶ into a PHP application. Laravel heavily depends upon Composer, and you'll use the `composer.json` file to identify the packages you'll like to integrate into your Laravel application. If you're not familiar with Composer by the time you're done reading this book you'll wonder how you ever lived without it. In fact in this introductory chapter alone we'll use it several times to install various useful packages.
- `composer.lock`: This file contains information about the state of your project's installed Composer packages at the time these packages were last installed and/or updated. Like the `bootstrap` directory, you will rarely if ever directly interact with this file.
- `config`: This directory contains several files used to configure various aspects of your Laravel application, such as the database credentials, the cache, e-mail delivery, and session settings.
- `database`: This directory contains the directories used to house your project's database migrations and seed data (migrations and database seeding are both introduced in Chapter 3).
- `gulpfile.js`: Laravel 5.0 introduced a new feature called *Laravel Elixir*. Elixir relies upon the `Gulpfile.js` file to define various [Gulp.js](http://gulpjs.com/)³⁷ tasks useful for automating 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.
- `phpunit.xml`: Even trivial web applications should be accompanied by an automated test suite. Laravel leaves little room for excuse to avoid this best practice by automatically configuring your application to use the popular [PHPUnit](http://phpunit.de/)³⁸ test framework. The `phpunit.xml` is PHPUnit's application configuration file, defining characteristics such as the location of the application tests. We'll return to the topic of testing repeatedly throughout the book.
- `public`: The `public` directory serves as your application's web root directory, housing the `.htaccess`, `robots.txt`, and `favicon.ico` files, in addition to a file named `index.php` that is the *first* file to execute when a user accesses your application. This file is known as the *front controller*, and it is responsible for loading and executing the application. It's because the `index.php` file serves as the front controller that you needed to identify the `public` directory as your application's root directory when configuring `Homestead.yaml` earlier in this chapter.
- `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](http://github.com/wjgilmore/todoparrot)³⁹ 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 such as CoffeeScript and SaaS files.

³⁴<https://getcomposer.org>

³⁵<http://swiftmailer.org/>

³⁶<http://www.doctrine-project.org/>

³⁷<http://gulpjs.com/>

³⁸<http://phpunit.de/>

³⁹<http://github.com/wjgilmore/todoparrot>

- **routes:** The `routes` directory is new to 5.3. It replaces the old `app/Http/routes.php` file, and separates your application's routing definitions into three separate files: `api.php`, `console.php`, and `web.php`. Collectively, these files determine how your application responds to different endpoints. We'll return to these files repeatedly throughout the book, beginning in Chapter 2.
- **server.php:** The `server.php` file can be used to bootstrap your application for the purposes of serving it via PHP's built-in web server. While a nice feature, Homestead offers a far superior development experience and so you can safely ignore this file and feature.
- **storage:** The `storage` directory contains your project's cache, session, and log data.
- **tests:** The `tests` directory contains your project's PHPUnit tests. Testing is a recurring theme throughout this book, and thanks to Laravel's incredibly simple test integration features I highly encourage you to follow along closely with the examples provided in these sections.
- **vendor:** The `vendor` directory is where the Laravel framework code itself is stored, in addition to any other third-party code. You won't typically directly interact with anything found in this directory, instead doing so through the Composer interface.

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. Presuming you've configured Homestead and generated the Laravel project in the appropriate directory, you should be able to navigate to `http://dev.todoparrot.com` (`http://todoparrot.dev` if you're using Valet) and see the page presented in the below screenshot:



The Laravel splash page

So where does this splash page come from? 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](http://www.php-fig.org/psr/psr-4/)⁴⁰, meaning your project controllers, models, and other key resources are namespaced. The default namespace is set to `App`, which is pretty generic, however if you don't plan on distributing your application to third-parties then the default is going to be just fine. If you did want to update your project's namespace to something unique, such as `Todoparrot`. You can do so using the artisan CLI's `app:name` command:

```
1 $ php artisan app:name Todoparrot
2 Application namespace set!
```

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

Because I'm not distributing `IgniteRental` as software which might be integrated into an application, I've left the default namespace as `App`.

Configuring Your Laravel Application

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

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

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

⁴⁰<http://www.php-fig.org/psr/psr-4/>

- `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. You'll learn all about Laravel's database support in chapters 3 and 4.
- `filesystems.php`: The `filesystems.php` configuration file defines the file system your project will use to manage assets such as file uploads. Thanks to Laravel's integration with [Flysystem](https://github.com/thephpleague/flysystem)⁴¹, support is available for a wide variety of adapters, among them the local disk, Amazon S3, Azure, Dropbox, FTP, Rackspace, and Redis.
- `mail.php`: As you'll learn in Chapter 5 it's pretty easy to send an e-mail from your Laravel application. The `mail.php` configuration file defines various settings used to send those e-mails, including the desired driver (a variety of which are supported, among them Sendmail, SMTP, PHP's `mail()` function, and Mailgun). You can also direct 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](http://kr.github.io/beanstalkd/)⁴² or [Amazon Simple Queue Service](http://aws.amazon.com/sqs/)⁴³. The `queue.php` configuration file defines the desired queue driver and other relevant settings.
- `services.php`: If your application uses a third-party service such as Stripe for payment processing or Mailgun for e-mail delivery you'll use the `services.php` configuration file to define any third-party service-specific settings.
- `session.php`: It's entirely likely your application will use sessions to aid in the management of user preferences and other customized content. Laravel supports a number of different session drivers used to facilitate the management of session data, including the file system, cookies, a database, the Alternative PHP Cache, Memcached, and Redis. You'll use the `session.php` configuration file to identify the desired driver, and manage other aspects of Laravel's session management capabilities.
- `view.php`: The `view.php` configuration file defines the default location of your project's view files and the renderer used for pagination.

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

⁴¹<https://github.com/thephpleague/flysystem>

⁴²<http://kr.github.io/beanstalkd/>

⁴³<http://aws.amazon.com/sqs/>

Configuring Your Environment

Your application will likely require access to database credentials and other sensitive information such as API keys for accessing third party services. This confidential information should never be shared with others, and therefore you'll want to take care it isn't embedded directly into the code. Instead, you'll want to manage this data within *environment variables*, and then refer to these variables within the application.

Laravel supports a very convenient solution for managing and retrieving these variables thanks to integration with the popular [PHP dotenv](https://github.com/vlucas/phpdotenv)⁴⁴ package. When developing your application you'll define environment variables within the `.env` file found in your project's root directory. The default `.env` file looks like this:

```
1 APP_ENV=local
2 APP_KEY=base64:Xq9+pNBKPC1IskLbT7M3Y08kzQ=
3 APP_DEBUG=true
4 APP_LOG_LEVEL=debug
5 APP_URL=http://localhost
6
7 DB_CONNECTION=mysql
8 DB_HOST=127.0.0.1
9 DB_PORT=3306
10 DB_DATABASE=homestead
11 DB_USERNAME=homestead
12 DB_PASSWORD=secret
13
14 BROADCAST_DRIVER=log
15 CACHE_DRIVER=file
16 SESSION_DRIVER=file
17 QUEUE_DRIVER=sync
18
19 REDIS_HOST=127.0.0.1
20 REDIS_PASSWORD=null
21 REDIS_PORT=6379
22
23 MAIL_DRIVER=smtp
24 MAIL_HOST=mailtrap.io
25 MAIL_PORT=2525
26 MAIL_USERNAME=null
27 MAIL_PASSWORD=null
28 MAIL_ENCRYPTION=null
29
```

⁴⁴<https://github.com/vlucas/phpdotenv>

```
30 PUSHER_APP_ID=  
31 PUSHER_KEY=  
32 PUSHER_SECRET=
```

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

```
1 'mysql' => [  
2     'driver' => 'mysql',  
3     'host' => env('DB_HOST', 'localhost'),  
4     'port' => env('DB_PORT', '3306'),  
5     'database' => env('DB_DATABASE', 'forge'),  
6     'username' => env('DB_USERNAME', 'forge'),  
7     'password' => env('DB_PASSWORD', ''),  
8     'charset' => 'utf8',  
9     'collation' => 'utf8_unicode_ci',  
10    'prefix' => '',  
11    'strict' => true,  
12    'engine' => null,  
13 ],
```

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

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

Useful Development and Debugging Tools

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



The debugging and development utilities discussed in this section are specific to Laravel, and do not take into account the many other tools available to PHP in general. Be sure to check out [Xdebug](http://xdebug.org/)⁴⁵, [FirePHP](http://www.firephp.org/)⁴⁶, and the many tools integrated into PHP IDEs such as [Zend Studio](http://www.zend.com/en/products/studio)⁴⁷ and [PHPStorm](http://www.jetbrains.com/phpstorm/)⁴⁸.

The dd() Function

Ensuring the `.env` file's `APP_DEBUG` variable is set to `true` is the easiest way to view information about any application errors, because Laravel will dump error- and exception-related information directly to the browser. However, 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()`⁴⁹ helper function, which will dump a variable's contents to the browser and halt further script execution. For example suppose you defined an array inside a Laravel application and wanted to output its contents to the browser. Here's an example array:

```
1 $items = [  
2     'items' => [  
3         'Pack luggage',  
4         'Go to airport',  
5         'Arrive in San Juan'  
6     ]  
7 ];
```

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

```
1 dd($items);
```

Passing `$items` into `dd()` will cause the array contents to be dumped to the browser window as depicted in the below screenshot.

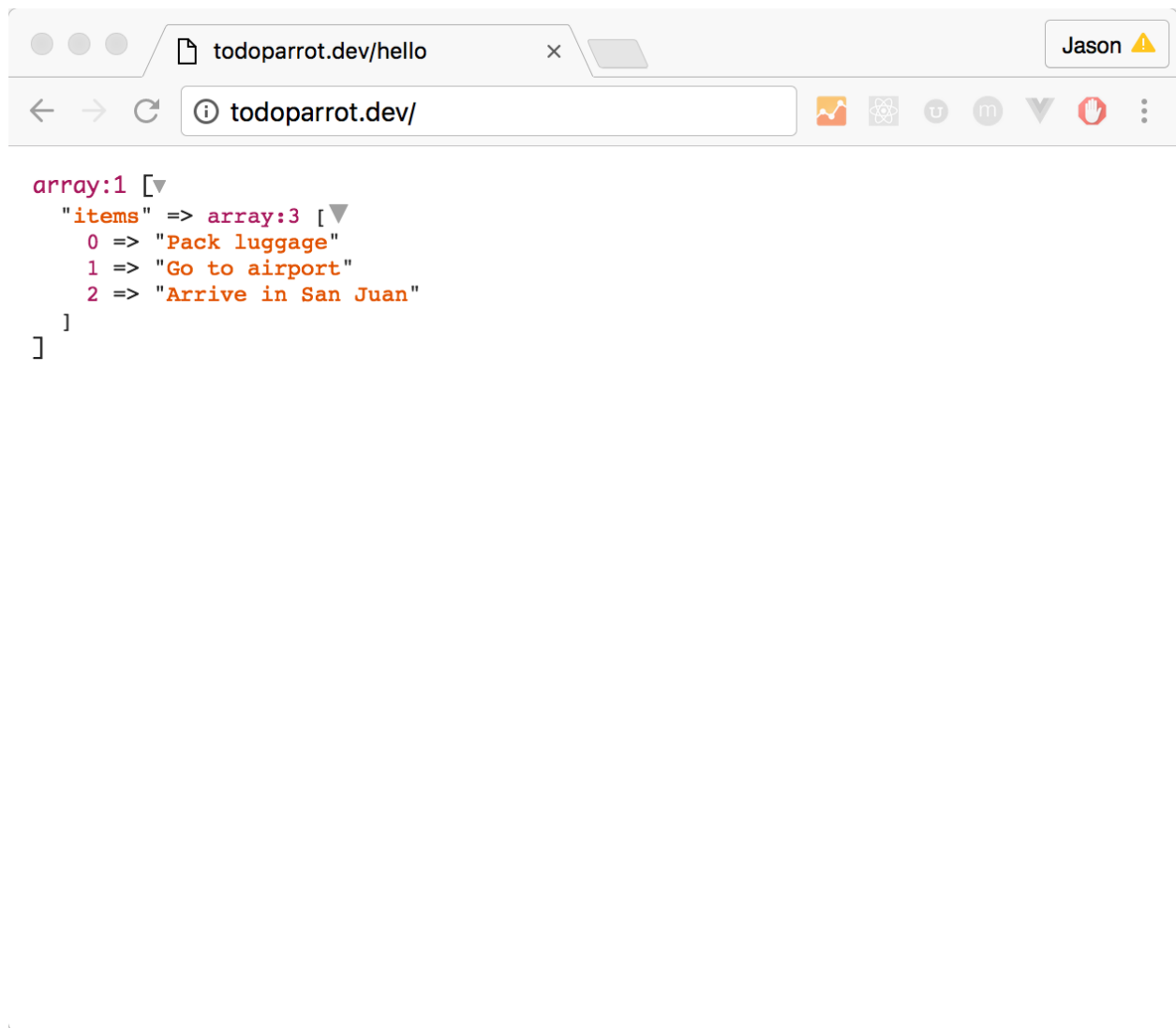
⁴⁵<http://xdebug.org/>

⁴⁶<http://www.firephp.org/>

⁴⁷<http://www.zend.com/en/products/studio>

⁴⁸<http://www.jetbrains.com/phpstorm/>

⁴⁹<https://laravel.com/docs/master/helpers#method-dd>



dd() function output



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

The Laravel Logger

While the `dd()` helper function is useful for quick evaluation of a variable's contents, taking advantage of Laravel's logging facilities is a more effective approach if you plan on repeatedly monitoring one or several data structures or events without interrupting script execution. Laravel will by default log error-related messages to the application log, located at `storage/logs/laravel.log`.

Because Laravel's logging features are managed by [Monolog](#)⁵⁰, 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](#)⁵¹ via [FirePHP](#)⁵², to the [Chrome console](#)⁵³ using [Chrome Logger](#)⁵⁴, or even trigger alerts via e-mail, [HipChat](#)⁵⁵ or [Slack](#)⁵⁶. Further, if you're using the Laravel Debugbar (introduced later in this chapter) you can easily peruse these messages from the Debugbar's Messages tab.

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

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

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

```
1 [2016-09-21 09:14:29] local.DEBUG: array (  
2     'items' =>  
3     array (  
4         0 => 'Pack luggage',  
5         1 => 'Go to airport',  
6         2 => 'Arrive in San Juan',  
7     ),  
8 )
```

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:

⁵⁰<https://github.com/Seldaek/monolog>

⁵¹<https://getfirebug.com/>

⁵²<http://www.firephp.org/>

⁵³<https://developer.chrome.com/devtools/docs/console>

⁵⁴<http://craig.is/writing/chrome-logger>

⁵⁵<http://hipchat.com/>

⁵⁶<https://www.slack.com/>

```
1 \Log::info('Just an informational message.');
```

```
2 \Log::warning('Something may be going wrong.');
```

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

```
4 \Log::critical('Danger, Will Robinson! Danger!');
```

Integrating the Logger and FirePHP

When monitoring the log file it's common practice to use the `tail -f` command (available on Linux and OS X) to view any log file changes in real time. You can however avoid the additional step of maintaining an additional terminal window for such purposes by instead sending the log messages to the [Firebug](#)⁵⁷ console, allowing you to see the log messages alongside your application's browser output. You'll do this by integrating [FirePHP](#)⁵⁸.

You'll first need to install the Firebug and [FirePHP](#)⁵⁹ extensions. If you're running Firefox, both are available via Mozilla's official add-ons site. If you're running another browser such as Chrome, you can install [Firebug Lite](#)⁶⁰ and FirePHP4Chrome. After restarting your browser, you can begin sending log messages directly to the browser console by adding the following to `bootstrap/app.php`:

```
1 $app->configureMonologUsing(function($monolog) {
```

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

```
3 });
```

After saving the changes, you can log for instance the `$items` array just as you did previously:

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

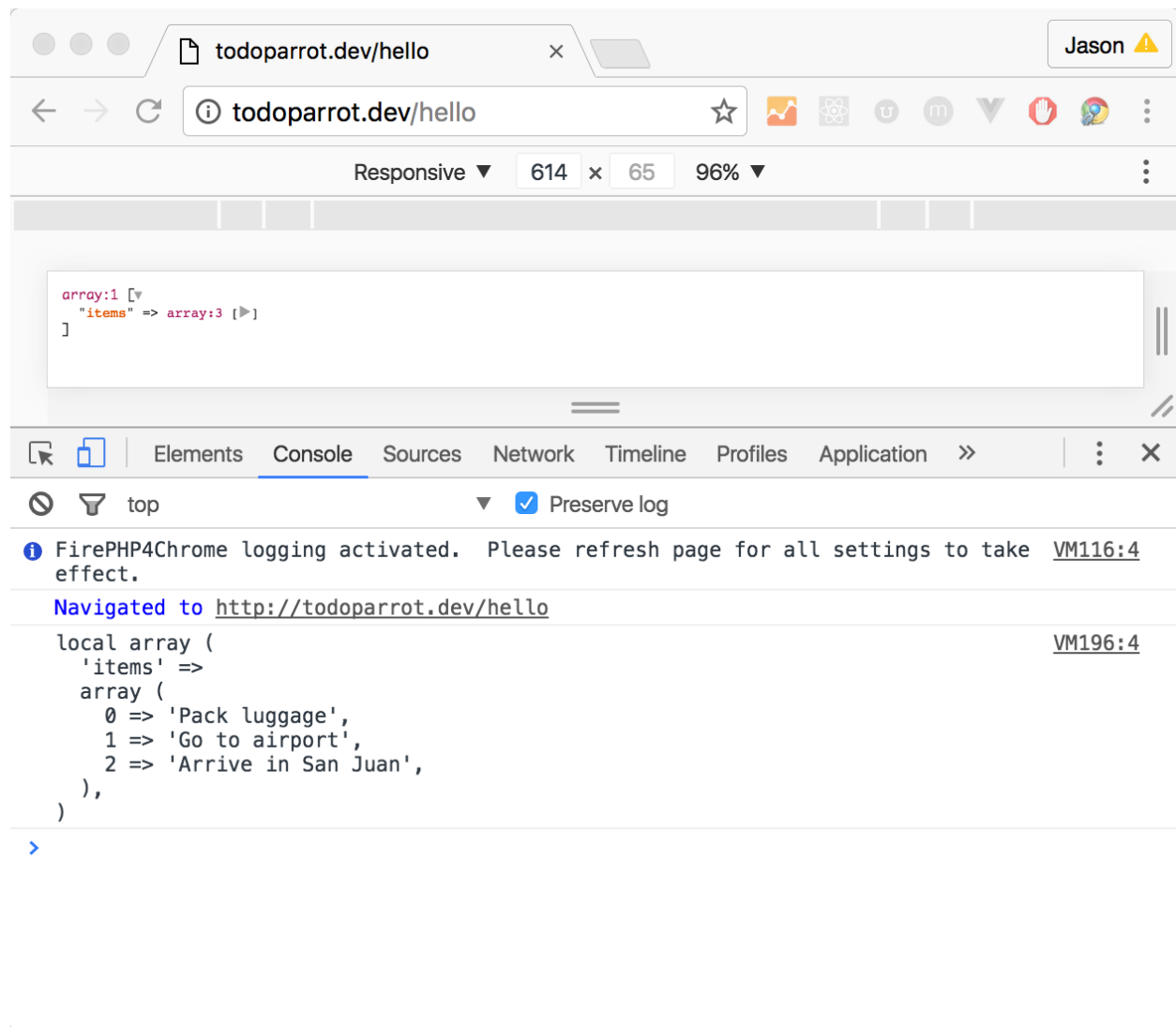
Once executed, the `$items` array will appear in your browser console as depicted in the below screenshot.

⁵⁷<https://getfirebug.com/>

⁵⁸<http://www.firephp.org/>

⁵⁹<https://addons.mozilla.org/en-US/firefox/addon/firephp/>

⁶⁰<https://getfirebug.com/firebuglite>



Logging to the Chrome console via Firebug Lite and 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:


```
1 $ php artisan tinker
2 Psy Shell v0.7.2 (PHP 7.0.6 cli) by Justin Hileman
3 >>>
```

Tinker uses [PsySH](http://psysh.org/)⁶¹, 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](http://psysh.org/)⁶² to learn more about what this great utility can do. In the meantime, let's get used to the interface:

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

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

```
1 >>> sort($items);
2 => true
3 >>> $items;
4 => [
5     "Arrive in San Juan",
6     "Go to airport",
7     "Pack luggage"
8 ]
9 >>>
```

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

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

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

⁶¹<http://psysh.org/>

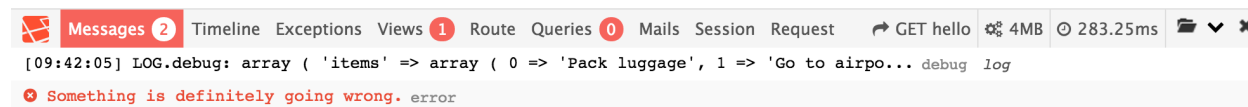
⁶²<http://psysh.org/>

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](#)⁶³ that provides easy access to the status of these events and much more by straddling the bottom of your browser window (see below screenshot).



TODOParrot



error debug

The Laravel Debugbar

The Debugbar is visually similar to [Firebug](#)⁶⁴, consisting of multiple tabs that when clicked result in context-related information in a panel situated below the menu. These tabs include:

- **Messages:** Use this tab to view log messages directed to the Debugbar. I'll show you how to do this in a moment.
- **Timeline:** This tab presents a summary of the time required to load the page.
- **Exceptions:** This tab displays any exceptions thrown while processing the current request.
- **Views:** This tab provides information about the various views used to render the page, including the layout.
- **Route:** This tab presents information about the requested route, including the corresponding controller and action.

⁶³<https://github.com/barryvdh/laravel-debugbar>

⁶⁴<http://getfirebug.com>

- **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.
- **Session:** This table presents any session-related information made available while processing the request.
- **Request:** This tab lists information pertinent to the request, including the status code, request headers, response headers, and session attributes.

To install the Laravel Debugbar, execute the following command:

```
1 $ composer require barryvdh/laravel-debugbar --dev
2 Using version ^2.2 for barryvdh/laravel-debugbar
3 ./composer.json has been updated
4 Loading composer repositories with package information
5 ...
6 $
```

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

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

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

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

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

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

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

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

Save the changes and reload the home page within the browser. Check the Debugbar's Messages panel and you'll see the logged message! Like the Laravel logger, the Laravel Debugbar supports the log levels defined in [PSR-3⁶⁵](#), 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 PHPUnit support with every new Laravel project. PHPUnit is a very popular *unit testing framework* which allows you to create and execute well-organized tests used to confirm all parts of your application are working as expected.

Each new Laravel application even includes an example test which you can use as a reference for beginning to write your own tests! You'll find this test inside the tests directory. It's named `ExampleTest.php`, and it demonstrates how to write a test that accesses the project home page, and determines whether the text `Laravel 5` is visible:

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

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

⁶⁵<http://www.php-fig.org/psr/psr-3/>

```
1 $ phpunit
2 PHPUnit 5.5.4 by Sebastian Bergmann and contributors.
3
4 .                                     1 / 1 (100%)
5
6 Time: 262 ms, Memory: 10.00MB
7
8 OK (1 test, 2 assertions)
```

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

```
1 ->see('PHP');
```

Replace the string `Laravel` with anything you please, such as `PHP`. If you reload the browser after saving the changes you'll see the updated text. Now run `execute phpunit` anew:

```
1 $ phpunit
2 PHPUnit 5.5.4 by Sebastian Bergmann and contributors.
3
4 F                                     1 / 1 (100%)
5
6 Time: 262 ms, Memory: 10.00MB
7
8 There was 1 failure:
9
10 1) ExampleTest::testBasicExample
11 <head>
12 <meta charset="utf-8">
13 <meta http-equiv="X-UA-Compatible" content="IE=edge">
14 ...
15 Failed asserting that the page contains the HTML [PHP].
16 Please check the content above.
17
18 ...
19
20 FAILURES!
21 Tests: 1, Assertions: 2, 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 [Laravel](http://laravel.com/docs/master/testing)⁶⁶ 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 application. Onwards!

⁶⁶<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 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 the user's request, processing these requests through a *controller* and *action*. This chapter offers a wide-ranging introduction to all of these concepts. You'll also learn how to use Laravel Elixir to automate otherwise tedious tasks such as CSS and JavaScript compilation. 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 example project and viewed the default landing page within the browser. The page was pretty sparse, consisting of the text "Laravel 5". If you view the page's source from within the browser, you'll see a few CSS styles are defined, a Google font reference, and some simple HTML. You'll find this view in the file `welcome.blade.php`, found in the directory `resources/views`. Open this file in your PHP editor, and update it to look like this:

```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Welcome to TODOParrot</title>
6   </head>
7   <body>
8     <h1>Welcome to TODOParrot</h1>
9   </body>
10 </html>
```

Reload the application's home page within the browser, and you should see the header `Welcome to TODOParrot`. Congratulations! You've just created your first Laravel view.

So why is this particular view returned when you navigate to the home page? The `welcome.blade.php` view is served by a *route definition* found in the `routes/web.php` file:

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

This code block tells Laravel to serve the `welcome.blade.php` file when a GET request is made to the application's homepage, represented by the forward slash (/). Although the majority of your views will be served in conjunction with a controller (more about this in a bit), if some particular page contains purely static content (such as an “About Us” page) then the above approach is a perfectly acceptable solution.

Because Laravel presumes all views will use the `blade.php` extension (you'll learn more about the meaning of `blade` later in this chapter), Laravel saves you the hassle of referencing the extension. Of course, you're free to name the view prefix whatever you please; for instance try renaming `welcome.blade.php` to `hola.blade.php`, and then update the view function to look like this:

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

Reload the browser and you'll see the same outcome as before, despite the filename change.

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

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

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

Creating Your First Controller

The lone default route serves the important purpose of giving you *something* to see when accessing the home page of a newly generated Laravel application, however in practice you'll rarely serve

views directly through the `routes.php` file. This is because the majority of your views will contain some degree of dynamic data, and this dynamic data is typically retrieved and passed into a view by way of a *controller*.

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

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

When generating controllers with `make:controller`, Laravel will by default create an empty class devoid of any properties or methods. The controller files will be placed in `app/Http/Controllers`, and assigned whatever name you supplied as an argument (in the case of the above example, the name will be `WelcomeController.php`). The generated class file looks like this:

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

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

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



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

This example presumes you'd like to adhere to the best practice discussed earlier pertaining to organizing views within subdirectories. To do so, create a new directory inside `resources/views`

named `welcome`, and inside it create a file named `index.blade.php`. Open `index.blade.php` and for the moment just add a line of text such as `Welcome to TODOParrot`.

After saving these changes open the `web.php` file (`routes/web.php`). Replace the lone defined route with the following route:

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

This route tells Laravel to respond to GET requests made to the application home page (`/`) by executing the `WelcomeController`'s `index` action. Save these changes, return to the browser, and reload the home page. You should see the text you added to the newly created `index.blade.php` file!

Managing Your Application Routes

As of Laravel 5.3, the files found in the `routes` directory are tasked with managing your application routes. Traditionally speaking, “application” means “web application” or more familiarly “website”, however modern applications often consist of more than URLs accessible via a web browser. For instance, in addition to a public-facing website, your Laravel project might also offer an API. Further, you might create a console-based interface which might be used to conveniently carry out various administrative tasks such as backing up the database or assembling and displaying useful statistics. This is why in that `routes` directory you’ll find an `api.php`, `console.php`, and `web.php` files; each is tasked with managing routes associated with these three areas: API, console, and web.

So why separate these three sets of routes in the first place? Notably because web-related routes will likely be used in conjunction with features such as cookie-based authentication and user sessions, something not needed when building an API. Also, juggling both API- and web-related routes in the same routes file (which was the case prior to 5.3) can get downright annoying, with some applications sporting route numbers climbing into the hundreds.

In the previous section you created a new route in the `web.php` configuration file, and accessed that route via the browser. Indeed we’ll spend much of the rest of this book focused on web-related routing, however at least some of what you’ll learn in this regards will be relevant for API development. In Chapter 9 you’ll learn how to use the `console.php` file to create console-based endpoints for managing different aspects of your application. With that said, the rest of this section covers material that is equally important to web and API developers alike, so pay careful attention!

Defining RESTful controllers

These days it’s commonplace to dedicate each controller to managing an associated resource. For instance a controller named `ListsController` might be responsible for retrieving all lists, retrieving a list detail view, inserting a new list, modifying an existing list, and deleting a list. These types

of controllers are often created in such a way so as to conform to [REST⁶⁷](#) conventions. Laravel supports RESTful controllers, and although I won't formally introduce the concept until Chapter 3 I nonetheless thought this a suitable spot to at least show you how to generate a RESTful controller skeleton and define the associated routes within the `routes.php` file. To generate a RESTful controller, pass the `--resource` flag to the `make:controller` command:

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

If you have a look at the newly created controller (located in `app/Http/Controllers/ListsController.php`), you'll see the class contains seven different methods, each representing one of the seven default RESTful actions (all of which are introduced in the next chapter if you're not familiar with this topic). With the methods defined, you'll next want to register the associated endpoints within the `web.php` file. You can do so using the `Route::resource()` method:

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

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

Method	URI	Name
GET	/lists	lists.index
GET	/lists/new	lists.create
POST	/lists	lists.store
GET	/lists/{id}	lists.show
GET	/lists/{id}/edit	lists.edit
PUT	/lists/{id}	lists.update
DELETE	/lists/{id}	lists.destroy

Of course, each corresponding action found in the `Lists` controller will need to be implemented, but thanks to the `Route::resource` shortcut at least we don't have to worry about defining each route manually. In the next chapter you'll learn how to start implementing these actions.

Defining Route Parameters

If you look closely at the above table you'll see some route URIs include the string `:id`. These are placeholders for variables. For instance if you wanted to retrieve a list associated with the ID 42, that variable would be passed along to the `Lists` controller's `show` action within the URI like so: `/lists/42`.

⁶⁷http://en.wikipedia.org/wiki/Representational_state_transfer

Of course, as the application grows in complexity you'll probably want to augment your RESTful controllers with custom capabilities falling outside the scope of the seven standard routes. In doing so, these custom routes will probably require dynamic parameters. For instance, suppose you wanted to offer users an alternative view of their various TODO lists in which the lists are organized by category. The URL for the housework-related lists might look like this:

```
1 http://todoparrot.com/lists/category/home
```

Because the category is variable according to which category the user would like to display, your custom route definition will look like this:

```
1 Route::get('lists/category/{category}', 'ListsController@category');
```

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

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

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.

Incidentally, when augmenting RESTful controllers, always remember you'll want to place custom routes *above* the `Route::resource` definition. So for instance to augment the `Lists` controller with the category-related route, your route definitions for this controller will be defined in the following order:

```
1 Route::get('lists/category/{category}', 'ListsController@category');
2 Route::resource('lists', 'ListsController');
```

Multiple Route Parameters

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

```

1 Route::get(
2     'lists/category/{category}/{subcategory}',
3     'ListsController@category'
4 );

```

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

```

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

```

Optional Route Parameters

All of the route parameters we've defined thus far are *required*. Neglecting to include them in the URL will result in an error. Sometimes however you might prefer to define these **parameters as optional**, and assign default values should the optional parameter(s) not be provided. To define an optional parameter you'll append a question mark onto the parameter name, like this:

```

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

```

You can then identify the parameter is optional in the associated action by defining it as optional in the method declaration and then checking and responding accordingly to the value in the method body:

```

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

```



Some readers might wonder what that cryptic `?:` syntax is all about. This is a shortcut to PHP's ternary operator. The shortcut was introduced in PHP 5.3, and just saves a bit of typing.

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 `lists.category`:

```

1 Route::get('lists/category/{category}',
2     [
3         'as' => 'lists.category',
4         'uses' => 'ListsController@category'
5     ]
6 );

```

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

```

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

```

Or more preferably, if you take advantage of the powerful Laravel Collective package (introduced later in the chapter), you can use `link_to_route` to programmatically generate the entire URL:

```

1 {!! link_to_route('blog.category', 'PHP', ['category' => 'php']) !!}

```

Once rendered to the browser, both of the above examples will produce the following HTML:

```

1 <a href="/blog/category/php">PHP</a>

```

While it may seem odd to programmatically generate URLs, route aliases are very useful particularly in the early stages of development when your URL structure is still rather fluid. For instance, I use route aliases and `link_to_route()` almost exclusively on all client projects since the client may wish to change the URI structure at some later point. Doing so is easy since it only involves updating the `routes/web.php` file.

Listing Routes

As your application progresses it can be easy to forget details about the various routes. You can view a list of all available routes using the `route:list` command. The output of this command has grown to the point where displaying an example here is impossible due to space restrictions, so instead I invite you to execute the command and I'll explain the meaning of each column:

- **Domain:** It's possible to define sub-domain routes in your route files. If a sub-domain is associated with a given route, its name will be displayed here.
- **Method:** The HTTP method used to contact the endpoint, for example GET, POST, or DELETE.
- **URI:** The string used to represent a network resource. For instance, in previous examples we've been referring to the URI `blog/category/{category}`.

- **Name:** The name of the route alias. When working with RESTful controllers Laravel will provide defaults. For instance, the `Lists` controller's `index` action's default route alias is `lists.index`. You are free to override these defaults, and if you've assigned an alias to a custom route you'll find it listed here.
- **Action:** The controller action associated with the route. For instance, the `Lists` controller's `index` action will be identified here as `ListsController@index`.
- **Middleware:** Laravel supports filtering route requests using middleware. If a route is affected by one or more middleware, those middleware names will be listed here. You'll learn all about route middleware in Chapter 6.

Caching Routes

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

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

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

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

This command will delete the cached routes file, causing Laravel to return to parsing the route definitions found in the `routes` directory until you again decide to cache the routes.



Executing the `route:cache` command on a newly generated Laravel project will confusingly fail because example routes found in both `routes/api.php` and `routes/web.php` use closures. Laravel's caching mechanism doesn't support caching closure-based routes and so if you'd like to see this command in action you'll need to first comment out or remove those closure-based routes.

Introducing the Blade Template Engine

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

In order for Laravel to recognize a Blade-augmented view, you'll need to use the `.blade.php` extension. In this section we'll work through a number of different examples involving Blade syntax and the `welcome.blade.php` view.

Displaying Variables

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

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

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

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

The first line presents an example of a Blade comment. Blade comments are enclosed within the `{{--` and `--}}` tags, and will be removed from the rendered web page.

The second line references the variable name. Variables are referenced in Laravel views in the same fashion as you would see in any PHP script. However, the variable is additionally enclosed within curly brackets so Laravel knows what parts of the page reference variables.

After saving the changes, reload the home page and you should see “San Juan Vacation” embedded into the view!

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

```
1 public function index()  
2 {  
3     $name = 'San Juan Vacation';  
4     return view('welcome')  
5         ->withName($name);  
6 }
```

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

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

Incidentally, although I always prefer to do so for readability’s sake, you’re not required to include a space between the curly brackets and variable. Laravel will parse the following identically to the above:

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

Displaying Multiple Variables

You’ll certainly want to pass multiple variables into a view. You can do so as demonstrated earlier using the `with` method, but passing in an array:

```
1 public function index()  
2 {  
3  
4     $data = [  
5         'name' => 'San Juan',  
6         'date' => date('Y-m-d')  
7     ];  
8  
9     return view('welcome')->with($data);  
10  
11 }
```

To display the `$name` and `$date` variables within the view, just update your view to reference both:

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

You could also use multiple with methods, like so:

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

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

```
1 $name = 'San Juan Vacation';
2 $date = date('Y-m-d');
3 return view('welcome', compact('name', 'date'));
```

The `$name` and `$date` variables defined in your action will then automatically be made available to the view. If this is confusing see the PHP manual's `compact()` function documentation at <http://php.net/compact>⁶⁸.

Determining Variable Existence

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

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

Escaping Dangerous Input

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

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

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

⁶⁸<http://php.net/compact>

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

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

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

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

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

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

Looping Over an Array

TODOParrot users spend a lot of time working with lists, such as the list of tasks, 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 summary of Blade's looping capabilities). Let's demonstrate each by iterating over an array into the index view. Begin by modifying the WelcomeController.php index method to look like this:

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

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

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

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

Because the array could be empty, consider using the @forelse construct instead to display an appropriate message should no array items exist:

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

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

If Conditional

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

```
1 @if (count($lists) > 0)
2   <ul>
3     @forelse ($lists as $list)
4       <li>{{ $list }}</li>
5     @empty
6       <li>You don't have any lists saved.</li>
7     @endforelse
8   </ul>
9 @else
10  <p>
11    You haven't created any lists!
12  </p>
13 @endif
```

Blade also supports the `if-elseif-else` construct:

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

Managing Your Application Layout

The typical web application consists of a design elements such as a header and footer, and these elements are generally found on every page. Because eliminating redundancy is one of Laravel's central tenets, clearly you won't want to repeatedly embed elements such as the site logo and navigation bar within every view. Instead, you'll use Blade 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 `app.blade.php`. Add the following contents to this newly created file:

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Welcome to TODOParrot</title>
6 </head>
7 <body>
8
9     @yield('content')
10
11 </body>
12 </html>
```

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

```
1 @extends('layouts.app')
2
3 @section('content')
4
5 <h1>Welcome to TODOParrot</h1>
6
7 <p>
8     TODOParrot is the ultimate productivity application sporting
9     a soothing tropical theme.
10 </p>
11
12 @endsection
```

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

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

Defining Multiple Layout Sections

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

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

```
13     </p>
14     @show
15 </div>
16 </div>
17 </body>
```

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

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

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

```
1 @extends('layouts.app')
2
3 @section('content')
4     <h1>Welcome to TODOParrot!</h1>
5 @endsection
6
7 @section('advertisement')
8     @parent
9     <p>
10         Buy the TODOParrot Productivity guide for $10!
11     </p>
12 @endsection
```

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

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

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

Taking Advantage of View Partial

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

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

Notice how I'm using a view variable (`$link`) in the partial. When importing the partial into your view you can optionally pass a variable into the view like so:

```
1 <table class="table borderless">
2   @foreach ($links as $link)
3
4     @include('partials.row', ['link' => $link])
5
6   @endforeach
7 </table>
```

Integrating Images, CSS and JavaScript

Your project images, CSS and JavaScript should be placed in the project's `public` directory. While you could throw everything into `public`, for organizational reasons I prefer to create `images`, `css`, and `javascript` directories. Regardless of where in the `public` directory you place these files, you're free to reference them using standard HTML or can optionally take advantage of a few helpers available via the [Laravel Collective HTML package](https://github.com/LaravelCollective/html)⁶⁹. For instance, the following two statements are identical:

⁶⁹<https://github.com/LaravelCollective/html>


```

1 <!-- Standard HTML markup -->
2 
3
4 <!-- Using the LaravelCollective/html package -->
5 {!! HTML::image('images/logo.png', 'TODOParrot logo') !!}

```

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

```

1 <!-- Standard HTML markup -->
2 <link rel="stylesheet" href="/css/app.min.css">
3 <script src="/javascript/jquery-1.10.1.min.js"></script>
4 <script src="/javascript/bootstrap.min.js"></script>
5
6 <!-- Using the LaravelCollective/html package -->
7 {!! HTML::style('css/app.min.css') !!}
8 {!! HTML::script('javascript/jquery-1.10.1.min.js') !!}
9 {!! HTML::script('javascript/bootstrap.min.js') !!}

```

If you want to take advantage of these HTML helpers (and I highly recommend you do), you'll need to install the `LaravelCollective/HTML` package. This was previously part of the native Laravel distribution, but has been moved into a separate package as of version 5. Fortunately, installing the package is easy. First, use Composer to install the package:

```

1 $ composer require laravelcollective/html

```

This installed the package and additionally added the following line to your `composer.json` file:

```

1 "require": {
2     ...
3     "laravelcollective/html": "^5.2"
4 },

```

Next, add the following line to the `providers` array found in your `config/app.php` file:

```

1 Collective\Html\HtmlServiceProvider::class,

```

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

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

With these changes in place, you can begin using the `LaravelCollective/HTML` package. Be sure to check out [the GitHub README⁷⁰](#) of the Laravel documentation for a list of available helpers. I strongly suggest taking the time to do so, because I find `LaravelCollective/HTML` to be perhaps more invaluable than any other package, particularly so for designing web forms, a topic we'll discuss in great detail in chapter 5.

Introducing Elixir

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

The Laravel developers hope to reduce some of the time and hassle associated with these sort of tasks by providing a new API called [Laravel Elixir⁷¹](#). Elixir supports Gulp, providing an easy solution for integrating your Laravel project's with technologies such as Less, Sass, and Webpack. In this section you'll learn how to create and execute Elixir tasks in order to more effectively manage your project. 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

[Gulp⁷²](#) 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 plugins⁷³](#). In this section I'll show you how to install and configure Gulp for subsequent use within Elixir.

Installing Gulp

Gulp is built atop [Node.js⁷⁴](#), meaning you'll need to install Node.js. No matter your operating system this is easily done by downloading one of the installers via [the Node.js website⁷⁵](#). If you'd prefer to

⁷⁰<https://github.com/LaravelCollective/html>

⁷¹<https://github.com/laravel/elixir>

⁷²<http://gulpjs.com/>

⁷³<http://gulpjs.com/plugins/>

⁷⁴<http://nodejs.org>

⁷⁵<http://nodejs.org/download/>

build Node from source you can download the source code via this link. Mac users can install Node via Homebrew, while Linux users almost certainly can install Node via their distribution's package manager.

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

```
1 $ node -v
2 v6.4.0
```

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

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

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

```
1 $ gulp -v
2 [22:02:43] CLI version 3.9.0
```

With Gulp installed it's time to install Elixir!

Installing Elixir

Laravel 5 applications automatically include a file named `package.json` which resides in the project's root directory. As of version 5.3 the file looks like this:

```
1 {
2   "private": true,
3   "scripts": {
4     "prod": "gulp --production",
5     "dev": "gulp watch"
6   },
7   "devDependencies": {
8     "bootstrap-sass": "^3.3.7",
9     "gulp": "^3.9.1",
10    "jquery": "^3.1.0",
11    "laravel-elixir": "^6.0.0-9",
12    "laravel-elixir-vue": "^0.1.4",
13    "laravel-elixir-webpack-official": "^1.0.2",
14    "lodash": "^4.14.0",
```

```
15     "vue": "^1.0.26",
16     "vue-resource": "^0.9.3"
17   }
18 }
```

Node's `npm` package manager uses `package.json` to learn about and install a project's Node module dependencies. Several Node modules are defined by default, including notable solutions such as `gulp`, `jquery`, and `bootstrap-sass`. You can install these packages locally using the package manager like so:

```
1 $ npm install
```

Once complete, you'll find a new directory named `node_modules` has been created within your project's root directory, and within in it you'll find the package directories. This directory is used by `Node.js` to house the various packages installed per the `package.json` specifications, so you definitely do not want to delete nor modify it.

Running Your First Elixir Task

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

```
1 elixir(mix => {
2     mix.sass('app.scss')
3     .webpack('app.js');
4 });
```

This task actually chains together two separate tasks; The first, `mix.sass`, is used to compile `Sass` (<http://sass-lang.com/>) files into CSS which can then be imported into your project layout. The second, `webpack`, uses the popular `Webpack JavaScript bundler` (<https://webpack.github.io/>) to compile ECMAScript 2015 (also referred to as ES6) code into browser-supported JavaScript. This latter capability is particularly exciting for Laravel developers because it opens up a whole new world of possibility in terms of adding powerful JavaScript-based features to your applications. I'll return to the `Webpack` task later in this chapter so for now let's focus on `mix.sass`.

The `mix.sass` task compiles a file named `app.scss` which resides in `resources/assets/sass`. The default `app.scss` file contains the following CSS import statements:

```
1 // Fonts
2 @import url(https://fonts.googleapis.com/css?family=Raleway:300,400,600);
3
4 // Variables
5 @import "variables";
6
7 // Bootstrap
8 @import "node_modules/bootstrap-sass/assets/stylesheets/bootstrap";
```

When the above Gulp task is executed (I'll show you how in a moment), these import statements will result in the Google-hosted Raleway font, the `variables.scss` file (also found in `resources/assets/sass`), and the Bootstrap framework CSS being integrated into a single CSS file and saved to `/public/css/app.css`. You can then reference this compiled CSS file within your project layout. Further, because `app.scss` is Sass-based, you can take advantage of its powerful CSS extensions to more effectively manage your CSS declarations. For instance if you open `variables.scss` you'll see several CSS variables (that's right, CSS *variables*) are defined:

```
1 // Body
2 $body-bg: #f5f8fa;
3
4 // Borders
5 $laravel-border-color: darken($body-bg, 10%);
6 $list-group-border: $laravel-border-color;
7 $navbar-default-border: $laravel-border-color;
8 $panel-default-border: $laravel-border-color;
9 $panel-inner-border: $laravel-border-color;
10
11 <snip>
12
13 // Inputs
14 $input-border: lighten($text-color, 40%);
15 $input-border-focus: lighten($brand-primary, 25%);
16 $input-color-placeholder: lighten($text-color, 30%);
17
18 // Panels
19 $panel-default-heading-bg: #fff;
```

Because the imported Bootstrap CSS is Sass-enabled, these variables will be used within the Bootstrap stylesheet and the outcome compiled to `app.css`. You can peruse a large list of available variables within the bootstrap-sass GitHub repository (https://github.com/twbs/bootstrap-sass/blob/master/assets/stylesheets/bootstrap/_variables.scss).

The #f5f8fa color assigned to `$body-bg` is fairly light and difficult to distinguish from white, so let's change it to something a bit more obvious like #ACD1E9. After saving the changes execute the Elixir `mix.sass` task by running `gulp` within your project root directory:

```
1 $ gulp
2 [00:02:17] Using gulpfile ~/Code/dev.propertybot.com/gulpfile.js
3 [00:02:17] Starting 'all'...
4 [00:02:17] Starting 'sass'...
5 [00:02:18] Finished 'sass' after 1.04 s
6 [00:02:18] Starting 'webpack'...
7 [00:02:22]
8 [00:02:22] Finished 'webpack' after 4.04 s
9 [00:02:22] Finished 'all' after 5.09 s
10 [00:02:22] Starting 'default'...
11 [removing cool tabular output due to space reasons]
12 [00:02:22] Finished 'default' after 6.69 ms
```

Once complete, you'll find a compiled CSS file named `app.css` inside your project's `public/css` directory. Of course, in order to actually use the styles defined in the `app.css` file you'll need to reference it within the layout file (`resources/views/layouts/app.blade.php`):

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

Or if you're using the Laravel Collective HTML package, you can use the `HTML::style` method:

```
1 {!! HTML::style('css/app.css') !!}
```

Return to the browser and reload the home page. You'll see a light blue background has been applied.



A blue background

Be sure to check out the [Sass guide](http://sass-lang.com/guide)⁷⁶ for a comprehensive summary of what's possible with this powerful CSS extension syntax.

Watching for Changes

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

⁷⁶<http://sass-lang.com/guide>

```

1  $ gulp watch
2  [22:21:07] Using gulpfile ~/Code/dev.test.com/gulpfile.js
3  [22:21:07] Starting 'watch'...
4  [22:21:07] Finished 'watch' after 19 ms
5  [22:21:40] Starting 'sass'...
6
7  Fetching Sass Source Files...
8    - resources/assets/sass/app.scss
9
10 Saving To...
11   - public/css/app.css
12
13 [22:21:41] Finished 'sass' after 1.1 s
14 [22:21:41] gulp-notify: [Laravel Elixir] Sass Compiled!

```

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

Integrating the Bootstrap Framework

[Bootstrap](#)⁷⁷ is a blessing to design-challenged developers such as yours truly, offering an impressively comprehensive and eye-appealing set of widgets and functionality. Being a particularly incapable designer, I use Bootstrap as the starting point for all of my personal projects, often customizing it with a [Bootswatch theme](#)⁷⁸.

When you executed `npm install` in the last section, the `bootstrap-sass` package was installed and with it the Bootstrap source files in Sass format. When you ran `gulp`, the following line in `resources/assets/sass/app.scss` makes the Bootstrap code available to your project:

```

1  @import "node_modules/bootstrap-sass/assets/stylesheets/bootstrap";

```

Once added, provided you referenced the compiled CSS file (`/css/app.css`) in your project layout you're free to begin taking advantage of Bootstrap's various CSS widgets. For instance try adding a stylized hyperlink to the `welcome.blade.php` view just to confirm everything is working as expected:

```

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

```

⁷⁷<http://getbootstrap.com/>

⁷⁸<http://bootswatch.com/>



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

Keep in mind you aren't required to use Bootstrap via the `bootstrap-sass` module. You could also alternatively (and possibly preferably) use Bootstrap's recommended CDNs (Content Delivery Network) to add Bootstrap and jQuery (jQuery is required to take advantage of the Bootstrap JavaScript plugins:

```
1 <head>
2   ...
3   <link rel="stylesheet"
4     href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
5   <script
6     src="http://code.jquery.com/jquery-3.1.1.min.js"></script>
7   <script
8     src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js">
9   </script>
10 </head>
```

Integrating the Bootstrapper Package

Using Bootstrap as described above is perfectly fine, and in fact I do exactly that in TODOParrot. However, for those of you who prefer to use view helpers whenever possible, check out [Bootstrapper](#)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:

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

To install Bootstrapper, just execute the following Composer command:

```
1 $ composer require patricktallmadge/bootstrapper
```

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

⁷⁹<https://github.com/patricktallmadge/bootstrapper>

⁸⁰<https://github.com/patricktallmadge>


```

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

```

By registering the Bootstrapper service provider, Laravel will know 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 `config/app.php`. Paste the following rather lengthy bit of text into the bottom of the array:

```

1  'Accordion' => 'Bootstrapper\Facades\Accordion',
2  'Alert' => 'Bootstrapper\Facades\Alert',
3  'Badge' => 'Bootstrapper\Facades\Badge',
4  'Breadcrumb' => 'Bootstrapper\Facades\Breadcrumb',
5  'Button' => 'Bootstrapper\Facades\Button',
6  'ButtonGroup' => 'Bootstrapper\Facades\ButtonGroup',
7  'Carousel' => 'Bootstrapper\Facades\Carousel',
8  'ControlGroup' => 'Bootstrapper\Facades\ControlGroup',
9  'DropdownButton' => 'Bootstrapper\Facades\DropdownButton',
10 'Form' => 'Bootstrapper\Facades\Form',
11 'Helpers' => 'Bootstrapper\Facades\Helpers',
12 'Icon' => 'Bootstrapper\Facades\Icon',
13 'InputGroup' => 'Bootstrapper\Facades\InputGroup',
14 'Image' => 'Bootstrapper\Facades\Image',
15 'Label' => 'Bootstrapper\Facades\Label',
16 'MediaObject' => 'Bootstrapper\Facades\MediaObject',
17 'Modal' => 'Bootstrapper\Facades\Modal',
18 'Navbar' => 'Bootstrapper\Facades\Navbar',
19 'Navigation' => 'Bootstrapper\Facades\Navigation',
20 'Panel' => 'Bootstrapper\Facades\Panel',
21 'ProgressBar' => 'Bootstrapper\Facades\ProgressBar',
22 'Tabbable' => 'Bootstrapper\Facades\Tabbable',
23 'Table' => 'Bootstrapper\Facades\Table',
24 'Thumbnail' => 'Bootstrapper\Facades\Thumbnail',

```

With Bootstrapper installed, you're free to begin taking advantage of the numerous shortcuts it has to offer! See the [documentation](http://bootstrapper.eu1.frbit.net/)⁸¹ for all of the latest details.

Testing Your Views

Laravel's test-friendliness was dramatically in the 5.1 release, using a syntax that practically reads like English prose. For instance, let's create a new file named `WelcomeTest.php`:

⁸¹<http://bootstrapper.eu1.frbit.net/>

```
1 $ php artisan make:test WelcomeTest
```

This will create a new file named `WelcomeTest.php`, found in the `tests` directory:

```
1 <?php
2
3 use Illuminate\Foundation\Testing\WithoutMiddleware;
4 use Illuminate\Foundation\Testing\DatabaseMigrations;
5 use Illuminate\Foundation\Testing\DatabaseTransactions;
6
7 class WelcomeTest extends TestCase
8 {
9     /**
10      * A basic test example.
11      *
12      * @return void
13      */
14     public function testExample()
15     {
16         $this->assertTrue(true);
17     }
18 }
```

Add the following method to the `WelcomeTest` class:

```
1 public function testCanSeeTheContactUsLink()
2 {
3     $this->visit('/')
4     ->see('Contact Us');
5 }
```

As you can see, the meaning behind the syntax is so obvious that it hardly bears explanation. The test is going to visit the application home URI and confirm the string `Contact Us` is found somewhere in the rendered page. Confirm it works by opening the `welcome/index.blade.php` view and adding the following link:

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

Save the changes and run the test like so:

```
1 $ phpunit tests/WelcomeTest.php
2 PHPUnit 5.5.4 by Sebastian Bergmann and contributors.
3
4 ..
5
6 Time: 346 ms, Memory: 14.25Mb
7
8 OK (2 tests, 2 assertions)
```

You can make tests like this much more granular, actually simulating *clicking on the link* and confirming the user will be taken to the desired location:

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

At this point the test will fail because we haven't yet created a route handler for the URI `about/contact`. Stay tuned because in forthcoming chapters we'll continue to expand upon test-related matters, writing automated tests to test forms and user authentication are all implemented to the project specifications.

Conclusion

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

Chapter 3. Introducing Laravel Models

This chapter is the first of two devoted to the data-related aspects of your Laravel application. In this chapter 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 look at Laravel's *Query Builder* interface (including a demonstration of how to execute raw SQL) and a section on testing your models.

This introductory material sets the stage for the next chapter, in which you'll learn about more advanced topics such as model relationships, collections, and scopes.

Configuring Your Project Database

In Chapter 1 I briefly touched upon Laravel's database support. To summarize, Laravel supports four databases, including MySQL, PostgreSQL, SQLite, and Microsoft SQL Server. The `config/database.php` file tells Laravel which database you'd like your application to use, what authentication credentials to use to connect to this database, as well as defines a few other data-related settings. You won't however typically modify the contents of this file directly; instead you'll define the desired database type, name, and connection credentials using environment variables (notably the `.env` file during development). Even so, it's beneficial to know what's going on in this file, and so I've pasted in the contents of `config/database.php` below (with comments removed). Following the snippet I'll summarize the purpose of each setting:

```
1 <?php
2
3 return [
4
5     'fetch' => PDO::FETCH_OBJ,
6
7     'default' => env('DB_CONNECTION', 'mysql'),
8
9     'connections' => [
10
```

```
11     'sqlite' => [  
12         'driver' => 'sqlite',  
13         'database' => env('DB_DATABASE', database_path('database.sqlite')),  
14         'prefix' => '',  
15     ],  
16  
17     'mysql' => [  
18         'driver' => 'mysql',  
19         'host' => env('DB_HOST', 'localhost'),  
20         'port' => env('DB_PORT', '3306'),  
21         'database' => env('DB_DATABASE', 'forge'),  
22         'username' => env('DB_USERNAME', 'forge'),  
23         'password' => env('DB_PASSWORD', ''),  
24         'charset' => 'utf8',  
25         'collation' => 'utf8_unicode_ci',  
26         'prefix' => '',  
27         'strict' => false,  
28         'engine' => null,  
29     ],  
30  
31     'pgsql' => [  
32         'driver' => 'pgsql',  
33         'host' => env('DB_HOST', 'localhost'),  
34         'port' => env('DB_PORT', '5432'),  
35         'database' => env('DB_DATABASE', 'forge'),  
36         'username' => env('DB_USERNAME', 'forge'),  
37         'password' => env('DB_PASSWORD', ''),  
38         'charset' => 'utf8',  
39         'prefix' => '',  
40         'schema' => 'public',  
41         'sslmode' => 'prefer',  
42     ],  
43  
44 ],  
45  
46 'migrations' => 'migrations',  
47  
48 'redis' => [  
49  
50     'cluster' => false,  
51  
52     'default' => [  

```

```

53         'host' => env('REDIS_HOST', 'localhost'),
54         'password' => env('REDIS_PASSWORD', null),
55         'port' => env('REDIS_PORT', 6379),
56         'database' => 0,
57     ],
58
59 ],
60
61 ];

```

Let's review each setting:

- **fetch:** The Eloquent ORM will by default return database results as instances of anonymous PHP objects. 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 strongly 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'll set this using the `DB_CONNECTION` environment variable, assigning the variable a value of `mysql` (the default), `pgsql` (PostgreSQL), `sqlite` (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 a remote database.
- **connections:** This array defines the database authentication credentials for each supported database. Keep in mind Laravel will only consider the database associated with the `default` setting value (although it is possible to configure Laravel to use multiple databases), therefore you can leave the unused database options untouched (or entirely remove them). Several additional environment variables will then be used to supply the database's connection credentials. For instance, MySQL users will define the `DB_HOST`, `DB_PORT`, `DB_DATABASE`, `DB_USERNAME`, and `DB_PASSWORD`, or assume the default settings (`localhost`, `3306`, `forge`, `forge`, and no password, respectively).
- **migrations:** This setting defines the name of the table used for managing your project's migration status (more about this later in the chapter). 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](http://redis.io/)⁸² 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 has already been created for you by way of the `Homestead.yaml` file, negating the need to go through these additional steps. With the database defined, it's time to begin interacting with it!

⁸²<http://redis.io/>

Introducing the Eloquent ORM

Object-relational mappers (ORM) are 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, [impedance mismatch](http://en.wikipedia.org/wiki/Impedance_matching)⁸³ is the term used to describe the challenges associated with using an object-oriented language such as PHP or Ruby in conjunction with a relational database, because the programmer is faced with the task of somehow mapping the application objects to database tables. An ORM solves this problem by providing a convenient interface for converting application objects to database table records, and vice versa. Additionally, most ORMs offer a vast array of convenient features useful for querying, inserting, updating, and deleting records, managing table relationships, and dealing with other aspects of the data life cycle.

Creating Your First Model

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

```
1 $ php artisan make:model ToDoList -m
2 Model created successfully.
3 Created Migration: 2016_10_04_011840_create_todo_lists_table
```

Notice I also supplied the `-m` option to the `make:model` command. This option tells Laravel to additionally create a companion migration (we'll talk about migrations in the next section). You'll find the new model in `app/ToDoList.php`. It looks like this:

```
1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class ToDoList extends Model
8 {
9     //
10 }
```

⁸³http://en.wikipedia.org/wiki/Impedance_matching

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. Although the class is empty, we'll begin expanding its contents soon enough.



Like most web frameworks, Laravel expects the model name to be singular form (`TodoList`), and the underlying table names to be plural form (`todo_lists`). Further, when a model uses camel casing (e.g. `ListItem`, `AppointmentReminder`), the corresponding table will use underscores to separate each capitalized word (e.g. `list_items`, `appointment_reminders`).

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

Introducing Migrations

With the model created, you'll typically create the corresponding database table, done through a fantastic Laravel feature known as *migrations*. Migrations offer a file-based approach to changing the structure of your database, allowing you to create and drop tables, add, update and delete columns, and add indexes, among other tasks. Further, you can easily revert, or *roll back*, any changes if a mistake has been made or you otherwise reconsider the decision. Finally, because each migration is stored in a text file, you can manage them within your project repository.

To demonstrate the power of migrations, let's have a look at the migration file that was created along with the `TodoList` model (presuming you supplied the `-m` option). This migration file is named `2016_10_04_011840_create_todo_lists_table`, and it was placed in the `database/migrations` directory. Open up this file and you'll see the following contents:

```
1  <?php
2
3  use Illuminate\Support\Facades\Schema;
4  use Illuminate\Database\Schema\Blueprint;
5  use Illuminate\Database\Migrations\Migration;
6
7  class CreateTodoListsTable extends Migration
8  {
9
10     public function up()
11     {
12         Schema::create('todo_lists', function (Blueprint $table) {
```



```

13         $table->increments('id');
14         $table->timestamps();
15     });
16 }
17
18 public function down()
19 {
20     Schema::dropIfExists('todo_lists');
21 }
22 }

```

Like a model, a Laravel migration is just a standard PHP class, except in this case the class extends the Migration class. The `up()` and `down()` methods 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 migration's `up()` method:

```

1 public function up()
2 {
3     Schema::create('todo_lists', 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 `todo_lists`. An anonymous function (closure) passed along as the `Schema::create` method's second parameter defines the table columns:

- `$table->increments('id')`: The `increments` method indicates we want to create an automatically incrementing integer column that will additionally serve as the table's primary key. This is pretty standard procedure when creating relational database tables, so you're generally not going to remove nor modify this field.
- `$table->timestamps()`: The `timestamps` method tells 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. Whether you choose to omit these fields is entirely up to you, although I generally leave them in place since they can often serve as a useful debugging mechanism.

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 [documentation](#)⁸⁴ for a complete list, otherwise stay tuned as we'll cover various other methods in the examples to come.

We want each `todo_lists` 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 include these fields:

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

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

- `$table->string('name')`: The `string` method indicates we want to create a variable character column (commonly referred to as a VARCHAR by databases such as MySQL and PostgreSQL) named `name`. Remember, the `Schema` class is database-agnostic, and therefore leaves it to whatever supported Laravel database you happen to be using to determine the maximum string length, unless you use other means to constrain the limit.
- `$table->text('description')`: The `text` method indicates we want to create a text column (commonly referred to as TEXT by databases such as MySQL and PostgreSQL). I've additionally defined it to be `nullable` since users might not always desire to write list descriptions.

The example's `down()` method is much easier to understand because it consists of a single instruction: `Schema::dropIfExists('todo_lists')`. When executed it will remove, or *drop* the `lists` table.

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

```
1 $ php artisan migrate
2 Migrated: 2016_10_04_011840_create_todo_lists_table
```

Because this is the very first migration, Laravel will also create several other tables, including `migrations`, `password_resets`, and `users`. The `password_resets` and `users` tables are used to manage user-related data, and we'll talk about both in Chapter 7. The `migrations` table is used to keep track of the current migration version. This version number correlates with the name of the migration file. For instance after running the `2016_10_04_011840_create_todo_lists_table` migration the `migrations` table looks like this:

⁸⁴<https://laravel.com/docs/master/migrations#columns>

```

1  mysql> select * from migrations;
2  +-----+-----+
3  | migration                                | batch |
4  +-----+-----+
5  | 2014_10_12_000000_create_users_table    |      1 |
6  | 2014_10_12_100000_create_password_resets_table |      1 |
7  | 2016_10_04_011840_create_todo_lists_table |      1 |
8  +-----+-----+

```

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 (although you'll soon learn this default behavior can be overridden and specific migrations can be rolled back).



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

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

```

1  mysql> show tables;
2  +-----+
3  | Tables_in_dev_todoparrot_com |
4  +-----+
5  | todo_lists                    |
6  | migrations                    |
7  | password_resets               |
8  | users                        |
9  +-----+
10 4 rows in set (0.00 sec)

```

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

```

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

```

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

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

```

1  $ php artisan migrate:rollback --step=1
2  Rolled back: 2016_10_04_011840_create_todo_lists_table

```

Notice I'm using `--step=1` here because we only want to remove the most recently created migration. Omitting this option will result in *all* migrations executed in the most recent batch, which in most cases isn't going to be desired.

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

Dealing with Unsupported Data Types

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

```

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

```

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

Defining Column Modifiers

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

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

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

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

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

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

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

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

You can review a complete list of supported column types and other options in [the Laravel documentation](http://laravel.com/docs/master/migrations)⁸⁵.

⁸⁵<http://laravel.com/docs/master/migrations>

Adding and Removing Columns

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

```
1 $ php artisan make:migration add_done_to_todo_lists_table --table=todo_lists
2 Created Migration: 2016_10_04_0211533_add_done_to_todo_lists_table
```

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

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

Finally, run the migration to modify the `todo_lists` table:

```
1 $ php artisan migrate
2 Migrated: 2016_10_04_0211533_add_done_to_todo_lists_table
```

Controlling Column Ordering

One of the beautiful aspects of an ORM such as Eloquent and migrations is the ability to essentially forget about matters such as column ordering. Even so, my penchant for obsessing over details often leads to the desire to manage column ordering so as to ensure the primary and foreign keys are placed at the beginning of the table, and timestamps at the end of the table. If you would like to insert a column into a specific location within the table structure, use the `after` method:

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

Other Useful Migration Commands

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 database at some point during development:

```
1 $ php artisan migrate:status
2 +-----+-----+
3 | Ran? | Migration |
4 +-----+-----+
5 | Y    | 2014_10_12_000000_create_users_table |
6 | Y    | 2014_10_12_100000_create_password_resets_table |
7 | Y    | 2016_10_04_011840_create_todo_lists_table |
8 +-----+-----+
```

I also regularly use the `migrate:reset` and `migrate:refresh` commands to completely rollback and completely rebuild the migrations, respectively. The `migrate:reset` command is useful when you want to undo *all* migrations, since `migrate:rollback` will only undo one batch/step at a time. The `migrate:refresh` command is useful particularly when you make some adjustments to various schemas and would like to quickly tear down and rebuild all of the migrations.

Creating a RESTful Controller

We'll logically want to interact with the model within the TODOParrot application in order to notably retrieve lists, learn more about a specific list, create a new list, update a list, and delete a list. These tasks are so central to web applications that most popular web frameworks, Laravel included, implement *representational state transfer* (REST), an approach to designing networked applications that codify the way in which these tasks (create, retrieve, update, and delete) are implemented. RESTful applications use the HTTP protocol and a series of well-defined URL endpoints to implement the seven actions defined in the following table (first presented in chapter 2 but recreated here for your convenience):

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

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

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

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

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

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use Illuminate\Http\Request;
6
7 use App\Http\Requests;
8
9 class ListsController extends Controller
10 {
11
12     public function index()
13     {
14         //
15     }
16 }
```



```
17     public function create()  
18     {  
19         //  
20     }  
21  
22     public function store(Request $request)  
23     {  
24         //  
25     }  
26  
27     public function show($id)  
28     {  
29         //  
30     }  
31  
32     public function edit($id)  
33     {  
34         //  
35     }  
36  
37     public function update(Request $request, $id)  
38     {  
39         //  
40     }  
41  
42     public function destroy($id)  
43     {  
44         //  
45     }  
46  
47 }
```

Like a model, a controller is just a typical PHP class that extends a specific Laravel class (the `Controller` class) which gives it special characteristics. Laravel-generated RESTful controllers will by default contain seven public methods (referred to as *actions* in framework parlance), with each intended to correspond with an endpoint defined in the earlier table. However, Laravel doesn't know you intend to use the newly generated controller in a RESTful fashion until the route definitions are defined.

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

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

If you try to access `/lists` you'll be greeted with a blank page. This is because the `Lists` controller's `index` action does not yet identify a view to be rendered! So let's create that view. Begin by creating a new directory named `lists` inside `resources/views`. This is where all of the views associated with `Lists` controller will be housed. Next, create a file named `index.blade.php`, placing it inside this directory. Add the following contents to it:

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

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

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

Save the file and navigate to `/lists` within the browser. 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 `Lists` controller.

Interacting with the Todolist Model

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

```
1 $ php artisan tinker
2 Psy Shell v0.7.2 (PHP 7.0.6 cli) by Justin Hileman
3 >>>
```

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

```
1 >>> namespace App;
2 => null
3 >>> $list = new Todolist;
4 => App\Todolist {#711}
```

Next, assign a list name and description:

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

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

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

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

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

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

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

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

```
1 >>> Todolist::count();
2 => 1
```

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

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

```

1 mysql> select * from todo_lists;
2 +----+-----+-----+-----+-----+
3 | id | name   | notes | created_at           | updated_at           |
4 +----+-----+-----+-----+-----+
5 |  1 | San Juan | ...   | 2016-10-04 21:58:38 | 2016-10-04 21:58:38 |
6 +----+-----+-----+-----+-----+
7 1 row in set (0.00 sec)

```

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

```

1 >>> exit;

```

Integrating a Model Into Your Controller

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

```

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

```

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

```

1 <?php
2
3 namespace App\Http\Controllers;
4
5 use Illuminate\Http\Request;
6
7 use App\Http\Requests;
8
9 use App\Todolist;
10
11 class ListsController extends Controller {

```

```
12
13 ...
14
15 }
```

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

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

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

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

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

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

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

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:

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

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

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

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

```
1 >>> $lists = TodoList::all();
2 >>> $lists->each(function($list) { printf("%s\n", $list->name); });
```

You'll likely rarely want to use the `all()` method unless the target model is associated with a trivial (perhaps a few hundred) 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".

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 = TodoList::find(1);
2 => App\TodoList {#682
3     id: 1,
4     name: "San Juan Vacation",
5     created_at: "2016-03-29 21:58:38",
6     updated_at: "2016-03-29 21:58:38",
7     note: "Vacation planning",
8 }
```

The returned `$list` object properties can then be easily accessed as needed:

```
1 >>> $list->note;
2 => "Vacation planning"
```

Implementing the RESTful Show Action

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

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

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

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

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

```

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

```

After saving the changes, navigate to the Lists controller’s show URI (append an appropriate ID to /lists/ such as /lists/1) and you should see output that looks something like this (HTML tags included for clarity):

```

1  <h1>San Juan Vacation</h1>
2  <p>
3  Created on: 2017-10-04 21:58:38<br />
4  Last modified: 2017-10-04 21:58:38
5  </p>
6
7  <p>
8  Vacation planning
9  </p>

```

Brilliant! We’re now able to view more information about a specific list.

Gracefully Handling Requests for Nonexistent Records

The code presented in the previous example works great if the supplied ID does indeed match a record in the `todo_lists` table, but what happens if the user attempts to access a record that doesn’t exist, such as /lists/23245? The `find` method would return a `null` value, meaning any attempts to retrieve a property in the view would result in the error “Trying to get property of non-object”. Chances are you’ll want to avoid the possibility a non-object is ever passed into the view in the first place. You can do so using the `findOrFail` method instead:

If the desired record is not found, an exception of type `ModelNotFoundException` will be thrown. One easy way to handle this exception is by catching it directly within the action:


```
1 use Illuminate\Database\Eloquent\ModelNotFoundException;
2
3 ...
4
5 public function show($id)
6 {
7
8     try {
9
10         $list = TodoList::findOrFail($id);
11         return view('lists.show')->with('list', $list);
12
13     } catch(ModelNotFoundException $e) {
14
15         return redirect()->route('lists.index');
16
17     }
18
19 }
```

After modifying the action, test the changes out by accessing some record you know to not exist, such as `/lists/42`. The non-existent ID will be passed to `findOrFail`, presumably not be found, and redirect the user to the `Lists` controller.



Of course, it would be useful to provide the user with an error message indicating why the redirection occurred. In Chapter 5 I'll show you how!

If you'd prefer to configure your application to automatically respond to any `ModelNotFoundException` exceptions with a standard 404 page, modify the `app/Exceptions/Handler.php`'s `render()` method to look like this:

```
1 use Illuminate\Database\Eloquent\ModelNotFoundException;
2
3 ...
4
5 public function render($request, Exception $e)
6 {
7
8     if ($exception instanceof ModelNotFoundException)
9     {
10         abort(404);
11     }
```

```
12
13     return parent::render($request, $e);
14
15 }
```

That `abort()` function will result in a view named `404.blade.php` being rendered. This view doesn't yet exist, but as you know by now creating one is easy enough. Just make sure you place it in `resources/views/errors`, because this is Laravel's default location for storing these error-oriented views.

Selecting Specific Columns

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

```
1 $ php artisan tinker
2 Psy Shell v0.7.2 (PHP 7.0.6 â€” cli) by Justin Hileman
3 >>> namespace App;
4 => null
5 >>> $list = TodoList::select('name', 'note')->first();
6 >>> echo $list->name;
7 San Juan Vacation
8 >>> echo $lists;
9 {"name":"San Juan Vacation","note":"Vacation planning"}
```

Counting Records

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

```
1 >>> TodoList::count();
2 50
```

You can also use the `count()` method within your view in conjunction with a collection to determine how many records have been selected:

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

Ordering Records

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

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



You'll use the `get` method to retrieve records when using methods other than `all` or `find`, unless you're solely interested in the first item in that collection, in which case you can use `first`.

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

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

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

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

This is equivalent to executing the following SQL statement:

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

Using Conditional Clauses

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

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

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

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

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

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

You can use `where` in conjunction with `!=` to retrieve records *not equal* to a given value:

```
1 $incompleteLists = TodoList::where('complete', '!=', 1)->get();
```

Still other variations of the `where` method exist. For instance, you can use `whereBetween()` to retrieve records having a column value falling within a range:

```
1 $teenUsers = User::whereBetween('age', [13, 19])->get();
```

Similarly, `whereNotBetween()` can be used to filter a range out of the results:

```
1 $adultUsers = User::whereNotBetween('age', [13, 19])->get();
```

Incidentally, it's perfectly acceptable to chain `where*()` methods together to produce a compound conditional clause. The following example will retrieve a list of adult users who live in Ohio:

```
1 $adultUsers = User::whereNotBetween('age', [13, 19])
2   ->where('state', 'Ohio')->get();
```

This will produce the following SQL:

```
1 select * from users where `age` between 13 and 19 and `state` = 'Ohio'
```

Grouping Records

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

```

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

```

This query can be reproduced in Laravel like so:

```

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

```

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

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

```

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

```

You'll often want to group records in conjunction with a filter. For instance, what if you only wanted to retrieve a grouped count of lists created after 2015? You could use `groupBy` in conjunction with `where`:

```
1 use DB;
2
3 $lists = TodoList::select(
4     DB::raw('year(created_at) as year'),
5     DB::raw('count(name) as count'))
6     ->groupBy('year')
7     ->where('year', '>', '2016')->get();
```

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

```
1 $lists = TodoList::select(
2     DB::raw('year(created_at) as year'),
3     DB::raw('count(name) as count'))
4     ->groupBy('year')
5     ->having('year', '>', '2016')->get();
```

Limiting Returned Records

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

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

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

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

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

```
1 SELECT * from todo_lists 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 present a widget highlighting the most recently created list:

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

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

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

Retrieving a Random Record

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

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

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

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

This is equivalent to executing the following SQL:

```
1 select * from `todo_lists` 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 having the name San Juan Vacation exists use the following statement:

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

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

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

Paginating Results

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

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

```
1 select id, name from todo_lists order by name asc limit 10 offset 0  
2 select id, name from todo_lists order by name asc limit 10 offset 10
```

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

```
1 select id, name from todo_lists order by name asc limit 10
```

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

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


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

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

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

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



Laravel's pagination ribbon

If presenting a list of numbered links to each set of paginated results isn't important, and a simple set of left and right arrows will suffice, use the `simplePaginate()` method instead as you'll benefit from a slight performance improvement:

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

Inserting New Records

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

```
1 $list = new TodoList;
2 $list->name = 'San Juan Vacation';
3 $list->note = 'Vacation planning';
4 $list->save();
```

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

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

```
1 $list = TodoList::create(  
2     [  
3         'name' => 'San Juan Vacation',  
4         'note' => 'Vacation planning'  
5     ]  
6 );
```

However the `create()` method won't work without some additional configuration, because allowing for a record to be created by mass-assigning values via an array could present a security risk. For instance, an improperly configured web form handler could allow an attacker to inject additional fields into a form and send them along with the expected fields. We'll talk more about such security hazards in chapter 5, so for the moment just understand that if you did want to allow mass-assignment by way of the `create()` method, you'll need to identify the fields which are allowed to be assigned in this fashion via the associated model's `$fillable` property, as demonstrated here:

```
1 class TodoList extends Model  
2 {  
3  
4     protected $fillable = ['name', 'note'];  
5  
6 }
```

If the model's underlying table consisted of an unwieldy number of fields and you knew which ones were *not* allowed to be mass-assigned, you could instead use the `$guarded` property:

```
1 class User extends Model  
2 {  
3  
4     protected $guarded = ['is_admin'];  
5  
6 }
```

This means that all fields in the `users` table are mass-assignable *except* for `is_admin`.

Creating a Record if It Doesn't Exist

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

```
1 $list = TodoList::firstOrCreate(
2     [
3         'name' => 'San Juan Vacation',
4         'note' => 'Vacation planning!'
5     ]
6 );
```

If you instead just want to create a new model instance if a record matching the provided attribute isn't found, use `firstOrCreate()`:

```
1 $list = TodoList::firstOrCreate(['name' => 'San Juan Vacation']);
2 $list->note = 'Too much to do before vacation!';
3 $list->save();
```

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

```
1 $list = TodoList::updateOrCreate(
2     ['name' => 'San Juan Vacation'],
3     ['note' => 'Too much to do before vacation!']
4 );
```

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

Implementing the RESTful Insert Feature

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

Updating Existing Records

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

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

Implementing the RESTful Update Feature

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

Deleting Records

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

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

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

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

Implementing the RESTful Destroy Method

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

```
1 public function destroy($id)
2 {
3     TodoList::destroy($id);
4     return redirect()->route('lists.index');
5 }
```

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

Soft Deleting Records

In many cases you won't ever actually want to truly remove records from your database, but instead annotate them in such a way that they are no longer displayed within the application. 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 todo_lists table:

```
1 $ php artisan make:migration add_soft_delete_to_todo_lists \
2 > --table=todo_lists
3 Created Migration: 2016_10_04_221109_add_soft_delete_to_todo_lists
```

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

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

Save the changes and run the migration:

```
1 $ php artisan migrate
2 Migrated: 2016_10_04_221109_add_soft_delete_to_todo_lists
```

After the migration has completed you'll next want to open up the target model and use the `SoftDeletes` trait:

```
1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
6 use Illuminate\Database\Eloquent\SoftDeletes;
7
8 class Todolist extends Model {
9
10     use SoftDeletes;
11
12     protected $dates = ['deleted_at'];
13
14 }
```

Although not strictly necessary, adding the `deleted_at` attribute to the `$dates` array as demonstrated above will cause any returned `deleted_at` values to be of type `Carbon`. Adding timestamps to the `$dates` property is useful because you can use convenient `Carbon` methods such as `toFormattedDateString()` and `toDateTimeString()` to easily present formatted dates and times, not to mention manipulate date/time values in fantastically useful ways. See the [Carbon docs](http://carbon.nesbot.com/docs/)⁸⁶ for more information.

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:

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

⁸⁶<http://carbon.nesbot.com/docs/>

Introducing Query Builder

Chances are you'll be able to carry out most desired database operations using Eloquent, however you'll occasionally want to exercise some 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 `todo_lists` table using Query Builder like this:

```
1 use DB;
2
3 ...
4
5 $lists = DB::table('todo_lists')->get();
```

Prior to 5.3, Query Builder returned result sets in arrays, however as of 5.3 these sets are returned as collections, meaning you can iterate over and interact with these results sets in precisely the same manner as described earlier in this chapter:

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

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

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

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

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

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

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

Unlike earlier examples, this returns an array of objects (even in 5.3).

If for some reason you wanted to insert, update, or delete data using raw SQL, you can use the `DB::insert`, `DB::update`, and `DB::delete` methods, respectively:

```

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

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

```

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

As mentioned, this isn't intended to be anything more than a brief introduction to Query Builder. See [the documentation](http://laravel.com/docs/master/queries)⁸⁷ for a much more comprehensive summary of what's available.

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 `getPropertyAttribute`. Suppose you wanted to override the casing of lists input by users, displaying the list names in solely lowercase. You could use an accessor to do so:

```

1 class TodoList extends Model {
2
3     public function getNameAttribute($value)
4     {
5         return strtolower($value);
6     }
7
8 }
```

With this accessor defined, even if a user's username were set to `WJGilmore` it will always be returned as `wjgilmore` when the username property is referenced:

⁸⁷<http://laravel.com/docs/master/queries>


```
1 $list = TodoList::find(1); // name attribute is San Juan
2 ...
3 [{ $list->name }] // will be returned as san juan
```

Frankly I'm not a fan of accessors, as even if the lowercase list name 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 a `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:

```
1 class User extends Model {
2
3     public function getNameAttribute()
4     {
5         return $this->first_name . " " . $this->last_name;
6     }
7
8 }
```

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

```
1 $list = User::find(12);
2 echo $list->name;
```

Defining Mutators

You'll use a *mutator* (also known as a *setter*) when you'd like to modify the value of an attribute. Sticking with the user-oriented theme, users would logically sign into the application using a username or e-mail address and password. For security reasons the password should be stored in the database using a hashed format (as you'll learn in Chapter 7 Laravel conveniently takes care of all of this for you), 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 create a mutator for the password attribute. Laravel recognizes mutators when the method is defined using the `setAttributeNameAttribute` convention, meaning you'll want to create a method named `setPassword`:

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

This example uses Laravel's Hash class to generate a hash, accepting the plaintext password passed into the method, generating the hash, and assigning the hash to the class instance's password attribute. Here's an example:

```
1 $user = new User;
2 $user->password = 'blah';
3 echo $user->password;
4 $2y$10$e3ufaVnBFWM/SeFc4ZyAhe8u5UR/K0ZUc5IjCPUv0Yv6IVuk7Be7q
```

Defining Custom Methods

Custom methods can greatly reduce the amount of logic cluttering your controllers. For instance, suppose a future version of the List model includes an `due_date` attribute which the user can use to define a date in which the list should be completed. You can define a convenience method to determine whether the list's due date has arrived:

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

This example uses the fantastic [Carbon](#)⁸⁸ library, a PHP 5.3+ DateTime extension which is automatically available to your Laravel applications. With the `isDueToday` method in place, you can easily determine whether a list's due date has arrived:

```
1 $list = TodoList::find(12);
2
3 ...
4
5 @if ($list->isDueToday())
6     This list is due today!
7 @endif
```

Seeding the Database

Many beginning developers tend to spend an inordinate amount of time fretting over populating the development database with an appropriate amount of test data, often doing so by tediously completing and submitting a form (such as one which might be used to create lists for instance). This is a waste of valuable time which can otherwise be spent improving and expanding the application. Instead, you should treat this data as being entirely transient, and take advantage of Laravel's database *seeding* capabilities to quickly and conveniently populate the database for you.

As an added bonus, you can use Laravel's database seeding capabilities for 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, it will seem like 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:

⁸⁸<https://github.com/briannesbitt/Carbon>

```

1  <?php
2
3  use Illuminate\Database\Seeder;
4
5  class DatabaseSeeder extends Seeder
6  {
7
8      public function run()
9      {
10         // $this->call(UsersTableSeeder::class);
11     }
12
13 }
```

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 classes, and when `db:seed` executes, the instructions found in those seeder classes will be executed as well. You'll find an example call to a hypothetical class called `UsersTableSeeder` in the `DatabaseSeeder` file:

```
1  // $this->call('UsersTableSeeder::class');
```

The corresponding `UsersTableSeeder.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. You can create the seeder skeleton by executing the `db:seed` command:

```

1  $ php artisan make:seed TodoListTableSeeder
2  Seeder created successfully.
```

You'll find the newly created `TodolistTableSeeder.php` file in `database/seeds/`. Update the file to look like this:

```

1  <?php
2
3  namespace App;
4
5  use App\TodoList;
6
7  use Illuminate\Database\Seeder;
8
9  class TodoListTableSeeder extends Seeder {
10
```

```
11 public function run()
12 {
13
14     TodoList::create([
15         'name' => 'San Juan Vacation',
16         'note' => 'Things to do before we leave for Puerto Rico!'
17     ]);
18
19     TodoList::create([
20         'name' => 'Home Winterization',
21         'note' => 'Winter is coming.'
22     ]);
23
24     TodoList::create([
25         'name' => 'Rental Maintenance',
26         'note' => 'Cleanup and improvements for new tenants'
27     ]);
28
29 }
30
31 }
```

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

```
1 DB::table('todo_lists')->truncate();
2 $this->call('App\TodoListTableSeeder');
```

After saving the changes you'll need to rebuild the classmap. This is important! Neglecting to do this will result in lost time and tears:

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

Finally, run the seeder anew:

```
1 $ php artisan db:seed
2 Seeded: App\TodoListTableSeeder
```

Check your project database and you should see several new records in the `todo_lists` 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](https://github.com/fzaninotto/Faker)⁸⁹ (which is automatically included in Laravel projects) to easily create large amounts of sample data.

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

```
1 class TodoListTableSeeder extends Seeder {
2
3     public function run()
4     {
5
6         $faker = \Faker\Factory::create();
7
8         foreach(range(1,50) as $index)
9         {
10
11             TodoList::create([
12                 'name'          => $faker->sentence(2),
13                 'description' => $faker->sentence(4),
14             ]);
15
16         }
17
18     }
19
20 }
```

In this example a new instance of the `Faker` class is created. 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](http://en.wikipedia.org/wiki/Lorem_ipsum)⁹⁰-style sentences consisting of two and four words, respectively.

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

⁸⁹<https://github.com/fzaninotto/Faker>

⁹⁰http://en.wikipedia.org/wiki/Lorem_ipsum

```

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

```

Random sentence generation is only a small part of what Faker can do. You can also use Faker to generate random numbers, lorem ipsum paragraphs, male and female names, U.S. addresses, U.S. phone numbers, company names, e-mail addresses, URLs, credit card information, colors, and more! Be sure to check out the [Faker documentation](https://github.com/fzaninotto/Faker)⁹¹ for examples of these other generators.

Creating Sluggable URLs

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

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

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

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

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

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

This string-based parameter is known as a *slug*, and thanks to the [eloquent-sluggable](https://github.com/cviebrock/eloquent-sluggable)⁹² package it's surprisingly easy to integrate sluggable URLs into your Laravel application. Begin by adding the eloquent-sluggable package to your `composer.json` file:

⁹¹<https://github.com/fzaninotto/Faker>

⁹²<https://github.com/cviebrock/eloquent-sluggable>

```
1 $ composer require cviebrock/eloquent-sluggable
```

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

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

After saving these changes, you're ready to begin creating sluggable URLs!

Creating Sluggable Models

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

```
1 use Cviebrock\EloquentSluggable\Sluggable;  
2  
3 class TodoList extends Model {  
4  
5     use Sluggable;  
6  
7     public function sluggable()  
8     {  
9         return [  
10             'slug' => [  
11                 'source' => 'name'  
12             ]  
13         ];  
14     }  
15  
16     ...  
17  
18 }
```

There are several important changes to this model, including:

- The `Sluggable` trait is referenced at the top of the file.
- Inside the class body you'll see the model uses the `Sluggable` trait.

- An method named `sluggable` is defined which identifies the database column where the slug should be saved, and the source column from where the slug should be created.

After saving the model changes you'll need to create a migration which adds the `slug` column to the table:

```
1 $ php artisan make:migration add_slug_column_to_todo_lists_table \  
2 > --table=todo_lists  
3 Created Migration: 2016_10_04_225240_add_slug_column_to_todo_lists_table
```

Open the newly created migration and modify the up and down methods to look like this:

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

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

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

With the slugs in place, all you need to do is retrieve the desired record by slug rather than the integer ID. You've already learned how to do this earlier in the chapter however the package provides a useful helper method for doing so called `findBySlug()`. To use it you'll need to modify the model to also use the `SluggableScopeHelpers` trait:

```
1 use Cviebrock\EloquentSluggable\SluggableScopeHelpers;
2
3 class TodoList extends Model
4 {
5
6     use Sluggable, SluggableScopeHelpers;
7
8     ...
9
10 }
```

For instance to find the list record associated with the slug `san-juan-vacation` you'll execute the following command:

```
1 $list = TodoList::findBySlug('san-juan-vacation');
```

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

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

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

Testing Your Models

Testing your models to ensure they are performing as desired is a crucial part of the application development process. Mind you, the goal isn't to test Eloquent's features; one can presume Eloquent continues to be tested by the Laravel development team. Instead, you want to focus on confirming proper functioning of features you incorporate into the application models, such as whether your model accessors and mutators are properly configured, whether your custom methods are behaving as expected, and as you'll learn in the next chapter, whether features such as relations and scopes are correctly defined. With this in mind, let's take some time to investigate a few testing scenarios. I'll presume you've successfully configured your Laravel testing environment as described in Chapter 1, however we have a bit of additional configuration work to do in order to talk to the database which I'll describe next.

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 adding test-specific environment variables to the `phpunit.xml` file, as this file is read in each time your test suite runs:

```
1 <env name="DB_HOST" value="127.0.0.1" />
2 <env name="DB_PORT" value="3306" />
3 <env name="DB_DATABASE" value="test_todoparrot_com" />
4 <env name="DB_USERNAME" value="test_todoparrot" />
5 <env name="DB_PASSWORD" value="secret" />
```

Of course, you'll also need to create this database and the authentication credentials.

Automatically Rebuilding the Test Database

It is crucial for you to ensure that your database structure and test data are in a *known state* prior to the execution of each and every test, otherwise you're likely to introduce all sorts of uncertainty into the very situations you're trying to verify. One foolproof way to do this is by completely tearing down and rebuilding your test database structure prior to and following each test, respectively. Fortunately, as of Laravel 5.1 doing so is a trivial task! Just add the following line to your test class (I'll show you how to generate this class in a moment):

```
1 <?php
2
3 use Illuminate\Foundation\Testing\WithoutMiddleware;
4 use Illuminate\Foundation\Testing\DatabaseMigrations;
5 use Illuminate\Foundation\Testing\DatabaseTransactions;
6
7 class TodoListTest extends TestCase
8 {
9
10     use DatabaseMigrations;
11
12 }
```

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

If your tests do not necessarily require the schema to be completely destroyed and rebuilt, you might instead consider using the `DatabaseTransactions` trait since it will encapsulate each test's database queries in a transaction, and roll back the transactions following each test's conclusion:

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

Creating a Model Factory

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

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

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

Add the following `TodoList` factory below the factory defined above:

```
1 $factory->define(App\TodoList::class, function ($faker) {
2     return [
3         'name' => $faker->sentence(2),
4         'note' => $faker->sentence(4)
5     ];
6 });
```

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

```
1 $listFactory = factory('App\TodoList')->create();
```

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

```
1 $listFactory = factory('App\TodoList')->make();
```

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

Creating Your First Model Test

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

```
1 <?php
2
3 use Illuminate\Foundation\Testing\WithoutMiddleware;
4 use Illuminate\Foundation\Testing\DatabaseMigrations;
5 use Illuminate\Foundation\Testing\DatabaseTransactions;
6
7 use App\TodoList;
8
9 class TodoListTest extends TestCase
10 {
11
12     use DatabaseMigrations;
```

```
13
14     public function testCanInstantiateTodolist()
15     {
16
17         $list = new TodoList;
18
19         $this->assertEquals(get_class($list), 'App\TodoList');
20
21     }
22
23 }
```

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

```
1 $ phpunit tests/models/TodoListTest.php
2 PHPUnit 5.5.4 by Sebastian Bergmann and contributors.
3
4 .
5
6 Time: 147 ms, Memory: 14.50Mb
7
8 OK (1 test, 1 assertion)
```

Great! Let's try something a tad more involved. How about a test involving the previously created `Todolist` factory? Add the following method to the `TodoListTest` class:

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

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

Summary

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

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

Thus far we've been taking a fairly simplistic view of the project database, interacting with a single model (`TodoList`) and its underlying `todo_lists` table. However, in the real world an application's database tables are like an interconnected archipelago, with bridges interconnecting two or even more islands. These allegorical causeways are what allows the developer to efficiently determine for instance which tasks are related to a particular list, associate a specific user with a set of lists, and find all users identified as living in the state of Ohio.

Such relationships are possible thanks to a process known as *database normalization*⁹³. Database normalization is an approach to data organization which 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, route model binding, and eager loading.

Introducing Relations

Although relatively simplistic as compared to other applications, the project database very much resembles the allegorical archipelago. For instance, each list found is mapped to a single category and user. Each list item is associated with 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? Such capabilities are surprisingly easy once you have the hang of things.

Laravel supports several types of relations, including:

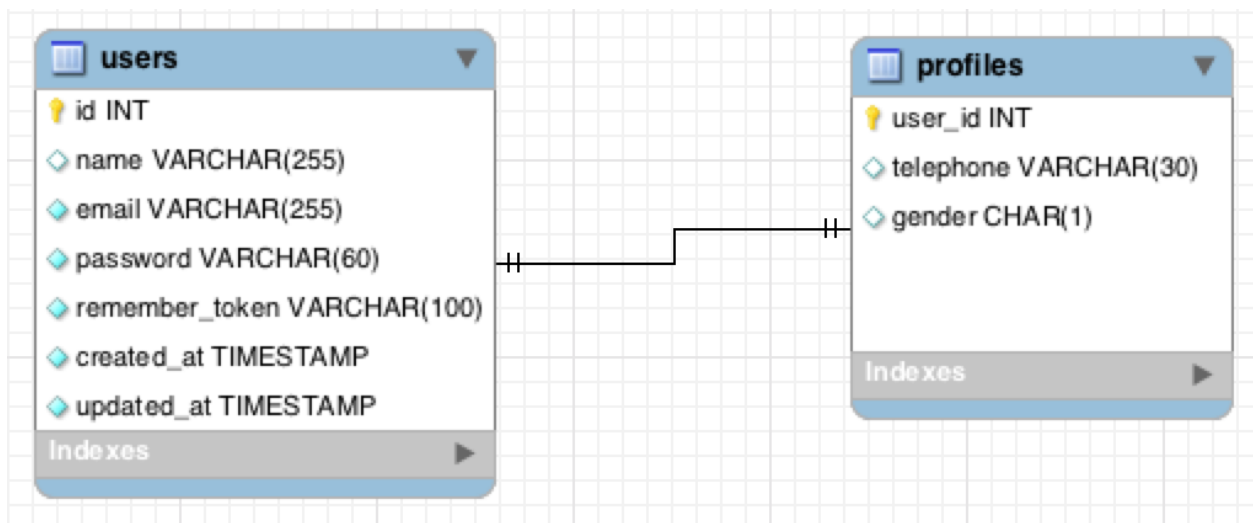
- **One-to-One Relation:** One-to-One relations are used when one entity can only belong with one other entity. For instance for organizational reasons you might choose to separate user authentication information (e-mail address and password) from profile-related characteristics such as their name, bio 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 for purely organizational purposes, because there is often little reason to manage uniquely-related data in multiple tables.

⁹³http://en.wikipedia.org/wiki/Database_normalization

- **Belongs To Relation:** The Belongs To relation assigns ownership of a particular record to another record. For instance, a task *belongs to* a list by configuring the database in such a way that each task record points to the list record for which it belongs. This is an important concept that often trips up beginners: always remember that the record doing the pointing is identified as *belonging to* the pointed-at record.
- **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, and each task *belongs to* a list.
- **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 associated to many categories, and each category could be associated with many books.
- **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 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 list's associated user. You can simplify the process used to perform this sort of analysis using the Has Many Through relation.
- **Polymorphic Relation:** Polymorphic relations are 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 several different types of data (blog entries and products, for instance). It would be inefficient to create separate comment-specific models/tables for these different types of data, 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 table. However, for the sake of demonstration let's suppose your application offered user authentication and profile management, and you wanted to separate the user's authentication (e-mail address, password) and profile (name, phone number, gender) data into two separate tables. This relationship is depicted in the below diagram.



An example one-to-one relationship

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

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

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

```
1 namespace App;
2
3 use Illuminate\Database\Eloquent\Model;
4
5 class Profile extends Model {
6
7     //
8
9 }
```

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

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

The bolded lines are the only four you'll need to add, as the other two will already be in place when you open the file. The first line results in the addition of an integer-based column named `user_id`. The second line identifies this column as being a foreign key which references the `users` table's `id` column.



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

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

```
1 $ php artisan migrate
2 Migrated: 2016_10_05_015236_create_profiles_table
```

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

Defining the One-to-One Relation

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

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

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

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

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

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

Creating a One-to-One Relation

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

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

Once executed, you'll find a newly created profile in the `profiles` table with a `user_id` of 1.

Deleting a One-to-One Relation

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

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

Configuring Cascading Deletions

It is unlikely you'd want a profile's user to be deleted without also deleting the profile, since it would result in an *orphaned* profile (a profile record which points to a nonexistent user ID). One way to avoid this is by deleting the related profile record before deleting the user record, 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 the task by taking advantage of the underlying database's ability to delete child tables when the parent table is deleted. You can specify this requirement when defining the foreign key in your table migration. I've modified the relevant lines of the earlier migration used to create the `profiles` table, attaching the `onDelete` option to the foreign key:

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

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

Introducing the Belongs To Relation

With the above example's `hasOne` relation in mind, 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 `belongsToMany` relation in the `Profile` model, as demonstrated here:

```
1 class Profile extends Model {
2
3     public function user()
4     {
5         return $this->belongsTo('App\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     ->first()->user->email;
```

Belongs To certainly isn't limited to use in conjunction with One-to-One relations, and in fact you'll rarely use it in this fashion although at this point in the chapter demonstrating it in this capacity seems warranted. It will become quite clear as this chapter progresses just how indispensable Belongs To is when used in conjunction with other relations such as Has Many.

Introducing One-to-Many Relations

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

Creating the Task Model

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

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

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

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

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

```
1 public function up()
2 {
3     Schema::create('tasks', function(Blueprint $table)
4     {
5         $table->increments('id');
6         $table->integer('todo_list_id')->unsigned();
7         $table->foreign('todo_list_id')
8             ->references('id')->on('todo_lists')
9             ->onDelete('cascade');
10        $table->string('name');
11        $table->text('description');
12        $table->boolean('done')->default(false);
13        $table->timestamps();
14    });
15 }
16
17 public function down()
18 {
19     Schema::drop('tasks');
20 }
```

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

```
1 $ php artisan migrate
2 Migrated: 2016_10_05_005821_create_tasks_table
```

Defining the One-to-Many Relation

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

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

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

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

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

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

Associating Tasks with a TODO List

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


```
1 use App\Task;
2 use App\TodoList;
3
4 ...
5
6 $list = TodoList::find(1);
7
8 $task = new Task;
9 $task->name = 'Walk the dog';
10 $task->note = 'Walk Barky the Mutt';
11
12 $list->tasks()->save($task);
13
14 $task = new Task;
15 $task->name = 'Make tacos for dinner';
16 $task->description = 'Mexican sounds really yummy!';
17
18 $list->tasks()->save($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 and view (created in the last chapter) to additionally display list tasks. The `Lists` controller's `show` action doesn't actually change at all, but I'll include it here anyway for easy reference:

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

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

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

Sorting and Filtering Related Records

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

```

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

```

If you plan on always filtering tasks in this fashion you can clean up the code a bit by enabling the filter within the relationship definition:

```

1 public function tasks()
2 {
3     return $this->hasMany('App\Task')
4         ->where('done', true);
5 }

```

Modifying a relationship’s default behavior is also useful for sorting. For instance, you can use `orderBy` to determine the order in which a category’s tasks are returned:

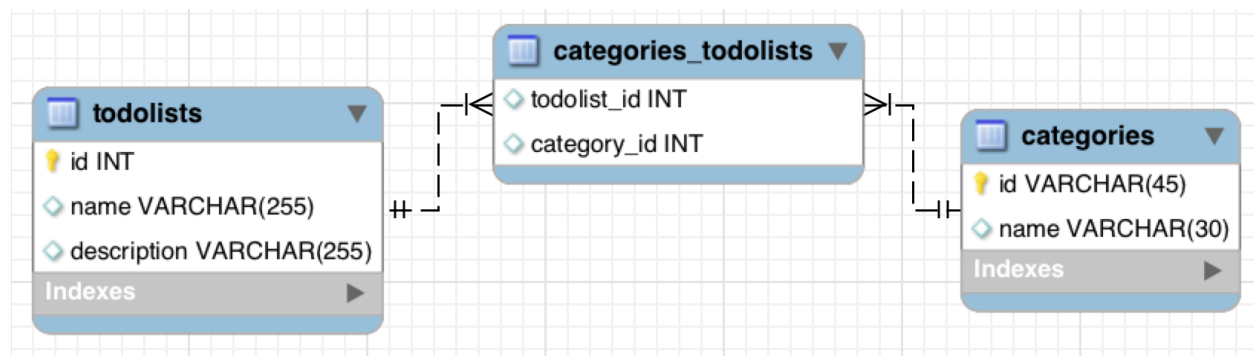
```

1 public function tasks()
2 {
3     return $this->hasMany('App\Task')
4         ->orderBy('created_at', 'desc')
5         ->where('done', true);
6 }

```

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. The pivot table name should be formed by concatenating the two related model names together with an underscore separating the names. Further, 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_todo_list`. 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 --migration
```

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

```
1 use Illuminate\Database\Eloquent\Model;
2
3 class Category extends Model {
4
5 }
```

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

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

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

```
1 $ php artisan migrate
```

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

```
1 $ php artisan make:migration create_category_todo_list_table \  
2 --create=category_todo_list  
3 Created Migration: 2016_10_05_011409_create_category_todo_list_table
```

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

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

This syntax is no different than that used for the earlier `tasks` table migration. The only real difference is that we're referencing two foreign keys rather than one.

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

```
1 $ php artisan migrate
```

Defining the Many-to-Many Relation

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

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

Notice I've chained the `withTimestamps` method to the return statement. This instructs Laravel to additionally update the `category_todo_list` 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 `$table->timestamps` from the migration), you can omit the `withTimestamps` method).

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

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

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

Associating Records Using the Many-to-Many Relation

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

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



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

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

```

1  mysql> select * from categories;
2  +----+-----+-----+-----+
3  | id | name      | created_at          | updated_at          |
4  +----+-----+-----+-----+
5  |  1 | Vacation | 2017-10-05 20:44:11 | 2016-10-05 20:44:11 |
6  +----+-----+-----+-----+
7
8  mysql> select * from category_todo_list;
9  +-----+-----+-----+-----+
10 | category_id | todo_list_id | created_at | updated_at |
11 +-----+-----+-----+-----+
12 |           1 |           1 | ...       | ...       |
13 +-----+-----+-----+-----+

```

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

```

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

```

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

```

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

```

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

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

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

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

Determining if a Relation Already Exists

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

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

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

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

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


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

Saving Multiple Relations Simultaneously

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

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

Traversing the Many-to-Many Relation

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

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

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

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

Synchronizing Many-to-Many Relations

Suppose you provide users with a multiple selection box that allows users to easily associate a list with one or more categories. Because the user can both select and deselect categories, you must take

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

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

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

Managing Additional Many-to-Many Attributes

Thus far the many-to-many examples presented in this chapter have been concerned with a join table consisting of two foreign keys and optionally the `created_at` and `updated_at` timestamps. But what if you wanted to manage additional attributes within this table, such as some additional description pertaining to the list/category relation?

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_todo_list` table created earlier in this section:

```
1 $ php artisan make:migration add_description_to_category_todo_list_table
2 Created Migration: 2016_10_05_012822_add_descripti...
```

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

```

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

```

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

```

1  $ php artisan migrate
2  Created Migration: 2016_10_05_012822_add_descript...

```

Finally, you'll need to modify the `TodoList` categories relation to identify the additional pivot column using the `withPivot()` method:

```

1  public function categories()
2  {
3      return $this->belongsToMany('App\Category')
4          ->withPivot('description')
5          ->withTimestamps();
6  }

```

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

```

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

```

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

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

Introducing Has Many Through Relations

Suppose TODOParrot’s CEO has just returned from the “Mo Big Data Mo Money” conference, flush with ideas regarding how user data can be exploited and sold to the highest bidder. 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. This means the user/country relations would look like this:

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

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

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

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

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

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

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

Introducing Polymorphic Relations

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

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

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

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

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

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

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

Finally, save the changes and run the migration:

```
1 $ php artisan migrate
2 Migrated: 2016_10_06_223902_create_comments_table
```

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

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

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

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

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

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

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

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

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


```

1 mysql> select * from comments;
2 +----+-----+-----+-----+-----+...
3 | id | body          | commentable_id | commentable_type | created_at | ...
4 +----+-----+-----+-----+-----+...
5 | 1  | Great work!   | 1              | App\ToDoList     | 2016-...   | ...
6 +----+-----+-----+-----+-----+...

```

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

```

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

```

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

```

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

```

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

Eager Loading

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

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

The `take()` method is Laravel syntax for limiting the number of returned records. Here's the query translated into SQLese:

```
1 select * from `users` limit 5
```

Thanks to Laravel's fantastic [Eloquent ORM](https://laravel.com/docs/master/eloquent)⁹⁴, you can pass the query results into your view and easily retrieve data associated via the `User` model's associations. Consider for instance a separate `State` model used to house a list of U.S. states. This `State` model would be related to `User` via a *HasMany* association:

⁹⁴<https://laravel.com/docs/master/eloquent>

```
1 class State extends Model
2 {
3
4     public function users()
5     {
6         return $this->hasMany( 'App\User' );
7     }
8
9 }
```

The User model would be related to State via a *BelongsTo* association:

```
1 class User extends Model
2 {
3
4     public function state()
5     {
6         return $this->belongsTo( 'App\State' );
7     }
8
9 }
10
11 ...
```

With this association in place, when the results are passed into the view we can easily retrieve the user's name and state:

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

Pretty innocent bit of code, right? It certainly seems so until you realize the initial query and ensuring iteration results in the execution of *6 queries*! Thus the name “N + 1”, because we’re executing one query to retrieve the five locations, and then *five additional queries* to retrieve the name of each user’s state name! This is because Laravel (and all other mainstream frameworks, for that matter) will by default not load each user’s state object until necessary.

In situations where you know you’re going to need to access an associated object you can use the `with` method to inform Laravel of your intent to do so and therefore preload the data:

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

When you subsequently access a User object's state name, the data will be immediately available because each user's state-related data was preloaded along with the original query! We can confirm this by firing up Tinker and comparing the two approaches:

```
1 >>> User::take(5)->get();
2 => Illuminate\Database\Eloquent\Collection {#720
3   all: [
4     App\User {#730
5       id: 1,
6       email: 'joe@example.com',
7       name: "Joe Smith",
8       created_at: "2016-04-05 01:05:37",
9       updated_at: "2016-04-05 01:05:37",
10    },
11    App\User {#730
12      id: 2,
13      email: 'mary@example.com',
14      name: "Mary Smith",
15      created_at: "2016-04-05 02:05:37",
16      updated_at: "2016-04-05 02:05:37",
17    },
18    ...
19  ],
20 }
```

And here is the output when with() is used. Note how each user is accompanied by a state object:

```
1 >>> User::take(5)->get();
2 => Illuminate\Database\Eloquent\Collection {#720
3   all: [
4     App\User {#730
5       id: 1,
6       email: 'joe@example.com',
7       name: "Joe Smith",
8       created_at: "2016-04-05 01:05:37",
9       updated_at: "2016-04-05 01:05:37",
10      state: App\State {#735
11        id: 1,
12        name: 'Ohio'
13      },
14    },
15    ...
16  ],
17 }
```

```
14     },
15     App\User {#730
16         id: 2,
17         email: 'mary@example.com',
18         name: "Mary Smith",
19         created_at: "2016-04-05 02:05:37",
20         updated_at: "2016-04-05 02:05:37",
21         state: App\State {#735
22             id: 2,
23             name: 'New York'
24         },
25     },
26     ...
27 ],
28 }
```

There you have it, eager loading demystified! Be sure to incorporate this approach into your applications, I guarantee you'll see significant performance improvements as a result of making this seemingly minor change.

Introducing Scopes

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

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

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

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

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

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

Creating Dynamic Scopes

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

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

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

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

Using Scopes with Relations

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

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

Summary

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

Chapter 5. Integrating Web Forms

Chances are you're going to spend quite a bit of time and effort 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](#)⁹⁵. While creating the HTML used to display these forms is easy enough, figuring out how to integrate the HTML alongside your models in a sane and secure fashion can quickly become confusing. Never fear though because in this chapter I'll show you how to wield total control over your Laravel forms, covering a variety of form integration scenarios you're sure to encounter when embedding forms into your future applications.

Web Form Fundamentals

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

The following example incorporates [HTML5-specific markup](#)⁹⁶, PHP-specific security features, and [Bootstrap-specific markup](#)⁹⁷ to produce a flexible, secure, and responsive form:

```
1 <form method="POST" action="/contact"
2   accept-charset="UTF-8" class="form">
3
4 <input name="_token" type="hidden" value="YLMxbvKETQ4Tz6zVuWhd6Xb1">
5
6 <div class="form-group">
7   <label for="name">Your Name</label>
8   <input class="form-control" placeholder="Your name"
9     name="name" type="text">
10 </div>
11
12 <div class="form-group">
13   <label for="email">Your E-mail Address</label>
```

⁹⁵<http://todoparrot.com/contact>

⁹⁶<http://diveintohtml5.info/forms.html>

⁹⁷<http://getbootstrap.com/>

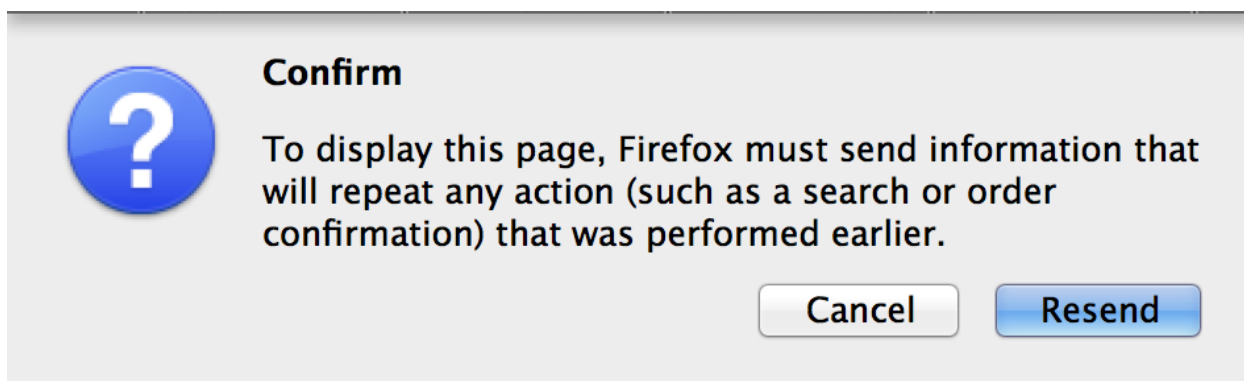
```
14   <input class="form-control" placeholder="Your e-mail address"
15       name="email" type="text">
16 </div>
17
18 <div class="form-group">
19   <label for="message">Your Message</label>
20   <textarea class="form-control" name="message" ></textarea>
21 </div>
22
23 <div class="form-group">
24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

Let's review the most relevant form features, beginning with the form enclosure. The form tag encloses the fields and other markup comprising the form:

```
1 <form method="POST" action="/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 developer not have taken additional steps to prevent unwanted consequences* when a POST form be submitted more than once.

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

The `action` attribute defines where the user will be taken when the form is submitted. Logically this should match a route definition found in your `routes/web.php` 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="pEa4MGDfD2ESgIeeGxWxGmV">
```

This field is security-related, preventing [cross-site request forgery \(CSRF\)](http://en.wikipedia.org/wiki/Cross-site_request_forgery)⁹⁸ by storing the randomly generated value in the field and also in a session variable. When the form is submitted, that field's value is compared with the session variable stored on the client to ensure they match; without such safeguards it would be possible for third parties 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:

⁹⁸http://en.wikipedia.org/wiki/Cross-site_request_forgery

```
1 <label for="name">Your Name</label>
2 <input class="form-control" placeholder="Your name"
3   name="name" type="text" required="required">
4
5 <label for="email">Your E-mail Address</label>
6 <input class="form-control" placeholder="Your e-mail address"
7   name="email" type="text" required="required">
8
9 <label for="message">Your Message</label>
10 <textarea class="form-control" name="message"
11   required="required"></textarea>
```

There's not much to discuss regarding these fields other than to remember each element's name attribute assigns the variable name subsequently used to retrieve whatever value the user provides when completing the form. Exactly how this is done will soon become apparent.

With this general overview complete, I'll spend the remainder of this chapter discussing Laravel's form integration features, drawing upon numerous examples found in the TODOParrot code.

Creating a User Feedback Form

Let's kick things off by building a simple contact form consisting of three fields, including the user's name, email address, and message (see below figure).

Your Name

Your name

Your E-mail Address

Your e-mail address

Your Message

Your message

Contact Us!

TODOParrot's contact form

Although we could create a contact-specific 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>⁹⁹) 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 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 `routes/web.php` file:

⁹⁹<http://todoparrot.com/about>

```
1 Route::get('contact',
2     [
3         'as' => 'contact',
4         'uses' => 'AboutController@create'
5     ]
6 );
7 Route::post('contact',
8     [
9         'as' => 'contact_store',
10        'uses' => 'AboutController@store'
11    ]
12 );
```

Next, you'll need to add the create and store actions to the newly created About controller, because we didn't specify the `--resource` option when generating the controller. Modify the controller to look like this:

```
1 namespace App\Http\Controllers;
2
3 use Illuminate\Http\Request;
4
5 use App\Http\Requests;
6
7 class AboutController extends Controller {
8
9     public function create()
10     {
11         return view('about.contact');
12     }
13
14     public function store()
15     {
16     }
17
18 }
```

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. Note my emphasis on *rendered* because you won't actually hand-code the form! Instead, I suggest you use a fantastic Laravel

package called [LaravelCollective/html](https://github.com/LaravelCollective/html)¹⁰⁰ (introduced in Chapter 2) to manage this tedious task for you. Below I've pasted in the section of code found in `TODOParrot's resources/views/about/contact.blade.php` view that's responsible for generating the contact form:

```

1  @extends('layouts.app')
2
3  @section('content')
4
5  <h1>Contact TODOParrot</h1>
6
7  <ul>
8      @foreach($errors->all() as $error)
9          <li>{{ $error }}</li>
10     @endforeach
11 </ul>
12
13 {!! Form::open(['route' => 'contact_store', 'class' => 'form']) !!}
14
15 <div class="form-group">
16     {!! Form::label('Your Name') !!}
17     {!! Form::text('name', null,
18         ['required',
19             'class'=>'form-control',
20             'placeholder'=>'Your name']) !!}
21 </div>
22
23 <div class="form-group">
24     {!! Form::label('Your E-mail Address') !!}
25     {!! Form::text('email', null,
26         ['required',
27             'class'=>'form-control',
28             'placeholder'=>'Your e-mail address']) !!}
29 </div>
30
31 <div class="form-group">
32     {!! Form::label('Your Message') !!}
33     {!! Form::textarea('message', null,
34         ['required',
35             'class'=>'form-control',
36             'placeholder'=>'Your message']) !!}
37 </div>

```

¹⁰⁰<https://github.com/LaravelCollective/html>

```

38
39 <div class="form-group">
40     {!! Form::submit('Contact Us!',
41         ['class'=>'btn btn-primary']) !!}
42 </div>
43 {!! Form::close() !!}
44
45 @endsection

```

You learned how to install this package in Chapter 2, but at that point we just configured the HTML Facade. To take advantage of the form-specific tags you'll need to additionally add the following alias to the config/app.php aliases array:

```

1 'Form'=> Collective\Html\FormFacade::class

```

If this is your first encounter with the `Form::open` helper then I'd imagine this example looks a tad scary. However once you build a few forms in this fashion I promise you'll wonder how you ever got along without it. Let's break down the key syntax used in this example:

```

1 {!! Form::open(['route' => 'contact_store', 'class' => 'form']) !!}
2 ...
3 {!! Form::close() !!}

```

The `Form::open` and `Form::close()` methods work together to generate the form's opening and closing tags. The `Form::open` method accepts an array containing various settings such as the route alias which in this case points to the About controller's store method, and a class used to stylize the form. The default method is POST however you can easily override the method to instead use GET by passing `'method' => 'get'` into the array. Additionally, the `Form::open` method will ensure the aforementioned CSRF-prevention `_token` hidden field is added to the form.

Next up you'll see the following `@foreach` block:

```

1 <ul>
2     @foreach($errors->all() as $error)
3         <li>{{ $error }}</li>
4     @endforeach
5 </ul>

```

This block is used to output any validation errors should one or more of the user-supplied field values not pass the validation tests (more on this in a moment).

Next you'll see a series of methods used to generate the various form fields. This is a relatively simplistic form therefore only a few of the available field generation methods are used,

including `Form::label` (for creating form field labels), `Form::text` (for creating form text fields), `Form::textarea` (for creating a form text area), and `Form::submit` (for creating a submit button). Note how the `Form::text` and `Form::textarea` methods all accept as their first argument a model attribute name (`name`, `email`, and `message`, respectively). All of the methods also accept an assortment of other options, such as class names and HTML5 form attributes.

Once you add this code to your project's `resources/views/about/contact.blade.php` file, navigate to `/contact` and you should see the same form as that found at <http://todoparrot.com/contact!>

With the form created, we'll next need to create the logic used to process the form contents and send the feedback to the site administrator via e-mail.

Creating the Contact Form Request

Laravel 5 introduced a new feature known as a *form request*. This feature is intended to remove form authorization and validation logic from your controllers, instead placing this logic in a separate class. TODOParrot uses form requests in conjunction with each form used throughout the site and I'm pleased to report this feature works meets its goal quite nicely.

To create a new form request you can use Artisan's `make:request` feature:

```
1 $ php artisan make:request ContactFormRequest
2 Request created successfully.
```

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

```
1 namespace App\Http\Requests;
2
3 use Illuminate\Foundation\Http\FormRequest;
4
5 class ContactFormRequest extends FormRequest {
6
7     public function authorize()
8     {
9         return false;
10    }
11
12    public function rules()
13    {
14        return [
15            //
16        ];
17    }
18 }
```

```
17     }
18
19 }
```

The `authorize` method determines whether the current user is authorized to interact with this form. I'll talk more about the purpose of this method in Chapter 7. For the purposes of the contact form we want any visitor to submit a contact request and so modify the method to return `true` instead of `false`:

```
1 public function authorize()
2 {
3     return true;
4 }
```

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:

```
1 public function rules()
2 {
3     return [
4         'name'      => 'required',
5         'email'     => 'required|email',
6         'message'  => 'required'
7     ];
8 }
```

The `required` and `email` validators used in this example are just a few of the many supported validation features. See Chapter 3 for more information about these rules. In the examples to come I'll provide additional examples demonstrating other available validators. Additionally, note how you can use multiple validators in conjunction with a form field by concatenating the validators together using a vertical bar (`|`).

After saving the changes to `ContactFormRequest.php` open the `About` controller and modify the `store` method to look like this:


```
1 use App\Http\Requests\ContactFormRequest;
2
3 ...
4
5 class AboutController extends Controller {
6
7     public function store(ContactFormRequest $request)
8     {
9
10         return redirect()-route('contact')
11             ->with('message', 'Thanks for contacting us!');
12
13     }
14
15 }
```

Note we've imported the `ContactFormRequest` class into the controller, and additionally declared the `$request` parameter to be of type `ContactFormRequest`. Although we haven't yet added the e-mail delivery logic, believe it or not this action is otherwise complete. This is because the `ContactForm` request will automatically handle the validation *and* display of validation error messages should validation fail. For instance submitting the contact form without completing any of the fields will result in three validation error found presented in the below screenshot being displayed:

Contact TODOParrot

- The name field is required.
- The email field is required.
- The message field is required.

Your Name

Your E-mail Address

Displaying contact form validation errors

These errors won't appear out of thin air of course; they'll be displayed via the `$errors` array included in the `contact.blade.php` view:

```
1 <ul>
2   @foreach($errors->all() as $error)
3     <li>{{ $error }}</li>
4   @endforeach
5 </ul>
```

You'll also want to inform the user of a successful form submission. To do so you can use a flash message, which is populated in the store method ("Thanks for contacting us!"). The message variable defined within 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 app.blade.php` so flash messages can be retrieved and displayed above any view:

```
1 @if (Session::has('message'))
2     <div class="alert alert-info">
3         {{ Session::get('message') }}
4     </div>
5 @endif
```

Only one step remains before the contact form is completely operational. We'll need to configure Laravel's mail component and integrate e-mail delivery functionality into the store method. Let's complete these steps next.

Configuring Laravel's Mail Component

Thanks to integration with the popular [SwiftMailer](http://swiftmailer.org/)¹⁰¹ 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](http://www.mailgun.com/)¹⁰² and [Mandrill](https://mandrill.com/)¹⁰³ 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 sets the host address of your SMTP server.
- **port:** The port setting sets the port used by your SMTP server.

¹⁰¹<http://swiftmailer.org/>

¹⁰²<http://www.mailgun.com/>

¹⁰³<https://mandrill.com/>

- `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 sets the encryption protocol used when sending e-mails.
- `username`: The `username` setting sets the account username.
- `password`: The `password` setting sets the account password.
- `sendmail`: The `sendmail` setting sets the server Sendmail path should you be using the `sendmail` driver.

Although you'll commonly find tutorials demonstrating how to use a Gmail account to send e-mail through a PHP application, I suggest against doing so. Using an Gmail account for such purposes may seem convenient because of their ubiquity, however Google has made it increasingly difficult to use Gmail accounts in this manner due largely to security and spamming concerns. Instead, consider using a third-party service created precisely for such purposes. One solution is [mailgun](https://www.mailgun.com/)¹⁰⁴, which offers a free pricing tier for users sending less than 10,000 e-mails per month.



While we all love no-cost solutions, be aware the pricing terms can change with little notice. Previous editions of this book promoted another solution which pulled the rug on their free tier after building a rather large following. I won't do them the favor of mentioning the name here, and only say you should be prepared to switch from one service to another should pricing terms suddenly become unfavorable.

To create a Mailgun account head over to <https://mailgun.com/>¹⁰⁵. After creating your account you'll be provided with a sandbox console which you can use to experiment with sending e-mail using [curl](https://curl.haxx.se/)¹⁰⁶ or a variety of programming languages (PHP included). Don't worry about this for now because Laravel includes Mailgun support so you won't have to worry about integrating any custom PHP code to send e-mail from your application.

You'll also be prompted to identify the domain from which your Mailgun-managed e-mails will be sent. At this point in time you're not required to do so in order to begin integrating Mailgun into your application because Mailgun also creates a sandbox domain which can be used to send up to 300 messages daily. To demonstrate integration we'll use this sandbox domain so click the `Continue To Your Control Panel` link to continue. You'll be taken to the Mailgun dashboard where you'll find your sandbox domain. Click on the domain name and you'll be taken to a page containing your sandbox domain authentication credentials. This is where configuration can be a bit confusing, because you actually won't need to modify the `config/mail.php` settings when using a third-party service such as Mailgun; instead, you'll only need to be concerned with the following settings found in `config/services.php`:

¹⁰⁴<https://www.mailgun.com/>

¹⁰⁵<https://mailgun.com/>

¹⁰⁶<https://curl.haxx.se/>

```

1  'mailgun' => [
2      'domain' => env('MAILGUN_DOMAIN'),
3      'secret' => env('MAILGUN_SECRET'),
4  ],

```

If you return to the Mailgun domain settings for your sandbox, you'll find both the domain (e.g. sandbox123456.mailgun.org) and the secret (identified as the API KEY). Add the following two lines to your .env file, updating my placeholders to reflect your actual credentials:

```

1  MAILGUN_DOMAIN=sandbox123456.mailgun.org
2  MAILGUN_SECRET=key-supersecret

```

Additionally, add the following three variables, which we'll use when sending the e-mail. Be sure to update MAIL_FROM to reflect the username associated with your sandbox domain settings. You can set MAIL_TO to the desired recipient, and MAIL_NAME just identifies the name associated with the sender's address:

```

1  MAIL_FROM=postmaster@sandbox123456.mailgun.org
2  MAIL_NAME="TODOParrot Support"
3  MAIL_TO=wj@wjgilmore.com

```

After saving these changes, we'll need to install the Guzzle HTTP client, which is used to facilitate communication with web services:

```

1  $ composer require guzzlehttp/guzzle

```

Next, return to the About controller's store method and modify it to look like this:

```

1  public function store(ContactFormRequest $request)
2  {
3
4      \Mail::send('emails.contact',
5          [
6              'name'          => $request->get('name'),
7              'email'         => $request->get('email'),
8              'user_message' => $request->get('message')
9          ], function($message)
10     {
11         $message->from(env('MAIL_FROM'));
12         $message->to(env('MAIL_TO'), 'Todoparrot Admin')
13         ->subject('TODOParrot Feedback');

```

```

14     });
15
16     return redirect()->route('contact')
17         ->with('message', 'Thanks for contacting us!');
18
19 }
```

The `Mail::send` method is responsible for initiating delivery of the e-mail. It accepts three parameters. The first parameter defines the name of the view used for the e-mail body template. The second parameter contains an array of data which will be made available to the e-mail template. In this case, the desired data originated in the contact form and is now made available through the `$request` object. The third parameter is a closure that gives you the opportunity to define additional e-mail related options such as the sender, recipient, and subject. Be sure to check out the [Laravel mail documentation](http://laravel.com/docs/master/mail)¹⁰⁷ for a complete explanation of the `Mail::send` method's features.



Notice I used the variable `user_message` to pass the contact form's message text into the view. This is because Laravel always passes a variable named `$message` into the e-mail views which is used for attachment-related matters, so you should take care to not override this variable otherwise seemingly mysterious errors will crop up.

Finally, you'll need to create the contact view which contains the email content. I suggest saving this file in `resources/views/emails`. Per the above example you'll need to name the file `contact.blade.php`. For the purposes of this example I created a very simple view that looks like this:

```

1  You received a message from TODOParrot.com:
2
3  <p>
4  Name: {{ $name }}
5  </p>
6
7  <p>
8  {{ $email }}
9  </p>
10
11 <p>
12 {{ $user_message }}
13 </p>
```

¹⁰⁷<http://laravel.com/docs/master/mail>

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](#)¹⁰⁸ for more details.

After saving these changes, return to the contact form, submit valid data and an e-mail should soon arrive in the inbox associated with the e-mail address supplied via the `to` method!



If you neglected my advice against using Gmail and are, experiencing problems, it could be because of a Gmail setting pertaining to third-party access to your account. Enable the “Less secure apps” setting at <https://www.google.com/settings/security/lesssecureapps>¹⁰⁹ to resolve the issue. Even if this resolves the issue, keep in mind you nonetheless should not use your Gmail account for production purposes.

Creating New 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. If you haven’t already done so based on examples found in earlier chapters, begin by creating a RESTful controller:

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

With the controller created we next need to inform Laravel that we’d like to identify the controller routes as RESTful. Open `routes/web.php` and add the following line:

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

Save the changes to `routes/web.php` and then open the newly created `Lists` controller. You’ll find seven actions (each representing one of the RESTful routes introduced in chapter 3). For easy reference I’ve pasted in the newly created controller, leaving only the two actions (create and show) we’ll use to add a new list:

¹⁰⁸<http://laravel.com/docs/master/mail>

¹⁰⁹<https://www.google.com/settings/security/lesssecureapps>

```

1 namespace App\Http\Controllers;
2
3 class ListsController extends Controller {
4
5     public function create()
6     {
7     }
8
9     public function store()
10    {
11    }
12
13 }

```

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



BTW, don't actually delete the other actions because we'll use them later in the chapter.

With the controller created and routes defined, it's time to create the form.

Creating the TODO List Form

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

```

1 @extends('layouts.app')
2
3 @section('content')
4
5 <h1>Create a New List</h1>
6
7 <ul>
8     @foreach($errors->all() as $error)
9         <li>{{ $error }}</li>
10    @endforeach
11 </ul>
12
13 {!! Form::open(['route' => 'lists.store', 'class' => 'form']) !!}

```

```
14
15 <div class="form-group">
16     {!! Form::label('note', 'List Name') !!}
17     {!! Form::text('name', null,
18         ['class' => 'form-control',
19         'placeholder' => 'San Juan Vacation']) !!}
20 </div>
21
22 <div class="form-group">
23     {!! Form::label('note', 'List Description') !!}
24     {!! Form::textarea('note', null,
25         ['class' => 'form-control',
26         'placeholder' => 'Things to do before leaving for vacation']) !!}
27 </div>
28
29 <div class="form-group">
30     {!! Form::submit('Create List', ['class' => 'btn btn-primary']) !!}
31 </div>
32 {!! Form::close() !!}
33
34 @endsection
```

Presuming you’ve reviewed the earlier section regarding the contact form, then most of the form syntax is familiar to you. After creating the form, you’ll need to modify the `List` controller’s `create` action to serve the view:

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

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

Create a New List

List Name

San Juan Vacation

List Description

Things to do before leaving for vacation

Create List

Creating a new TODO List

With the form in place and the create action updated, it's time to create the Form Request class used to validate the submitted form data.

Creating the List Form Request Class

In this section we'll create a form request that will be used to validate the form data. Begin by creating the form request class skeleton:

- 1 `$ php artisan make:request ListFormRequest`
- 2 `Request created successfully.`

Open the newly created form request class (`app/Http/Requests/ListFormRequest.php`) and you should see the following contents:

```
1 namespace App\Http\Requests;
2
3 use Illuminate\Foundation\Http\FormRequest;
4
5 class ListFormRequest extends FormRequest
6 {
7
8     public function authorize()
9     {
10         return false;
11     }
12
13     public function rules()
14     {
15         return [
16             //
17         ];
18     }
19 }
```

As a reminder, the `rules` method is used to define the validation rules which will be used in conjunction with the form fields. The list name and description are both logically required, so modify the method to look like this:

```
1 public function rules()
2 {
3     return [
4         'name' => 'required',
5         'note' => 'required'
6     ];
7 }
```

You'll also want to modify the `authorize` method to return `true` instead of `false`, because at this point in time we're going to allow anybody to use the form (I'll show you how to restrict access in chapter 7):

```
1 public function authorize()
2 {
3     return true;
4 }
```

Updating the List Controller's Store Action

With the other pieces of the puzzle in place, all that remains is to update the `List` controller's `store` action to process the form contents. Of course, `ListFormRequest` handles the tiresome matter of validation, leaving us to focus solely on what to do with the data should it pass muster. In this instance all we need to do is save the data to the database, as demonstrated in the below revised `store` method:

```
1 use App\ToDoList;
2 use App\Http\Requests\ListFormRequest;
3
4 ...
5
6 public function store(ListFormRequest $request)
7 {
8
9     $list = new ToDoList();
10    $list->name = $request->get('name');
11    $list->note = $request->get('note');
12
13    $list->save();
14
15    return \Redirect::route('lists.create')
16        ->with('message', 'Your list has been created!');
17
18 }
```

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 `List` controller and you'll see these two action method skeletons are already in place:

```
1  public function edit($id)
2  {
3  }
4
5  public function update($id)
6  {
7  }
```

The `edit` action is responsible for serving the form (which is filled in with the existing list's data), and the `store` action is responsible for saving the updated form contents to the database. Notice how both actions accept as input an `$id`. This is the primary key of the list targeted for modification. If you recall from Chapter 3's REST-related discussion, these two actions are accessed via (in the case of the `List`) controller `GET /lists/:id/edit` and `PUT /lists/:id`, respectively. If the `PUT` method is new to you, not to worry because Laravel handles all of the details associated with processing `PUT` requests, meaning all you have to do is construct the form and point it to the `update` route. Let's take care of this next.

Creating the TODO List Update Form and Edit Action

The form used to update a record is in most cases practically identical to that used to create a new record, with one very important difference; instead of `Form::open` you'll use `Form::model`:

```
1  {!! Form::model($list,
2  [
3      'route' => ['lists.update', $list->id],
4      'class' => 'form'
5  ]) !!}
6
7  ...
8
9  {!! Form::close() !!}
```

The `Form::model` method *binds* the enclosed form fields to the contents of a model record. Additionally, be sure to take note of how the list ID is passed into the `lists.update` route. This record is passed into the view like you would any other:

```

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

```

When you pass a `Todolist` record into the `Form::model` method, it will bind the values of any attributes to form fields with a matching name. Let's create the entire form, however before doing so you'll need to create a new view named `edit.blade.php` and place it in the `resources/views/lists` directory. Then place the following contents into this view:

```

1 {!! Form::model($list,
2     [
3         'method' => 'put',
4         'route' => ['lists.update', $list->id],
5         'class' => 'form'
6     ]
7 ) !!}
8
9 <div class="form-group">
10     {!! Form::label('name', 'List Name') !!}
11     {!! Form::text('name', null,
12         ['class' => 'form-control',
13             'placeholder' => 'San Juan Vacation']) !!}
14 </div>
15
16 <div class="form-group">
17     {!! Form::label('note', 'List Description') !!}
18     {!! Form::textarea('note', null,
19         ['class' => 'form-control',
20             'placeholder' => 'Things to do before leaving for vacation']) !!}
21 </div>
22
23 <div class="form-group">
24     {!! Form::submit('Update List', ['class' => 'btn btn-primary']) !!}
25 </div>
26 {!! 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://dev.todoparrot.com/lists/2/edit>) and you should see a populated form!



In cases where the form used to create and edit a record are identical in every fashion except for the use of `Form::open` and `Form::model`, consider storing the form fields in a partial view and then inserting that partial into the create and edit views between the form opener and `Form::close` method.

Updating the List Controller's Update Action

With the edit action and corresponding view in place all that remains is to update the update action. In most cases you'll be able to simply reuse the form request helper created for use in conjunction with the create action, and in this case we'll go ahead and do so:

```
1  public function update($id, ListFormRequest $request)
2  {
3
4      $list = TodoList::findOrFail($id);
5
6      $list->name = $request->get('name');
7      $list->note = $request->get('note');
8
9      $list->save();
10
11     return \Redirect::route('lists.edit',
12         [$list->id])->with('message', 'Your list has been updated!');
13
14 }
```

Make sure you update the input parameters passed into update to include the `ListFormRequest` request object. Once saved you should be able to edit existing lists!

Deleting TODO Lists

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, as indicated when running `route:list`:

```

1 $ php artisan route:list
2
3 ...+-----+-----+-----+-----+-----+
4 ...| Method | URI           | Name           | Action           |
5 ...+-----+-----+-----+-----+-----+
6 ...| DELETE | lists/{lists} | lists.destroy | ...ListController@destroy |
7 ...+-----+-----+-----+-----+-----+

```

This means you can't just create a hyperlink pointing users to the `lists.destroy`, because hyperlinks by default use the GET method. Instead you'll use a form with a stylized button to create the appropriate link, as demonstrated below:

```

1 {!! Form::open(
2     [
3         'route' => ['lists.destroy', $list->id],
4         'method' => 'delete'
5     ]) !!}
6     <button type="submit">Delete List</button>
7 {!! Form::close() !!}

```

Notice how the `Form::open` method's `method` attribute is overridden (the default is POST) to instead use DELETE. The form's `route` attribute identifies the `lists.destroy` route as the submission destination, passing in the list ID. When submitted, the user will be taken to the `Lists` controller's `destroy` action, which looks like this:

```

1 public function destroy($id)
2 {
3
4     TodoList::destroy($id);
5
6     return \Redirect::route('lists.index')
7         ->with('message', 'The list has been deleted!');
8
9 }

```

Associating Tasks with Categories

As you learned in Chapter 4 it's easy to programmatically associate categories with a list using a many-to-many relationship. To recap, you can associate a new task with an existing list within your project controller like so:

```
1 $list = TodoList::find(1);
2
3 $task = new Task;
4
5 $task->name = 'Walk the dog';
6 $task->description = 'Take Barky the Mutt for a walk';
7
8 $list->save();
9
10 // Associate categories 3 and 4 with this list
11 $list->categories()->attach([3,4]);
```

But how might you go about effectively integrating this feature into a web form, as depicted in the below screenshot? The answer is easier than you think. As a bonus I'll introduce two very useful features you'll likely use repeatedly when building Laravel-driven forms.

Create a New List

List Name

Gym Workout

List Description

Exercises at today's gym session.

Categories

Categories

Leisure
Exercise
Work
Home Remodeling
Landscaping

Create List

Creating a Nested List

To demonstrate how you might implement this feature, I'll revise the form used in the earlier section, "Creating the TODO List Form", adding a few additional fields for inputting several starter tasks:

```

1  {!! Form::open(['route' => 'lists.store', 'class' => 'form']) !!}
2
3  <div class="form-group">
4      {!! Form::label('List Name') !!}
5      {!! Form::text('name', null,
6          ['class' => 'form-control',
7              'placeholder' => 'e.g. San Juan Vacation']) !!}
8  </div>
9
10 <div class="form-group">
11     {!! Form::label('List Description') !!}
12     {!! Form::textarea('note', null,
```

```

13      ['class' => 'form-control',
14      'placeholder' => 'Things to do before leaving for vacation']) !!}
15  </div>
16
17  <h3>Categories</h3>
18
19  <div class="form-group">
20      {!! Form::label('Categories') !!}
21      {!! Form::select('categories', $categories, null,
22      ['multiple' => 'multiple',
23      'name' => 'categories[]'
24      ]) !!}
25  </div>
26
27  <div class="form-group">
28      {!! Form::submit('Create List', ['class' => 'btn btn-primary']) !!}
29  </div>
30  {!! Form::close() !!}

```

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

```

1  {!! Form::select('categories[]', $categories, null,
2  ['multiple'=>'multiple']) !!}

```

The `Form::select` method accepts four parameters. The first identifies the name of the field. The second identifies array used to populate the select field's id and name values for each option. The third field, which in this example is set to `null`, identifies any options (by ID) that should be selected by default. The fourth field identifies any HTML attributes which should be set. In this example we're ensuring the user can select multiple values. When rendered to the browser using the above code the multiple select box will look like this:

```

1  <select name="categories[]" multiple="multiple">
2      <option value="1">Leisure</option>
3      <option value="2">Exercise</option>
4      <option value="3">Work</option>
5      <option value="4">Home Remodeling</option>
6      <option value="5">Landscaping</option>
7  </select>

```

Next, you'll need to modify the `Lists` controller's `create` method to retrieve the list of categories used to populate the select field. Because you only want the `categories` table's `id` and `name` columns, you can use a convenient helper named `lists` which will create an array from the retrieved data, using the provided two columns for the array values and IDs. Here's an example executed within Tinker:

```

1  [1] > $c = App\Category::lists('name', 'id');
2  // array(
3  //      1 => 'Leisure',
4  //      2 => 'Exercise',
5  //      3 => 'Work',
6  //      4 => 'Home Remodeling',
7  //      5 => 'Landscaping'
8  // )

```



Laravel 5.1 Update Alert

As of Laravel 5.1 the `lists` method's behavior has changed in the sense that it returns a collection instead of an array. You can however ensure `lists` continues to return an array by chaining the `all` method, such as `App\Category::lists('name', 'id')->all()`.

Admittedly I find it weird you identify the column used for the array value before that used for the index, but in any case the method works great provided you keep this in mind, so all you'll need to do is retrieve the desired data using the `lists` method and pass it into the view:

```

1  public function create()
2  {
3      $categories = Category::lists('name', 'id');
4      return view('lists.create')->with('categories', $categories);
5  }

```

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

```

1  public function store(ListFormRequest $request)
2  {
3
4      $list = new TodoList();
5      $list->name = $request->get('name');
6      $list->note = $request->get('note');
7
8      $list->save();
9
10     if (count($request->get('categories')) > 0) {
11         $list->categories()->attach($request->get('categories'));
12     }

```

```

13
14     return redirect()->route('lists.create')
15         ->with('message', 'Your list has been created!');
16
17 }

```

Note how we first check the `categories` field to confirm it contains at least one category; if so the `attach` method is used to associate the selected categories with the newly created list. Of course, if the user is required to choose at least one category then consider encapsulating this validation within the associated form helper.

Uploading Files

I'm currently working on a Laravel 5 application which includes a restricted administration console used to manage products sold through an online catalog. Each product includes a name, SKU, price, description, and image. The image is uploaded using the `Form::file` helper made available through the [LaravelCollective/html](https://github.com/LaravelCollective/html)¹¹⁰ package, validated alongside the other form inputs using a Laravel 5 form request, and if valid, stored in a special directory. In this section I'll show you how to integrate a similar form and upload capabilities into your own application.

Let's begin with a simplified version of the form used in the project. Again, this uses the [LaravelCollective/html](https://github.com/LaravelCollective/html)¹¹¹ package's form helpers to generate the various form fields:

```

1  {!! Form::open(
2      [
3          'route'      => 'admin.products.store',
4          'class'      => 'form',
5          'novalidate' => 'novalidate',
6          'files'      => true]) !!}
7
8  <div class="form-group">
9      {!! Form::label('Product Name') !!}
10     {!! Form::text('name', null,
11         ['placeholder' => 'Chess Board']) !!}
12 </div>
13
14 <div class="form-group">
15     {!! Form::label('Product SKU') !!}
16     {!! Form::text('sku', null, ['placeholder' => '1234']) !!}
17 </div>

```

¹¹⁰<https://github.com/LaravelCollective/html>

¹¹¹<https://github.com/LaravelCollective/html>

```
18
19 <div class="form-group">
20     {!! Form::label('Product Image') !!}
21     {!! Form::file('image', null) !!}
22 </div>
23
24 <div class="form-group">
25     {!! Form::submit('Create Product!') !!}
26 </div>
27 {!! Form::close() !!}
28 </div>
```

Specific to the matter of file uploading there are two key characteristics of this form you'll need to keep in mind when implementing your own uploader:

- The `Form::open` method sets the `'files' => true` attribute. This results in the form data being encoded as `multipart/form-data`, which is required when files will be included as form data.
- The `Form::file` helper is used to generate the file upload control.

When rendered to the browser the form looks like this:

Create a New Product

Product Name

Product SKU

Product Image

 No file chosen

The file upload form

As you can see, the form is submitted to a route named `admin.products.store`. As is typical of any Laravel 5 application, the submitted form data is first routed through a form request. The validation rules are found in the form request's `rules()` method. Here's an example which validates the supplied image to ensure one is present and that it is specifically a PNG (image file):

```
1 public function rules()  
2 {  
3     return [  
4         'name' => 'required',  
5         'sku'  => 'required|unique:products,sku,' . $this->get('id'),  
6         'image' => 'required|mimes:png'  
7     ];  
8 }
```

You can validate uploads using plenty of other approaches such as ensuring it is a Word document or PDF. See the [Laravel documentation](http://laravel.com/docs/master/validation)¹¹² for more information. This request is passed into the `Admin/ProductController.php`'s `store` method, which looks like this:

¹¹²<http://laravel.com/docs/master/validation>

```
1 public function store(ProductRequest $request)
2 {
3
4     $product = new Product();
5     $product->name = $request->get('name');
6     $product->sku = $request->get('sku');
7
8     $product->save();
9
10    $imageName = $product->id . '.' .
11        $request->file('image')->getClientOriginalExtension();
12
13    $request->file('image')->move(
14        base_path() . '/public/images/catalog/', $imageName
15    );
16
17    return redirect()->route('admin.products.edit',
18        [$product->id])->with('message', 'Product added!');
19
20 }
```

In this action we first save the product, and then process the image. There are *plenty* of different approaches to processing the uploaded image; I'm keeping this simple and just saving the image using a name matching the product ID, so for instance if the saved product using the ID 42 then the associated uploaded image will be named 42.png. The image name is first created (and stored in `$imageName`) and then it is moved into the application's `/public/images/catalog` directory.

Believe it or not, uploading an image using Laravel 5 is really that simple!

Summary

This was one of the more entertaining chapters to write because it really illustrates how you can begin introducing interactive features into your Laravel application. Stay tuned as in forthcoming revisions I'll continue to expand this chapter and demonstrate more complicated form features!

Chapter 6. Introducing Middleware

In a nutshell, *middleware* is code that can be configured to 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 added 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 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 four ready-made middleware solutions, including:

- `Authenticate.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".
- `EncryptCookies.php`: Laravel cookies are encrypted by default, however you can use this middleware to identify the names of cookies which shouldn't be encrypted. This may be useful a third-party technology such as AngularJS or Vue.js needs to read a cookie set by Laravel.
- `RedirectIfAuthenticated.php`: This middleware is used to confirm a user is *not* signed into the application. If so, the user is redirected to the home page. See chapter 7 for more information about this middleware.
- `VerifyCsrfToken.php`: This middleware is used to manage CSRF protection. As of Laravel 5.1 you can disable CSRF protection on a per-URI basis by updating this middleware's `$except` array, or by altogether removing the middleware from the `app/Http/Kernel.php`'s `$middleware` array, and then selectively enabling it on a per-route basis (I'll show you how to do the former later in the chapter, and generally speaking you should never do the latter).

The `VerifyCsrfToken` middleware is automatically enabled for every route falling within the web group, however `Authenticated` and `RedirectIfAuthenticated` are intended to be selectively applied according to specific route requests. You'll find these latter two *route-level middlewares* registered in `app/Http/Kernel.php`, in addition to two other middleware useful for implementing basic access authentication (`AuthenticateWithBasicAuth`) and throttling API requests (`ThrottleRequests`):


```

1  protected $routeMiddleware = [
2      'auth' => \App\Http\Middleware\Authenticate::class,
3      'auth.basic' =>
4          \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
5      'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
6      'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
7  ];

```

In chapter 7 I'll talk more about the authentication-related middleware, although you'll learn more about their general operation in the later section, "How Route-Level Middleware Works".

There are also several group-level middlewares, which you'll find defined in `app/Http/Kernel.php`'s `$middlewareGroups` array:

```

1  protected $middlewareGroups = [
2      'web' => [
3          \App\Http\Middleware\EncryptCookies::class,
4          \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
5          \Illuminate\Session\Middleware\StartSession::class,
6          \Illuminate\View\Middleware\ShareErrorsFromSession::class,
7          \App\Http\Middleware\VerifyCsrfToken::class,
8      ],
9
10     'api' => [
11         'throttle:60,1',
12     ],
13 ];

```

The `EncryptCookies`, `AddQueuedCookiesToResponse`, `StartSession`, and `ShareErrorsFromSession` middlewares are used by Laravel to manage various session-related features and are therefore defined within the web group. `VerifyCsrfToken` is also defined as web group-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.

The `api` group contains a lone middleware reference to `throttle`. The `throttle` alias refers to the `ThrottleRequests` middleware defined in the `$routeMiddleware` array. This would be useful if you were creating a Laravel-based API and wanted to restrict the number of requests an account could make over a given period of time.

Finally, you'll find the `CheckForMaintenanceMode` middleware defined in the `$middleware` array. This array identifies application-level middleware, `CheckForMaintenanceMode` is found here 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. I'll show you how this feature works in chapter 8.

How Application and Group-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. Group-level middleware works similarly, but is applied to a specific route grouping. In this section you'll learn more about how application and group-level middleware works by examining the group-level `VerifyCsrfToken` middleware internals. The `VerifyCsrfToken.php` file is located here:

```
1 vendor/laravel/framework/src/Illuminate/  
2 Foundation/Http/Middleware/VerifyCsrfToken.php
```

If you open it you'll find the following class:

```
1 <?php  
2  
3 namespace Illuminate\Foundation\Http\Middleware;  
4  
5 use Closure;  
6 use Illuminate\Foundation\Application;  
7 use Symfony\Component\HttpFoundation\Cookie;  
8 use Illuminate\Contracts\Encryption\Encrypter;  
9 use Illuminate\Session\TokenMismatchException;  
10  
11 class VerifyCsrfToken  
12 {  
13  
14     protected $app;  
15  
16     protected $encrypter;  
17  
18     protected $except = [];  
19  
20     public function __construct(Application $app, Encrypter $encrypter)  
21     {  
22         $this->app = $app;  
23         $this->encrypter = $encrypter;  
24     }  
25  
26     public function handle($request, Closure $next)  
27     {  
28         if (
```

```
29         $this->isReading($request) ||
30         $this->runningUnitTests() ||
31         $this->shouldPassThrough($request) ||
32         $this->tokensMatch($request)
33     ) {
34         return $this->addCookieToResponse($request, $next($request));
35     }
36
37     throw new TokenMismatchException;
38 }
39
40 protected function shouldPassThrough($request)
41 {
42     foreach ($this->except as $except) {
43         if ($except !== '/') {
44             $except = trim($except, '/');
45         }
46
47         if ($request->is($except)) {
48             return true;
49         }
50     }
51
52     return false;
53 }
54
55 protected function runningUnitTests()
56 {
57     return $this->app->runningInConsole() &&
58         $this->app->runningUnitTests();
59 }
60
61 protected function tokensMatch($request)
62 {
63     $sessionToken = $request->session()->token();
64
65     $token = $request->input('_token') ?:
66         $request->header('X-CSRF-TOKEN');
67
68     if (! $token && $header = $request->header('X-XSRF-TOKEN')) {
69         $token = $this->encrypter->decrypt($header);
70     }
```

```

71
72     if (! is_string($sessionToken) || ! is_string($token)) {
73         return false;
74     }
75
76     return hash_equals($sessionToken, $token);
77 }
78
79 protected function addCookieToResponse($request, $response)
80 {
81     $config = config('session');
82
83     $response->headers->setCookie(
84         new Cookie(
85             'XSRF-TOKEN', $request->session()->token(),
86             time() + 60 * 120,
87             $config['path'], $config['domain'],
88             $config['secure'], false
89         )
90     );
91
92     return $response;
93 }
94
95 protected function isReading($request)
96 {
97     return in_array($request->method(), ['HEAD', 'GET', 'OPTIONS']);
98 }
99 }

```

All middleware implements a method named `handle`, which is responsible for processing the incoming request if the middleware implementation’s parameters for doing so are met. In the case of `VerifyCsrfToken`, the HTTP method used for the request must include HEAD, GET, or OPTIONS (as defined by the `isReading` method) and the `_token` 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 X-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.

In summary, while you’re free to define as many helper methods as desired in your middleware class, it’s crucial to implement the `handle` method, passing in the `$request` and `$next` parameters (representing the current request and the “next” step in the response pipeline). If your middleware

passes any defined conditions, the `$next` parameter should be returned; otherwise an exception should be thrown.

We'll implement a custom middleware in the later section, "Creating Your Own Middleware Solution", and still another for restricting access to an administrative console in chapter 9.

How Route-Level Middleware Works

Route-level middleware works identically to application-level middleware, except you can configure it to execute selectively in conjunction with a specific route or set of routes. For example if you look at the default Auth controller (`app/Http/Controllers/Auth/AuthController.php`) you'll see the guest middleware is invoked in the class constructor:

```
1 public function __construct()  
2 {  
3     $this->middleware('guestMiddleware()', ['except' => 'getLogout']);  
4 }
```

If you have a look at `app/Http/Kernel.php` you'll see the guest middleware is actually an abbreviated alias for the `RedirectIfAuthenticated` middleware. As defined in the above constructor, `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.php`) you'll see the `handle` method is really simple to understand in that it uses Laravel's built-in authentication capabilities to determine whether the user is already signed in (via the `check` method, which is introduced in the next chapter).

Creating Your Own Middleware Solution

As an exercise let's create a simple route-level middleware solution that sends a message to the Laravel log when invoked. Begin by creating a new middleware skeleton using Artisan's `make:middleware` command:

```
1 $ php artisan make:middleware RequestLogger  
2 Middleware created successfully.
```

This command created a new middleware class skeleton named `RequestLogger` that resides in the directory `app/Http/Middleware`. The class skeleton looks like this:

```
1 <?php
2
3 namespace App\Http\Middleware;
4
5 use Closure;
6
7 class RequestLogger
8 {
9
10     public function handle($request, Closure $next)
11     {
12         return $next($request);
13     }
14 }
```

Modify the `handle` method to log the visitor's IP address to the Laravel log, and then pass on the request:

```
1 public function handle($request, Closure $next)
2 {
3     \Log::info($request->ip());
4     return $next($request);
5 }
```

Next, open up `app/Http/Kernel.php` and register the new `RequestLogger` middleware:

```
1 protected $routeMiddleware = [
2     'auth' => \App\Http\Middleware\Authenticate::class,
3     'auth.basic' =>
4         \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
5     'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
6     'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
7     'iplogger' => \App\Http\Middleware\RequestLogger::class,
8 ];
```

Finally, just reference the `iplogger` alias whenever you'd like to execute the middleware. Just for the sake of demonstration I'll place the middleware call in the `Welcome` controller's constructor:

```
1 class HomeController extends Controller {
2
3     public function __construct()
4     {
5         $this->middleware('iplogger');
6     }
7
8     ...
9
10 }
```

After referencing the new middleware, reload an endpoint associated with the reference and check your log (storage/logs). If you are working from your local laptop you'll see a line that looks like this:

```
1 [2016-04-20 12:44:08] 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.

It is worth noting an important distinction regarding middleware behavior; 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:

```
1 public function handle($request, Closure $next)
2 {
3     $response = $next($request);
4     \Log::info($request->ip());
5     return $response;
6 }
```

Using Middleware Parameters

New to Laravel 5.1 is support for middleware parameters. This is a long-awaited and tremendously useful feature. To illustrate its practicality, suppose you wanted to create a middleware (we'll call it `ProMiddleware` and make it available via the `pro` alias) which granted only those users having accumulated a particular number of forum points access to a particular set of controllers.

```
1 public function handle($request, Closure $next, $points)
2 {
3     if (! $request->user()->totalPoints() < $points) {
4         return \Redirect::route('welcome');
5     }
6
7     return $next($request);
8 }
```

When defining the middleware-restricted route, you'll identify the middleware name and the parameter like so:

```
1 Route::group(['middleware' => 'pro:10000'], function()
2 {
3     Route::resource('pro', 'ProController');
4 });
```

Summary

Middleware is one of those seemingly mundane features that can dramatically streamline your application code, and thanks to the flexibility Laravel provides in integrating middleware into your code, the sky is the limit in terms of how you can control your project's request/response pipeline!

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 interaction through customized features. However, there are numerous matters one has to take into consideration when integrating account-related features into an application, including user registration, secure storage of user credentials, user authentication, lost password recovery, profile management, and general integration of tailored features.

Fortunately, Laravel 5 removes numerous headaches associated with implementing many of these aforementioned features for you! In this chapter I'll show you how to configure and incorporate these bundled authentication features into your own application. You'll also learn how to enable Laravel's new authentication throttling feature, a useful tool for helping keep the bad guys out.

Registering Users

Implementing the user registration feature seems to be a logical starting point. 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 and Users Table

You're spared the hassle of building a model and underlying table for managing user accounts, because all new Laravel applications include a `User` model intended expressly for this purpose. You'll find it in `app/User.php`, and it looks like this:

```
1  <?php
2
3  namespace App;
4
5  use Illuminate\Foundation\Auth\User as Authenticatable;
6
7  class User extends Authenticatable
8  {
9
10     protected $fillable = [
11         'name', 'email', 'password',
```

```
12     ];  
13  
14     protected $hidden = [  
15         'password', 'remember_token',  
16     ];  
17  
18 }
```

This `User` model looks rather different from the models we’ve created in earlier chapters, notably because it extends the `User` (you’ll see at the top of the file `Illuminate\Foundation\Auth\User` is aliased as `Authenticatable`) class rather than `Model`. This `User` class in turn implements several *contracts* and uses several *traits*. A contract defines an interface to a particular implementation of a set of features. For instance, `AuthenticatableContract` defines an interface for obtaining the user’s unique identifier and password and for managing the “remember me” token should it be enabled. Because the interface ensures the functionality is loosely coupled, you’re free to easily swap out the Laravel implementation for another.

The contracts work in unison with the *traits*. As you can see, the `User` model uses three traits, including `Authenticatable`, `Authorizable`, 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. This means you could for instance replace the implemented traits with your own, provided you meet the requirements defined in the contract. In fact, recent versions of Laravel have taken full advantage of these opportunities for extensibility, allowing developers to easily implement their own authentication, authorization, and password recovery features.



Philip Brown penned a [great introductory tutorial](http://cultttt.com/2014/06/25/php-traits/)¹¹³ to traits. It’s worth taking the time to read now if this concept is new to you.

As a reminder from topics introduced in chapter 3, 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 the `User` model, you’ll also find a corresponding `users` table migration, located in `database/migrations/2014_10_12_000000_create_users_table.php`. Because by this point in the book you likely already ran `artisan migrate`, the `users` table already exists in your project database. The table looks like this:

¹¹³<http://cultttt.com/2014/06/25/php-traits/>

```

1  mysql> describe users;
2  +-----+-----+-----+-----+...
3  | Field          | Type                | Null | Key | ...
4  +-----+-----+-----+-----+...
5  | id             | int(10) unsigned   | NO   | PRI | ...
6  | name          | varchar(255)       | NO   |     | ...
7  | email         | varchar(255)       | NO   | UNI | ...
8  | password      | varchar(60)        | NO   |     | ...
9  | remember_token | varchar(100)       | YES  |     | ...
10 | created_at    | timestamp          | NO   |     | ...
11 | updated_at    | timestamp          | NO   |     | ...
12 +-----+-----+-----+-----+...
13 7 rows in set (0.01 sec)

```

With the User model and users table in place, we'll next need to sort out how to create a registration form and wire it up to some registration logic.

Generating the Authentication Scaffolding

Despite being perfectly capable of carrying out the various account registration, sign in, sign out, and password recovery tasks, your Laravel application doesn't by default include the associated user-facing forms and routes. This is because the Laravel developers don't necessarily want to clutter up an application skeleton with views and features that aren't going to be used by a developer creating an API or a site not requiring user authentication. However, you can easily generate these endpoints, views, and forms by executing Artisan's `make:auth` command. Before doing so however, understand that if you've already started work on a home page using the default `welcome.blade.php` view, or have created a layout called `app.blade.php` inside the `views/layouts` directory. running this command will *completely* replace any changes you've made to these files! Therefore take care to create a copy of the files before running the command (or preferably just commit your changes to version control and then revert the changes):

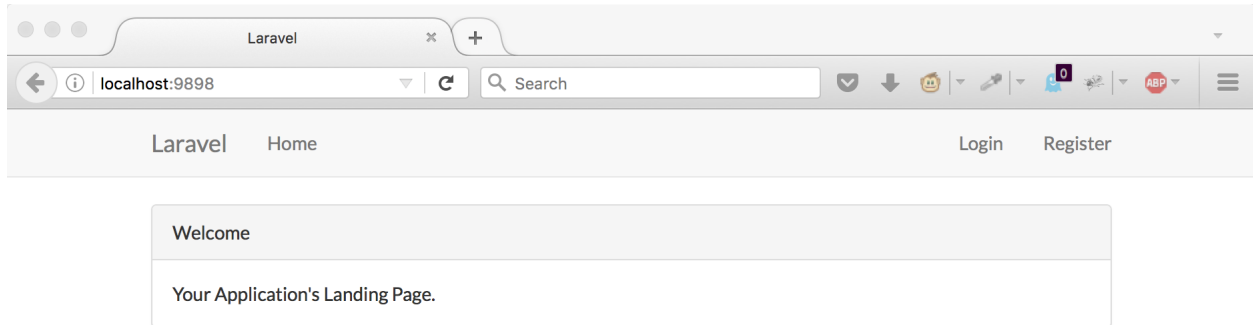
```

1  $ php artisan make:auth
2  Created View: ...parrot.com/resources/views/auth/login.blade.php
3  Created View: ...parrot.com/resources/views/auth/register.blade.php
4  Created View: ...parrot.com/resources/views/auth/passwords/email.blade.php
5  Created View: ...parrot.com/resources/views/auth/passwords/reset.blade.php
6  Created View: ...parrot.com/resources/views/auth/emails/password.blade.php
7  Created View: ...parrot.com/resources/views/layouts/app.blade.php
8  Created View: ...parrot.com/resources/views/home.blade.php
9  Created View: ...parrot.com/resources/views/welcome.blade.php
10 Installed HomeController.
11 Updated Routes File.
12 Authentication scaffolding generated successfully!

```

In addition to creating a slate of new views inside the directory `resources/views/auth`, you'll see the aforementioned `welcome.blade.php` and `layouts/app.blade.php` have been generated. Additionally, you'll find a confusingly named view called `home.blade.php`. We'll return to this latter file in just a bit, so don't worry about it for now.

Presuming `welcome.blade.php` is still responsible for your project's home page, if you head over to the browser and reload the site you'll see a rather different home page than what was presented in chapter 1:



The revamped home page

In addition, open `app/Http/routes.php` and you'll find the following two new route definitions:

```
1 Route::auth();  
2  
3 Route::get('/home', 'HomeController@index');
```

Just as `Route::resource()` is a shortcut for defining the seven RESTful routes, so is `Route::auth()` for defining the following routes:

```
1 // Authentication Routes...
2 $this->get('login', 'Auth\AuthController@showLoginForm');
3 $this->post('login', 'Auth\AuthController@login');
4 $this->get('logout', 'Auth\AuthController@logout');
5
6 // Registration Routes...
7 $this->get('register', 'Auth\AuthController@showRegistrationForm');
8 $this->post('register', 'Auth\AuthController@register');
9
10 // Password Reset Routes...
11 $this->get('password/reset/{token?}', 'Auth\PasswordController@showResetForm');
12 $this->post('password/email', 'Auth\PasswordController@sendResetLinkEmail');
13 $this->post('password/reset', 'Auth\PasswordController@reset');
```

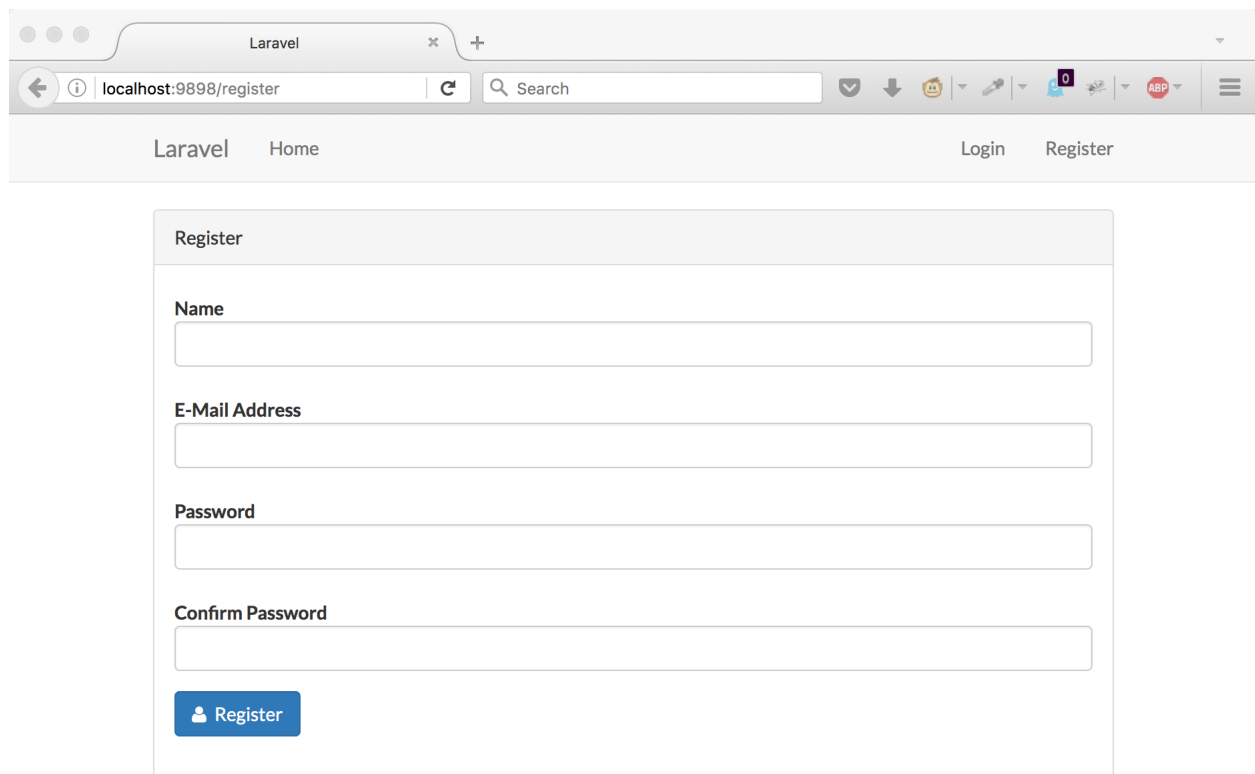
You'll become well-acquainted with these routes and the generated views in the sections to follow.

Introducing the Account Registration Feature

After running `make:auth`, your application's `app/Http/routes.php` file will include a reference to `Route::auth()`, which makes available the following two registration-related routes:

```
1 $this->get('register', 'Auth\AuthController@showRegistrationForm');
2 $this->post('register', 'Auth\AuthController@register');
```

The `get`-based route is responsible for presenting the registration form, while the `post`-based route is responsible for processing the form input. These routes point to `showRegistrationForm()` and `register()` methods found in the `Auth/AuthController.php` file, respectively. However, if you open this controller you won't actually find these methods, because they are made available by way of the `AuthenticatesAndRegistersUsers` trait, which in turn makes them available by way of the `RegistersUsers` trait located in `vendor/laravel/framework/src/Illuminate/Foundation/Auth/RegistersUsers.php`. I realize this probably sounds a tad confusing, but I invite you to have a look at the `RegistersUsers.php` file (where you will in fact find the `showRegistrationForm()` and `register()` methods), and everything will really start to make sense. Regardless of whether you do, rest assured your application can now successfully register users. In fact if you return to the home page and click the `Register` link, you'll be presented with the following form:

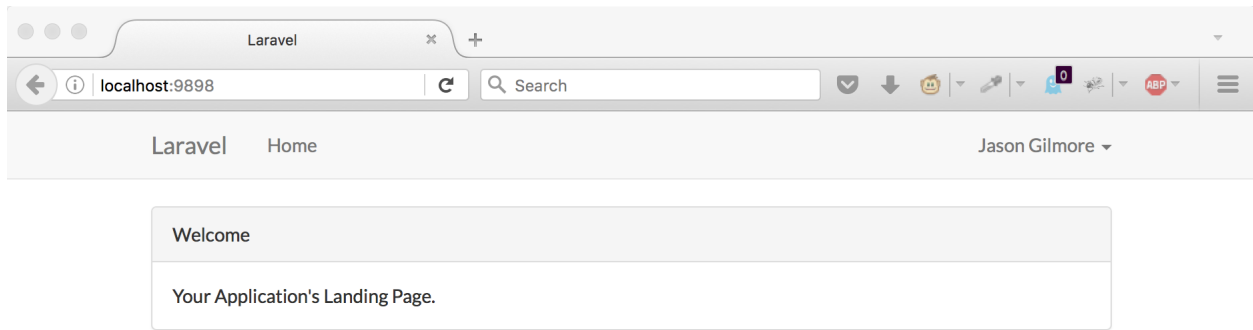


The screenshot shows a web browser window with the address bar displaying 'localhost:9898/register'. The page title is 'Laravel'. The navigation bar includes 'Laravel', 'Home', 'Login', and 'Register' links. The main content area is titled 'Register' and contains a form with the following fields: 'Name', 'E-Mail Address', 'Password', and 'Confirm Password'. Each field is represented by a text input box. At the bottom of the form is a blue button with a user icon and the text 'Register'.

The default registration page

Presuming you configured the project database as described in chapter 3, you can successfully register a test user simply by completing this form. That's right, there is *no additional coding* required to begin accepting user registrations. After creating an account, have a look at the project database's `users` table and you'll see the new record.

Further, Laravel will automatically sign you into the site and attempt to redirect to the `/home` URI. As you can see from the following screenshot, newly registered users are additionally signed in (note my name is displayed at the top right of the page):



Newly registered users are automatically signed in

This destination is managed by the following route, which was added to the `routes.php` file when you executed `make:auth`:

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

Chances are you'll want to change this post-authentication destination (newly authenticated users will similarly be taken here). To do so, you can either point the `/home` route to a different controller and action (and corresponding view), or you can override the destination URI by overriding the `$redirectTo` property within the `AuthController.php` file:

```
1 protected $redirectTo = '/';
```

Introducing the Account Sign In Feature

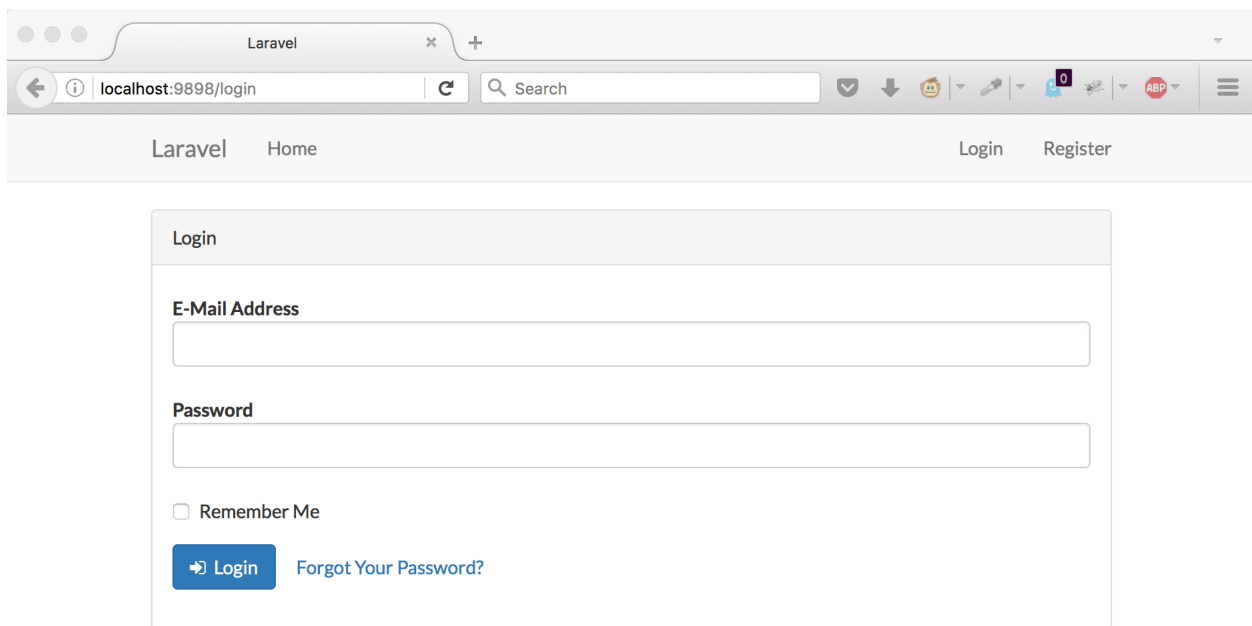
Newly registered users are automatically signed in following registration, however logically those users will want to end their session and sign out. Hopefully they'll return, and so you'll want to provide an easy solution for signing into their account.

When `make:auth` was executed, the following two authentication-related routes will be made available via the `Route::auth()` shortcut:

```
1 $this->get('login', 'Auth\AuthController@showLoginForm');  
2 $this->post('login', 'Auth\AuthController@login');
```

Like the registration routes, you'll see the `get`-based route is responsible for displaying the login form, while the `post`-based route is responsible for processing the form input and determining whether the credentials are valid. Further, while the `AuthController.php`'s `showLoginForm()` and `login()` methods are identified as being responsible for these respective endpoints, you won't actually find the methods in `AuthController.php` because they are instead made available via the `AuthenticatesUsers.php` trait (located in `vendor/laravel/framework/src/Illuminate/Foundation/Auth/`).

You can give authentication a whirl by first logging out of your newly registered account via the `/logout` endpoint, which will end your session and return you to the `welcome.blade.php` view. Next, click on `Login` where you'll be greeted with the form presented in the following screenshot:



The screenshot shows a web browser window with the address bar at `localhost:9898/login`. The page has a navigation bar with "Laravel" and "Home" on the left, and "Login" and "Register" on the right. The main content area displays a "Login" form. The form has a title "Login" and two input fields: "E-Mail Address" and "Password". Below these fields is a checkbox labeled "Remember Me". At the bottom of the form, there is a blue "Login" button and a link labeled "Forgot Your Password?".

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`.

Remember from the earlier registration-related discussion that successfully authenticated users will be redirected to the `/home` URI default. You can either override the default controller used for the `/home` URI in `routes.php`, or override the default redirection URL by adding the following property to the Auth controller (found in `app/Http/Controllers/Auth/AuthController.php`):

```
1 protected $redirectTo = '/';
```

Managing Authentication Throttling

Laravel 5.1 introduced a new feature called *authentication throttling* which will prevent further authentication attempts for one minute if the user attempts to sign in more than five times without success. At the time of this writing both the attempt count (5 attempts) and the delay (60 seconds) were hard coded into the `ThrottlesLogins` trait (located at `vendor/laravel/framework/src/Illuminate/Foundation/Auth/ThrottlesLogins.php`), however perhaps at some point in the future these values will be defined in a configuration file. In any case, authentication throttling is enabled by default, however if you'd like to disable it (not recommended), open your `AuthController` class (`app/Http/Controllers/Auth/AuthController.php`) and remove the `ThrottlesLogins` trait.

```
1 use Illuminate\Foundation\Auth\ThrottlesLogins;
2 ...
3
4 class AuthController extends Controller {
5
6     use AuthenticatesAndRegistersUsers, ThrottlesLogins;
7     ...
8
9 }
```

Signing Out of an Account

All Laravel applications include the ability to end an authenticated session by signing out of an account. The `Route::auth()` shortcut results in the following route being made available to your application:

```
1 $this->get('logout', 'Auth\AuthController@logout');
```

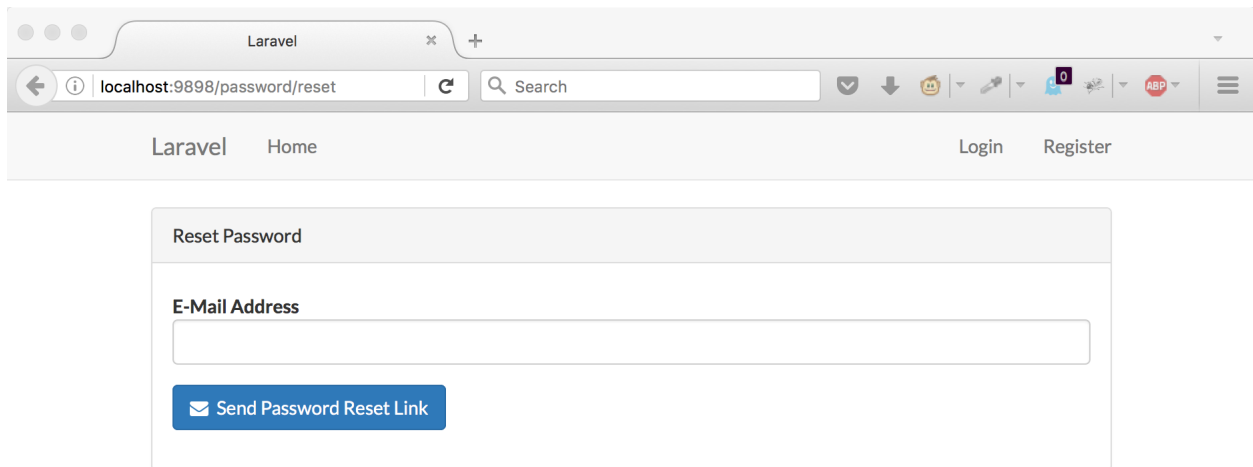
Therefore users can logout of your application via the `/logout` URI, at which point they'll be returned to the root URL (represented by `welcome.blade.php` by default).

Password Recovery

A convenient password recovery is also available after generating the authentication routes and views. The following three routes are made available via the `Route::auth()` shortcut:

```
1 $this->get('password/reset/{token?}', 'Auth\PasswordController@showResetForm');
2 $this->post('password/email', 'Auth\PasswordController@sendResetLinkEmail');
3 $this->post('password/reset', 'Auth\PasswordController@reset');
```

You can see the default recovery form by signing out of your account (via `/logout`) and after being returned to the home page, click `Login` and then the `Forgot Your Password?` link. You'll be taken to the password recovery form presented in the following screenshot:

A screenshot of a web browser showing the Laravel password reset form. The browser's address bar displays 'localhost:9898/password/reset'. The page has a light gray header with 'Laravel' on the left and 'Login' and 'Register' links on the right. The main content area features a white box with a gray header labeled 'Reset Password'. Inside this box, there is a label 'E-Mail Address' above a text input field. Below the input field is a blue button with a white envelope icon and the text 'Send Password Reset Link'.

Initiating password recovery

If you've configured your application's mail driver (see chapter 5), and the e-mail address exists in the users table, then a password recovery e-mail will be sent to the address you specified. The template used to format this e-mail is found in `resources/auth/emails/password.blade.php`. It's default contents look like this:

```

1 Click here to reset your password:
2 <a href="{ $link = url('password/reset', $token).'?email='.urlencode(
3 $user->getEmailForPasswordReset()) }}"> {{ $link }} </a>

```

If you look closely at the syntax you'll see a one-time URL is sent to the user. When the user clicks the link found in the e-mail, Laravel will consult the project database's `password_resets` table (like users, this table will automatically be created when you run migrations for the first time). Here's an example record found in the `password_resets` table:

```

1 mysql> select * from password_resets;
2 +-----+-----+-----+
3 | email          | token          | created_at      |
4 +-----+-----+-----+
5 | wj@wjgilmore.com | asdfasfaasfasdf | 2016-04-21 19:26:02 |
6 +-----+-----+-----+
7 1 row in set (0.00 sec)

```

When the user clicks this link, Laravel will consult `password_resets` for a matching e-mail address and token, and if they match the user will be able to choose a new password. As you can see, the default recovery e-mail is quite sparse, however you can update it to include whatever additional information you please provided the recovery link formatting remains intact.



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. Alternatively, set the `.env` file's `MAIL_DRIVER` setting to `log` and tail the `storage/logs/laravel.log` file to view the mail output.

The recovering user will have 60 minutes to click on the recovery link per the `config/auth.php` file's `passwords['users']['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 authenticated user's name you'll access `Auth::user()` like so:

```

1 Welcome back, {{ Auth::user()->name }}!

```

Of course, you'll want to ensure the user is authenticated before attempting to access the record in this fashion. You can do so by consulting `Auth::check()`:

```

1 @if (Auth::check())
2     Welcome back, {{ Auth::user()->name }}!
3 @else
4     Hello, stranger! <a href="/auth/login">Login</a>
5     or <a href="/auth/register">Register</a>.
6 @endif

```

Conversely, you can flip the conditional around, instead `Auth::guest()` to determine if the user is a guest:

```

1 @if (Auth::guest())
2     <li><a href="/auth/login">Login</a></li>
3     <li><a href="/auth/register">Register</a></li>
4 @else
5     Welcome back, {{ Auth::user()->name }}!
6 @endif

```

Restricting Access to Authenticated Users

As I mentioned in the earlier section, “Introducing the Account Registration Feature”, you can restrict access to a specific controller by referencing the `auth` middleware in the appropriate controller constructor:

```

1 public function __construct()
2 {
3     $this->middleware('auth');
4 }

```

When users attempt to access the restricted controller, Laravel will first check for a valid session. If the session exists, access to the controller will be granted; otherwise the user will be redirected to the sign in view.

If you’d like to restrict access to a group of routes, you can wrap the restricted routes in a `Route::group` method:

```
1 Route::group(['middleware' => 'auth'], function()  
2 {  
3  
4     Route::resource('lists', 'ListsController');  
5     Route::resource('blog', 'BlogController');  
6  
7 });
```

Restricting Forms to Authenticated Users

All generated form request classes automatically include a method named `authorize`. This method is used to determine whether the form is available to all users or to some restricted subset. by default it is set to `false`, and so in chapter 5 we updated the form requests discussed therein to instead return `true` because we had not yet integrated user accounts. To refresh your memory here's what a default `authorize` method looks like inside a newly generated form request:

```
1 <?php  
2  
3 namespace App\Http\Requests;  
4  
5 use App\Http\Requests\Request;  
6  
7 class ContactFormRequest extends Request {  
8  
9     public function authorize()  
10    {  
11        return false;  
12    }  
13  
14    ...  
15  
16 }
```

If you'd like to restrict a form request to authenticated users, you can modify the `authorize()` method to look like this:

```
1 public function authorize()  
2 {  
3     return Auth::check();  
4 }
```

Keep in mind you're free to embed into `authorize()` whatever logic you deem necessary to check a user's credentials. For instance if you wanted to restrict access to not only authenticated users but additionally only those who are paying customers, you can retrieve the user using `Auth::user()` and then traverse whatever associations are in place to determine the user's customer status.

Adding Custom Fields to the Registration Form

The default registration form only includes fields for the user's name, e-mail address, and password. However, what if you wanted to additionally ask for a username or perhaps a location such as the user's country? Fortunately, Laravel's authentication implementation makes this an incredibly easy task. I'll walk you through an example in which we require the user to additionally provide a unique username. Begin by updating the `users` table to include a `username` field:

```
1 $ php artisan make:migration add_username_field_to_users_table
```

Open the newly created migration file and modify the `up` method to look like this:

```
1 Schema::table('users', function(Blueprint $table)
2 {
3     $table->string('username');
4 });
```

Modify the `down` method to look like this:

```
1 Schema::table('users', function(Blueprint $table)
2 {
3     $table->dropColumn('username');
4 });
```

Run the migration and then open the `resources/auth/register.blade.php` form we created earlier in the chapter. Add a field for accepting the username:

```
1 <div class="form-group">
2     <label class="col-md-4 control-label">Username</label>
3     <div class="col-md-6">
4         <input type="text" class="form-control"
5             name="username" value="{{ old('username') }}">
6     </div>
7 </div>
```

Of course, Laravel won't automatically know what to do with this field, therefore you'll need to make one last change. Open the `AuthController.php` controller found in `app/Http/Controllers/Auth`, and modify the `validator` method to look like this:

```
1 protected function validator(array $data)
2 {
3     return Validator::make($data, [
4         'name' => 'required|max:255',
5         'email' => 'required|email|max:255|unique:users',
6         'username' => 'required|unique:users',
7         'password' => 'required|min:6|confirmed'
8     ]);
9 }
```

I've emphasized the line you'll need to add. Next, scroll down and modify the `create` method to look like this:

```
1 protected function create(array $data)
2 {
3     return User::create([
4         'name' => $data['name'],
5         'email' => $data['email'],
6         'username' => $data['username'],
7         'password' => bcrypt($data['password']),
8     ]);
9 }
```

Save these changes, and only one step remains. Open the `app/User.php` file and add the `username` field to the `$fillable` property like so:

```
1 protected $fillable = [  
2     'name', 'email', 'password', 'username'  
3 ];
```

After saving the changes to the User model, believe it or not you're ready to register users with usernames! It really is that easy.

Save these changes, and believe it or not you're ready to register users with usernames! It really is that easy.

Outside of this block you'll define the unrestricted routes, which would include those perhaps telling users more about the application and of course the sign in, sign out, registration, and password recovery routes.

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!

TODO: * make this chapter 8, and move the current chapter 8 heroku deployment section to an appendix. Let's have Heroku, DreamHost, and AWS appendices.

Chapter 8. Creating a Restricted Administration Console

Many applications require a certain level of ongoing monitoring and maintenance beyond the typical code-based improvements. For instance you might wish to add new categories or edit existing names and descriptions, view a comprehensive list of registered users, or keep tabs on content creation and interaction trends. Such tasks should logically be accessible only by project administrators, yet be conveniently accessible. One effective way to integrate these capabilities is via a restricted web-based administration console. In this chapter I'll show you a particularly simple yet effective solution for creating such a console.

Identifying Administrators

There are several third-party packages one can use to add role-based permissions to a Laravel 5 application, including perhaps most notably [Entrust](https://github.com/Zizaco/entrust)¹¹⁴. However if your goal is to simply separate typical users from administrators, then a much more simple solution is available. You'll want to add a new column to the users table named something like `is_admin`. This Boolean column will identify administrators by virtue of being set to `true`. Go ahead and create the migration now:

```
1 $ php artisan make:migration add_is_admin_to_user_table --table=users
2 Created Migration: 2016_04_07_130041_add_is_admin_to_user_table
```

Next, open the newly created migration file and modify the up and down methods to look like this:

```
1 public function up()
2 {
3     Schema::table('users', function(Blueprint $table)
4     {
5         $table->boolean('is_admin')->default(false);
6     });
7 }
8
9 public function down()
10 {
11     Schema::table('users', function(Blueprint $table)
```

¹¹⁴<https://github.com/Zizaco/entrust>

```

12     {
13         $table->dropColumn('is_admin');
14     });
15 }

```

After saving these changes, run the migration:

```

1 $ php artisan migrate
2 Migrated: 2016_04_07_130041_add_is_admin_to_user_table

```

After running the migration, all existing users will have their `is_admin` column set to `false` (the default as defined in the migration). Therefore to identify one or more users as administrators you'll need to login to your database and set those users' `is_admin` columns to `true`. For instance if you're using the `mysql` client you can login to the client and run the following command:

```

1 mysql> update users set is_admin = true where email = 'wj@wjgilmore.com';

```

Creating the Administration Controllers

Next we'll create an administration controller. In reality you'll likely wind up with several controllers which are collectively identified as being administrative in nature, so you can create a convenient *route grouping* which places them all under a *route prefix* and namespace. Create your first such controller by executing the following command:

```

1 $ php artisan make:controller admin/UserController
2 Controller created successfully.

```

This `make:controller` command is a bit different from the others you've executed so far throughout the book because we are *prefixing* the controller name with a directory name. In doing so, a directory named `admin` was created inside `app/Http/Controllers`, and inside `admin` you'll find the `UserController.php` directory. We'll use this controller to list and manage registered users. Next let's create the route grouping which identifies both the URI prefix and the namespace:

```

1 Route::group(['prefix' => 'admin', 'namespace' => 'admin'], function()
2 {
3     Route::resource('user', 'UserController');
4 });

```

Note the user of `Route::group`. This allows you to nest controller inside the definition block without redundantly declaring the prefix and namespace. So for instance at some time in the future you might have three or four administrative controllers. You can follow the same approach used to create the `User` controller, and then add them to `routes.php` like this:

```

1 Route::group(['prefix' => 'admin', 'namespace' => 'admin'], function()
2 {
3     Route::resource('category', 'CategoryController');
4     Route::resource('list', 'ListController');
5     Route::resource('product', 'ProductController');
6     Route::resource('user', 'UserController');
7 });

```

With this route definition in place, create a new directory named `admin` inside `resources/views`, and inside it create a directory named `user`. This will house the views associated with the new administrative `User` controller. Inside the `user` directory create a file named `index.blade.php` and add the following contents to it:

```

1 <h1>Registered Users</h1>
2
3 <ul>
4 @forelse ($users as $user)
5
6     <li>{{ $user->name }} ({{ $user->email }})</li>
7
8 @empty
9
10    <li>No registered users</li>
11
12 @endforelse
13 </ul>

```

Finally, open the new `User` controller (`app/Http/Controllers/admin/UserController.php`) and update the `index` action to look like this:

```

1 public function index()
2 {
3     $users = User::orderBy('created_at', 'desc')->get();
4     return view('admin.user.index')->withUsers($users);
5 }

```

With these changes in place you should be able to navigate to `/admin/user` and see a bulleted list of any registered users! Of course, before deploying this to production you'll want to restrict access to only those users identified as administrators. Let's do this next.

Restricting Access to the Administration Console

We want to allow only those users identified as administrators (their users table record's `is_admin` column is set to `true`). You might be tempted to make this determination by embedding code such as the following into your controller actions:

```
1  if (Auth::user()->is_admin != true) {  
2      return \Redirect::route('home')->withMessage('Access denied!');  
3  }
```

Don't do this! This is a job perfectly suited for custom middleware. Let's create a middleware which neatly packages this sort of logic, and then associate that middleware with our administrative controllers:

```
1  $ php artisan make:middleware AdminAuthentication  
2  Middleware created successfully.
```

This command created a new middleware skeleton named `AdminAuthentication.php` which resides inside `app/Http/Middleware`. Open this file and update it to look like the following:

```
1  namespace App\Http\Middleware;  
2  
3  use Closure;  
4  use Illuminate\Contracts\Auth\Guard;  
5  use Illuminate\Http\RedirectResponse;  
6  
7  class AdminAuthentication {  
8  
9      protected $auth;  
10  
11     public function __construct(Guard $auth)  
12     {  
13         $this->auth = $auth;  
14     }  
15  
16     public function handle($request, Closure $next)  
17     {  
18         if ($this->auth->check())  
19         {  
20             if ($this->auth->user()->is_admin == true)  
21             {
```

```

22         return $next($request);
23     }
24 }
25
26 return new RedirectResponse(url('/'));
27
28 }
29
30 }

```

Most of this should be familiar by now given coverage of authentication and middleware in chapters 5 and 6, respectively, so I won't belabor the changes. Save this file and then open `App/Http/Kernel.php` and register the middleware inside the `$routeMiddleware` array:

```

1 protected $routeMiddleware = [
2     ...
3     'admin' => \App\Http\Middleware\AdminAuthentication::class,
4 ];

```

With the middleware registered, all that remains is to associate the middleware with the route group:

```

1 Route::group(
2     [
3         'prefix' => 'admin',
4         'namespace' => 'admin',
5         'middleware' => 'admin'
6     ], function()
7     {
8         Route::resource('user', 'UserController');
9     });

```

Once you've saved this change to the `routes.php` file, your administrative controllers will be restricted to administrators!

Summary

Hopefully this brief chapter adequately demonstrated just how easy it is to create a restricted administrative console for your Laravel applications. Of course, if you require more sophisticated role-based features then definitely check out a package such as [Entrust](https://github.com/Zizaco/entrust)¹¹⁵ however for more simplistic requirements I certainly suggest embracing this straightforward approach!

¹¹⁵<https://github.com/Zizaco/entrust>

Chapter 9. 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](https://affiliate-program.amazon.com/)¹¹⁶. 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](https://affiliate-program.amazon.com/gp/advertising/api/detail/main.html)¹¹⁷. To implement such a solution you would typically write a script using a package such as [ApaiIO](https://github.com/Exeu/apai-io)¹¹⁸, and then schedule the script's execution using your server's [Cron](http://en.wikipedia.org/wiki/Cron)¹¹⁹ 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:

¹¹⁶<https://affiliate-program.amazon.com/>

¹¹⁷<https://affiliate-program.amazon.com/gp/advertising/api/detail/main.html>

¹¹⁸<https://github.com/Exeu/apai-io>

¹¹⁹<http://en.wikipedia.org/wiki/Cron>

```
1 <?php namespace App\Console;
2
3 use Illuminate\Console\Scheduling\Schedule;
4 use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
5
6 class Kernel extends ConsoleKernel {
7
8     protected $commands = [
9         'todoparrot\Console\Commands\Inspire',
10    ];
11
12    protected function schedule(Schedule $schedule)
13    {
14        $schedule->command('inspire')->hourly();
15    }
16
17 }
```

The protected `$commands` property registers any custom commands you'd like to include in the Artisan list output. An example custom command (`Inspire`) is already defined, which you'll find in `app/Console/Commands/Inspire.php`. Whether you plan on scheduling custom Artisan commands or executing them directly from the terminal you'll need to reference You're not strictly limited to scheduling Artisan commands, although as you'll soon see it is quite easy to create custom Artisan commands containing the desired logic.

The `inspire` command registered in the `$commands` array is scheduled for execution in the `schedule` method. In this example you can see it has been scheduled to execute every hour (at the top of the hour). I'll talk about other scheduling options in a moment. If you execute the `inspire` command manually you'll be presented with a random quote:

```
1 $ php artisan inspire
2 Simplicity is the ultimate sophistication. - Leonardo da Vinci
```

Next I'll show you how you can create your own custom Artisan command and schedule it for execution using the command scheduler.

Creating a Custom Artisan Command

You can create your own Artisan commands which can neatly package any PHP logic you desire. To create a command use the `make:console` generator:

```
1 $ php artisan make:console UpdateCatalog --command=amazon:update
2 Console command created successfully.
```

This creates a command skeleton in `app/Console/Commands/UpdateCatalog.php`. For organizational purposes I've define a custom command name `amazon:update`, as perhaps in the future I'd like to create other Amazon-related commands and so would like them all placed under the `amazon` namespace. Open up `app/Console/Commands/UpdateCatalog.php` and you'll find the following class:

```
1 <?php namespace Todoparrot\Console\Commands;
2
3 use Illuminate\Console\Command;
4 use Symfony\Component\Console\Input\InputOption;
5 use Symfony\Component\Console\Input\InputArgument;
6
7 class UpdateCatalog extends Command {
8
9     protected $name = 'amazon:update';
10
11     protected $description = 'Command description.';
12
13     public function __construct()
14     {
15         parent::__construct();
16     }
17
18     public function fire()
19     {
20         //
21     }
22
23     protected function getArguments()
24     {
25         return [
26             ['example', InputArgument::REQUIRED,
27              'An example argument.'],
28         ];
29     }
30
31     protected function getOptions()
32     {
33         return [
```



```

34         ['example', null, InputOption::VALUE_OPTIONAL,
35         'An example option.', null],
36     ];
37 }
38
39 }

```

The `$name` and `$description` properties define the command's execution name and description, respectively, both of which will be included in the Artisan `list` output once we register it within the `app/Console/Kernel.php` `$commands` array. The `fire` method encapsulates the logic which will execute when the command is run. The `getArguments` and `getOptions` methods can be used to define both required and optional command arguments and options, respectively.

You'll see that the `getArguments` method defines a required argument. For the purposes of this exercise we're not interested in arguments nor options, so comment out the `return` statement:

```

1  protected function getArguments()
2  {
3      return [
4          // ['example', InputArgument::REQUIRED,
5          // 'An example argument.'],
6      ];
7  }

```



The Laravel documentation discusses Artisan command arguments, options, and other features. See [this page](http://laravel.com/docs/master/commands)¹²⁰ for more information.

Next, update the `fire` method to look like this:

```

1  public function fire()
2  {
3      $this->info("Amazon catalog updated!");
4  }

```

Save your changes and then register the command within `app/Console/Kernel.php`:

¹²⁰<http://laravel.com/docs/master/commands>

```
1 protected $commands = [  
2     'todoparrot\Console\Commands\Inspire',  
3     'todoparrot\Console\Commands\UpdateCatalog'  
4 ];
```

After saving the changes you should see the custom command in the Artisan `list` output:

```
1 $ php artisan list  
2 ...  
3 Available commands:  
4 ...  
5 amazon  
6 amazon:update          Updates the TODOParrot book catalog.  
7 ...
```

You can now execute the `amazon:update` command from the terminal:

```
1 $ php artisan amazon:update  
2 Amazon catalog updated!
```

Scheduling Your Command

As was perhaps made obvious by the earlier example, scheduling your command within `app/Console/Kernel.php` is easy. If you'd like `amazon:update` to run hourly, you'll use the `hourly` method:

```
1 protected function schedule(Schedule $schedule)  
2 {  
3     $schedule->command('amazon:update')->hourly();  
4 }
```

Updating Amazon product information hourly seems a bit aggressive. Fortunately, you have plenty of other options. To run a command on a daily basis (midnight), use `daily`:

```
1 $schedule->command('amazon:update')->daily();
```

To run it at a specific time, use the `dailyAt` method:

```
1 $schedule->command('amazon:update')->dailyAt('18:00');
```

If you need to run a command very frequently, you can use an `every` method:

```
1 $schedule->command('amazon:update')->everyFiveMinutes();
2 $schedule->command('amazon:update')->everyTwentyMinutes();
```

See the [Laravel documentation](#)¹²¹ for other scheduling options.

Enabling the Scheduler

With your tasks created and scheduled, you'll need to add a single entry to your server's crontab file:

```
1 * * * * * php /path/to/artisan schedule:run 1>> /dev/null 2>&1
```

Once saved, your application's `schedule:run` Artisan command will run once per minute. It will in turn execute any jobs that you've defined using the Laravel command scheduler.

Other Scheduling Options

If defining a custom Artisan command seems overkill, you can optionally define some logic for execution directly within the `schedule` method:

```
1 $schedule->call(function()
2 {
3     // Send some e-mail
4 }->daily();
```

You can also schedule terminal commands for execution like so:

```
1 $schedule->exec('/path/to/some/command')->daily();
```

The new command scheduler is a pretty powerful tool that eliminates the need to separately manage regularly executing tasks. This is a supremely well-implemented feature, and may very well be my favorite Laravel 5 capability.

Optimizing Your Application

Before deploying your application you'll logically want to ensure the code has been properly optimized for a production environment. Frankly, entire books have been written about tuning web applications, and therefore this discussion could go in many directions and really never even begin scratch the surface. Therefore I think it makes the most sense to focus on a few key *Laravel-specific* optimization features, for the moment leaving third-party optimization solutions out of the discussion. In future iterations I'll selectively expand this section to cover other topics.

¹²¹<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](#)¹²². You can improve performance by using Artisan's `optimize` command to significantly improve the efficiency in which your project classes are loaded. You'll invoke `optimize` like so:

```
1 $ php artisan optimize
```

This command will by default run Composer's `dump-autoload` command with the `--optimize` option. This command will in turn create `vendor/composer/autoload_classmap.php` which contains an array consisting of all class names and the paths to their corresponding files. This file is then subsequently used to quickly load third-party classes because the array can be used for referential purposes rather than requiring the autoloader to separately find and open each class file.

Additionally, Laravel will further optimize matters by concatenating all of its own native classes into a single file found at `storage/framework/compiled.php`, and also create a file named `services.json` in the same directory. This file is used to optimize the loading of your project's service providers. Further, it will cache all of your project's views within `storage/framework/views`.

However, if your project's `APP_DEBUG` configuration setting is set to `true` (the default when in your development environment), this command will not work as intended because Laravel presumes you'll always want to be working with the very latest versions of your files rather than rely on a cache. You can override this behavior in the local environment by including the `--force` option:

```
1 $ php artisan optimize --force
2 Generating optimized class loader
3 Compiling common classes
4 Compiling views
```

Now you'll be able to peruse `storage/framework/compiled.php` and `storage/framework/services.json` even when working in the local environment.

You can replace `compiled.php` and `services.json` by running `optimize` anew, keeping in mind you'll need the `--force` option if you'd like to experiment with it locally. If you'd like to remove these files altogether, run the following command:

```
1 $ php artisan clear-compiled
```

¹²²<https://getcomposer.org/>

This command (also confusingly) does not however automatically delete the compiled views which were generated when the `--force` option was provided. At this time there is not any documented solution for deleting these files via Artisan, so you could optionally (and rather easily) write your own Artisan command for doing so, or just navigate to `storage/framework/views` and manually delete all of the files.

Caching Route Definitions

The route definitions found in `app/Http/routes.php` are by default read into the framework as part of the bootstrapping process. You can cache these routes by encoding and serializing them using Artisan's `route:cache` command:

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

This cache file is stored in `vendor/routes.php`. If this file exists, Laravel will refer to it rather than parsing the source route definitions file. You can delete the route cache file using `route:clear`:

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

Optimizing Your CSS and JavaScript

You'll always want to minimize the number and size of requests required to render your site within the user's browser. One of the easiest things you can do in this regards is to optimize your CSS and JavaScript, as well as take advantage of content delivery networks.

Combining your CSS using Elixir and a CSS preprocessor such as [Less](http://lesscss.org/)¹²³ is pretty easy; take a look at `resources/assets/less/app.less` and you'll see how to use Less' `@import` directive to combine multiple CSS files:

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

¹²³<http://lesscss.org/>

Laravel's default `gulpfile.js` uses `mix.less` to compile the `app.less` file, saving the combined CSS output to `public/css/app.css`. This is great because it combines the more than 40 Bootstrap CSS source files found in `resources/assets/less/bootstrap` and the custom Less CSS found in `app.less` into a single file. However you'll additionally want to minify the CSS (remove all whitespace and comments). You can do so by passing `--production` to `gulp`:

```
1 $ gulp --production
```

Just taking the default `app.less` and Bootstrap files into account, using `--production` results in a 20% reduction in the compiled `app.css` file size!

You can additionally combine your JavaScript files together. For instance if you are managing two separate CoffeeScript files in `resources/assets/coffee` named `test.coffee` and `test2.coffee`, and wanted to combine the compiled output into a single file within `public/js` you can update `gulpfile.js` like so:

```
1 mix.coffee().scriptsIn('public/js', 'public/js');
```

When the CoffeeScript files found in `resources/assets/coffee` are compiled and saved to `public/js`, the `scriptsIn` method will subsequently concatenate these files together and save them to a file named `all.js`. Passing `--production` to the `gulp` command will additionally minify the concatenated JavaScript file.

When relying on third-party libraries such as jQuery you'll almost certainly want to use a CDN (Content Delivery Network) rather than locally host your own copy. This may seem counterintuitive, however the reasoning behind this best practice is explained [here](#)¹²⁴.

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 a 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](#)¹²⁵, and introduce you to [Laravel Forge](#)¹²⁶. 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.

¹²⁴<http://encosia.com/3-reasons-why-you-should-let-google-host-jquery-for-you/>

¹²⁵<http://heroku.com>

¹²⁶<https://forge.laravel.com>

Deploying to Heroku

[Heroku](http://heroku.com)¹²⁷ 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](http://www.wjgilmore.com/books/easy-heroku-rails.html)"¹²⁸). 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](https://www.heroku.com/pricing)¹²⁹ 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>¹³⁰). 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/>¹³¹). The Heroku Toolbelt is a terminal utility you'll use to manage various aspects of your Heroku-hosted project, including the actual deployment process, migrating your database, and interacting with the Heroku servers in various ways. To install the Heroku Toolbelt, head on over to <https://toolbelt.heroku.com/>¹³², 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`:

¹²⁷<http://heroku.com>

¹²⁸<http://www.wjgilmore.com/books/easy-heroku-rails.html>

¹²⁹<https://www.heroku.com/pricing>

¹³⁰<https://signup.heroku.com>

¹³¹<https://toolbelt.heroku.com/>

¹³²<https://toolbelt.heroku.com/>

```

1  $ heroku
2  Usage: heroku COMMAND [--app APP] [command-specific-options]
3
4  Primary help topics, type "heroku help TOPIC" for more details:
5
6  addons      #  manage addon resources
7  apps        #  manage apps (create, destroy)
8  ...
9  update      #  update the heroku client
10 version     #  display version

```

You'll be greeted with a lengthy list of commands. Introducing all of these commands is well out of the scope of this chapter, however feel free to take a moment to read the command descriptions and learn more about them by executing `heroku help` and then the name of the command (e.g. `heroku help logs`).

Deploying Your Application

With your Heroku account created and the Heroku Toolbelt installed, it's time to deploy a Laravel application. As you'll soon see, this is incredibly easy to do. For purposes of this example let's just deploy a new application:

```

1  $ composer create-project laravel/laravel dev.herokutest.com --prefer-dist
2  Installing laravel/laravel (v5.0.0)
3  - Installing laravel/laravel (v5.0.0)
4  ...
5  Compiling views
6  Application key [9UCBk7IDjvAGrkLOUBXw43yYKlymlqE3Y] set successfully.

```

With the project created, you'll next want to create a `Procfile`, placing this file in your Laravel project's root directory. The file's capitalization is important, and it should not have an extension. Heroku reads this `Procfile` to determine what types of processes should launch when your application is deployed to one of their servers. In the case of a Laravel application we want to declare a web process type, identify the web server used to serve the application, and identify the application's document root directory, which in the case of Laravel is `public`. Therefore the `Procfile` should consist of the following single line:

```

1  web: vendor/bin/heroku-php-apache2 public

```

Incidentally, other options are available; see [the Heroku PHP documentation](https://devcenter.heroku.com/articles/custom-php-settings#setting-the-document-root)¹³³ 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:

¹³³<https://devcenter.heroku.com/articles/custom-php-settings#setting-the-document-root>


```
1 $ git init
2 Initialized empty Git repository in /Users/wjgilmore/Software/dev.herokutest.com\
3 /.git/
4 $ git add .
5 $ git commit -m "First commit"
```

You'll want to use Git in particular because not only do all new Laravel projects come with some Git-specific features (`.gitignore` files in the appropriate directories, namely), but Heroku will also interact with your local Git repository to make deployment even easier than it otherwise would be. If you're not familiar with Git I suggest reading at least the first few chapters of "Pro Git"¹³⁴ (free to read online) and checking out the interactive Git tutorial at <https://try.github.io/>¹³⁵.

With your repository created, it's time to deploy! Use the Heroku Toolbelt to initialize a new Heroku project:

```
1 $ heroku create
2 Creating lit-retreat-6653... done, stack is cedar-14
3 https://lit-retreat-6653.herokuapp.com/ | https://git.heroku.com/lit-retreat-665\
4 3.git
5 Git remote heroku added
```

Note how this command created a new name for your application (in my case, `lit-retreat-6653`), and then identified a URL where the application can be accessed. If you head over to your project's URL now, you'll see a standard Heroku welcome placeholder.

Additionally, it created a Git "remote". A remote repository is simply a Git repository for your project that resides somewhere else. You can push changes to these repositories, and pull changes from them. In the case of Heroku we'll only ever push changes to the newly created Git remote. I'll show you how to push these changes in just a moment but first we need to make one quick configuration change. Namely, you need to tell Heroku what *buildpack* to use. Buildpacks tell Heroku more about the software that should be configured on the server when your application is installed. You can do so using the Heroku Toolbelt's `config:add` command:

```
1 $ heroku config:add \
2 > BUILDPACK_URL=https://github.com/heroku/heroku-buildpack-php
3 Setting config vars and restarting lit-retreat-6653... done, v5
4 BUILDPACK_URL: https://github.com/heroku/heroku-buildpack-php
```

Incidentally, if you're managing multiple applications in Heroku you'll additionally need to specify the application name using the `--app` flag.

Finally, it's time to deploy! You can push your local changes to this remote by executing the following command:

¹³⁴<https://progit.org/>

¹³⁵<https://try.github.io/>

```
1  $ git push heroku master
2  Counting objects: 5, done.
3  Delta compression using up to 4 threads.
4  Compressing objects: 100% (5/5), done.
5  Writing objects: 100% (5/5), 416 bytes | 0 bytes/s, done.
6  Total 5 (delta 4), reused 0 (delta 0)
7  remote: Compressing source files... done.
8  remote: Building source:
9  remote:
10 remote: -----> Fetching custom git buildpack... done
11 remote: -----> PHP app detected
12 remote: -----> No runtime required in composer.json, defaulting to PHP 5.6.5.
13 remote: -----> Installing system packages...
14 ...
15 remote: -----> Launching... done, v6
16 remote:          https://lit-retreat-6653.herokuapp.com/ deployed to Heroku
17 remote:
18 remote: Verifying deploy... done.
19 To https://git.heroku.com/lit-retreat-6653.git
20    4ece26b..938feb8  master -> master
```

Congratulations! Your application has been deployed. Head on over to your designated URL and you should see the default Laravel splash page.



The URL generated when you created the Heroku application is just for testing purposes; you can easily swap it out with a custom domain. See the Heroku documentation for more details.

Migrating Your Database

If you've been closely following along and deployed a brand new Laravel application, then presumably you successfully saw the default splash page load to your designated Heroku URL. However, if your project is backed by a database, then you'll additionally need to at a minimum ensuring that any outstanding migrations are executed following deployment. However, if this is your first interaction with Heroku in the context of the new application, you'll need to provision the database, which you can do with the Heroku Toolbelt:

```
1 $ heroku addons:add heroku-postgresql:hobby-dev
2 Adding heroku-postgresql:hobby-dev on lit-retreat-6653...
3 done, v8 (free)
4 Attached as HEROKU_POSTGRESQL_NAVY_URL
5 Database has been created and is available
6 ! This database is empty. If upgrading, you can transfer
7 ! data from another database with pgbackups:restore.
8 Use `heroku addons:docs heroku-postgresql` to view documentation.
```

This command creates a new PostgreSQL database. Specifically, this database is identified by the plan *Hobby Dev*, which is free but has some significant limitations (notably a limit of 10,000 rows). If you are interested in using Heroku for any long term project I strongly suggest carefully learning more about the various PostgreSQL plans [here](#)¹³⁶.

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](#)¹³⁷ addon. However for reasons of convenience I'll stick to using PostgreSQL in this section if for any other reason because you'll find the majority of Heroku's documentation tends to be PostgreSQL-centric.

With the database created, execute the following command to learn more about your database's access credentials:

```
1 $ heroku config --app lit-retreat-6653 | grep DATABASE_URL
2 DATABASE_URL: postgres://USERNAME:PASSWORD@HOSTNAME:PORT/DATABASE
```

In the command output I've swapped out my access credentials with placeholders so you can easily identify the constituent parts. However, you don't actually need to write these down, because the `DATABASE_URL` variable is automatically stored in your server's configuration settings. In order to transparently manage your database configuration variables in both the development and production environments, you could save yourself quite a bit of hassle by using PostgreSQL locally and saving an identically-formatted environment variable within your local environment.



Obviously you'll also need to install and configure PostgreSQL within your local environment if it's not already available. See <http://www.postgresql.org>¹³⁸ for installation instructions.

Exactly how you'll do this will depend upon your particular operating system, so consult the appropriate online documentation for more details. However, once the local environment variable

¹³⁶<https://addons.heroku.com/heroku-postgresql>

¹³⁷<https://devcenter.heroku.com/articles/cleardb>

¹³⁸<http://www.postgresql.org/>

is in place there are a variety of ways you can reference it within your code. One of the most straightforward ways involves parsing the variable as a URL using PHP's `parse_url()` function directly within the `config/database.php` file. Also, you'll need to set the database default to `pgsql`:

```

1  'default' => 'pgsql',
2
3  ...
4
5  'pgsql' => [
6      'driver'   => 'pgsql',
7      'host'     => parse_url(getenv("DATABASE_URL"))["host"],
8      'database' => substr(parse_url(getenv("DATABASE_URL"))["path"], 1),
9      'username' => parse_url(getenv("DATABASE_URL"))["user"],
10     'password' => parse_url(getenv("DATABASE_URL"))["pass"],
11     'charset'  => 'utf8',
12     'prefix'   => '',
13     'schema'   => 'public',
14 ],

```

Save the changes and consider creating a model and corresponding migration to confirm you're able to properly connect to the new PostgreSQL database. After doing so, commit your changes and push them to Heroku:

```

1  $ git add .
2  $ git commit -m "Updated database configuration"
3  $ git push heroku master

```

Next, you'll want to migrate the database. You can easily do this using the Heroku Toolbelt's `run` command:

```

1  $ heroku run php artisan migrate --app lit-retreat-6653
2  Running `php artisan migrate` attached to terminal... up, run.6981
3  *****
4  *      Application In Production!      *
5  *****
6
7  Do you really wish to run this command? [y/N] y
8  Migration table created successfully.
9  Migrated: 2014_10_12_000000_create_users_table
10 Migrated: 2014_10_12_100000_create_password_resets_table
11 Migrated: 2015_01_30_032004_create_todolists_table.php

```

Your migrations are now in place!

Introducing Laravel Forge

Like Heroku, Laravel Forge (<https://forge.laravel.com/>¹³⁹) 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 repository and deploy it to the designated server.
- **Automated Monitoring:** Laravel Forge users have the option of using the popular [New Relic](#)¹⁴⁰ and [Papertrail](#)¹⁴¹ monitoring agents.
- **Simple Scheduling:** If you're using [Laravel Queues](#)¹⁴² (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](#)¹⁴³ or [DigitalOcean](#)¹⁴⁴ (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](#)¹⁴⁵ 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](#)¹⁴⁶.

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

¹³⁹<https://forge.laravel.com/>

¹⁴⁰<http://newrelic.com/>

¹⁴¹<https://papertrailapp.com/>

¹⁴²<http://laravel.com/docs/master/queues>

¹⁴³<https://www.linode.com/>

¹⁴⁴<https://www.digitalocean.com/>

¹⁴⁵<https://www.digitalocean.com/pricing/>

¹⁴⁶<http://mattstauffer.co/blog/getting-your-first-site-up-and-running-in-laravel-forge>

accessing the site and so you should put it in maintenance mode. To place your application in maintenance mode you'll execute Artisan's down command:

```
1 $ php artisan down
2 Application is now in maintenance mode.
```

At this point you can proceed with your system upgrade. Once the maintenance is complete, don't forget to bring the application back online using the up command:

```
1 $ php artisan up
2 Application is now live.
```

Obviously running either of these commands locally will only result in toggling your local, development application's maintenance status, therefore be sure to run the command on your production server in order to achieve the desired result. For instance to enable maintenance mode on Heroku (see the earlier section regarding deploying to Heroku for more context) you'd run the following command:

```
1 $ heroku run php artisan down --app lit-retreat-6653
```

Summary

While it's always fun to imagine and create new features, always keep in mind that nothing is real until you actually ship the application to the world. In order to do so in the most effective way possible you'll need to establish and implement a rigorous deployment process, and continually refine that process over time. In doing so, you'll be able to more rapidly and effectively respond to your users' needs, not to mention save your sanity.

Chapter 10. Introducing Lumen

I've spent the past nine chapters and more than 200 pages praising Laravel's virtues. The framework offers countless useful features, is endlessly configurable, and benefits from an enormous ecosystem of third-party packages.

But what if you didn't *need* all of these capabilities? What if your primary priority was performance above everything else? This role has historically been filled by *microframeworks*. Microframeworks are effectively stripped down variants of the so-called enterprise frameworks such as Laravel, Symfony, and Zend Framework. Although lacking many of the features packaged into these larger solutions, microframeworks counterbalance their relatively few capabilities by offering superior performance in addition to generally being easier to learn.

Fortunately, Laravel developers can now take advantage of an incredibly fast microframework that is nonetheless decidedly Laravelish in nature. It's called [Lumen](http://lumen.laravel.com/)¹⁴⁷, and it was first released by Laravel creator Taylor Otwell in April, 2015. Lumen offers many of the very same features also available to Laravel, and in fact a quick glance of the code powering a Lumen application might not suffice to tell the difference!

So for what sort of applications might you choose to use Lumen instead of its larger sibling? Again, generally speaking it might come into consideration anytime performance becomes a priority. For instance if a particular area of your web application is used with much greater frequency than other areas and it's beginning to degrade overall performance, then you might consider rebuilding that popular feature using Lumen and integrating it alongside the Laravel application. Alternatively, if you were building a REST API then you might consider using Lumen since the service likely won't require features such as Blade templating or Eloquent, two capabilities not enabled in Lumen by default (but are nonetheless available should you need them, as will be discussed later in this chapter).

In this chapter we'll build a simple REST API for the TODOParrot application. Specifically, we'll create a small [microservice](http://martinfowler.com/articles/microservices.html)¹⁴⁸ which is responsible for retrieving a few simple system statistics, returning them in JSON format. You could then consume this service within multiple venues, such as the TODOParrot website and maybe an iPhone application used by an administrator for monitoring the system. This sort of solution is precisely where Lumen's strengths lie given the framework's speed and bare bones feature set.



You'll find all of the source code used to power the sample Lumen application discussed in this chapter over on GitHub at <https://github.com/wjgilmore/status.todoparrot.com>¹⁴⁹.

¹⁴⁷<http://lumen.laravel.com/>

¹⁴⁸<http://martinfowler.com/articles/microservices.html>

¹⁴⁹<https://github.com/wjgilmore/status.todoparrot.com>

Creating Your First Lumen Application

You'll generate Lumen applications in much the same way you learned how to generate Laravel applications back in Chapter 1. Begin by installing the Lumen package:

```
1 $ composer global require "laravel/lumen-installer=~1.0"
```

Once installed you're ready to create your first Lumen application:

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

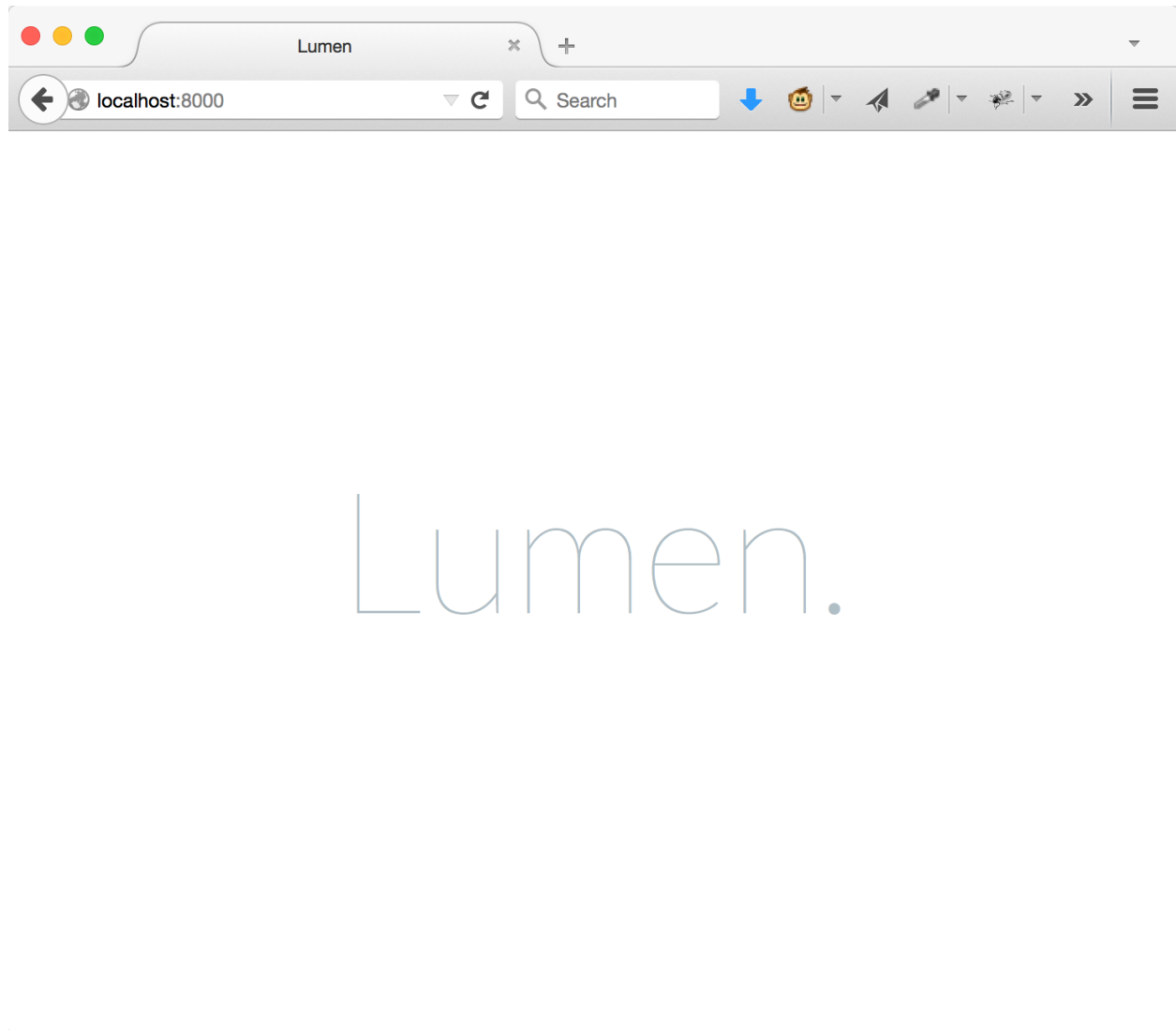
Next, you'll want to copy `.env.example`, creating `.env`. For the purposes of this example I've updated the `.env` `DB_*` parameters to use the same settings as those used for my local instance of the TODOParrot web application, thus giving the Lumen application access to the same database. With the `.env` file created, you'll next need to enable the `phpdotenv` library by opening the `bootstrap/app.php` file and uncommenting the following line:

```
1 Dotenv::load(__DIR__.'/../');
```

After saving these changes, return to the terminal and launch the application just to ensure everything is working properly. For sake of demonstration I'll launch it using the built-in PHP server:

```
1 $ php artisan serve
```

Open your browser, navigate to `http://localhost:8000`, and you should see the splash page presented in the following screenshot.



The default Lumen application homepage

Once you've confirmed the application is up and running, return to the terminal and execute `php artisan list` to view the list of available Artisan commands:

```
1 $ php artisan list
2 ...
3 cache
4   cache:clear      Flush the application cache
5   cache:table      Create a migration for the cache database table
6 db
7   db:seed          Seed the database with records
8 make
9   make:migration   Create a new migration file
```

10	migrate	
11	migrate:install	Create the migration repository
12	migrate:refresh	Reset and re-run all migrations
13	migrate:reset	Rollback all database migrations
14	migrate:rollback	Rollback the last database migration
15	migrate:status	Show the status of each migration
16		queue
17	queue:failed	List all of the failed queue jobs
18	queue:failed-table	Create a migration for the failed queue jobs
19		database table
20	queue:flush	Flush all of the failed queue jobs
21	queue:forget	Delete a failed queue job
22	queue:listen	Listen to a given queue
23	queue:restart	Restart queue worker daemons after their current job
24	queue:retry	Retry a failed queue job
25	queue:subscribe	Subscribe a URL to an Iron.io push queue
26	queue:table	Create a migration for the queue jobs database table
27	queue:work	Process the next job on a queue
28	schedule	
29	schedule:run	Run the scheduled commands

Notice this list is significantly smaller than that available to the typical Laravel application. Notably, almost all of the `make` commands are missing, meaning it's not even possible at this time to automate the generation of a controller. Frankly I'm not quite sure why the generators aren't available by default to Lumen application, however manually generating skeletons for controllers or models isn't really a big deal, as you'll see in a moment.

Creating a Status API

We'll create a RESTful API for retrieving various usage statistics. Begin by creating a new file named `StatusController.php` inside `app/Http/Controllers`, and adding the following contents to it:

```

1  <?php namespace App\Http\Controllers;
2
3  use App\Http\Requests;
4  use App\Http\Controllers\Controller;
5  use Illuminate\Http\Request;
6
7  class StatusController extends Controller {
8
9      public function index()
```

```
10     {
11         return response()->json(['status' => 'Polly wants a cracker!']);
12     }
13
14 }
```

Laravel's response helper is very useful when you'd like to return JSON because it will automatically set the Content-Type header to `application/json`. Properly setting this header is crucial in order to ensure clients can properly retrieve and parse the returned JSON, and therefore taking advantage of response in order to ensure this important (but easy to forget) matter is handled.

TODO: NEEDS UPDATED TO REFLECT FACT THE ROUTES ARE NOW WRAPPED IN NAMES-SPACE: <http://stackoverflow.com/questions/29692745/lumen-framework-routing-not-working>

Next, open the `app/Http/routes.php` file and add the following route declaration:

```
1 $app->get('/status', 'App\Http\Controllers\\StatusController@index');
```

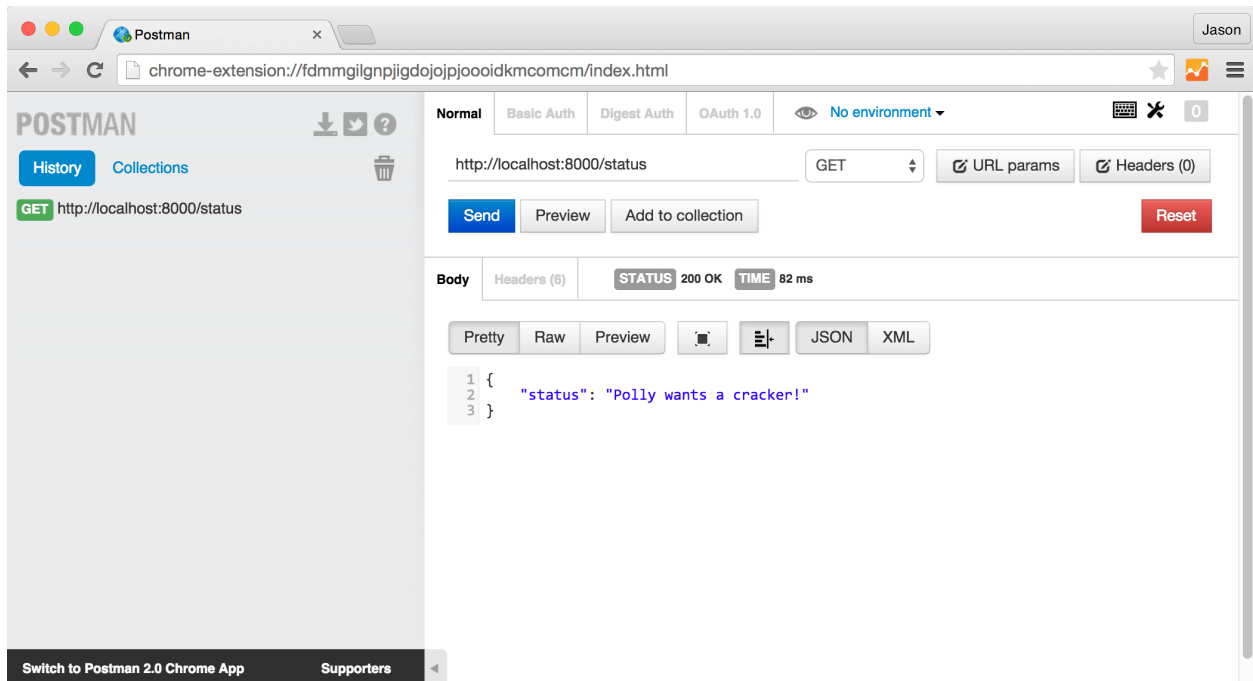
As you can see this route declaration differs quite significantly from the declarations found in a typical Laravel application. Most notably, you need to define the path to the route's associated controller and action, because Lumen is not going to autoload these controllers in order to avoid the performance cost of doing so.

After saving these changes, return to the browser and navigate to `/status`, and you should see the following JSON output to the browser window:

```
1 {"status": "Polly wants a cracker!"}
```

Incidentally, I tend to avoid interacting with JSON-based APIs in this manner during the development phase, instead using a Chrome extension called [Postman](https://www.getpostman.com)¹⁵⁰. Postman was created expressly for this purpose, allowing you to easily view JSON data and headers, set URL parameters, manage authentication settings, and conveniently save endpoints for later reference. Here's a screenshot demonstrating Postman being used to access the `/status` endpoint:

¹⁵⁰<https://www.getpostman.com>



Viewing a JSON endpoint using Postman

Talking to the Database

The previous `/status` endpoint was intended to highlight a few differences between Lumen and Laravel such as controller/action creation and route declarations; otherwise it is obviously pretty useless. So let's add a new endpoint which will return the total number of TODO lists created to date. Although you are already well aware how to talk to the database in a Laravel application, and have the option of doing so in the very same fashion within a Lumen application, there are a few additional configuration-related steps you'll need to take in order to do so.

Let's begin by making the `DB_*` configuration settings found in the `.env` file available to your application. Although these settings are already available in `.env`, a Lumen application doesn't yet know what to do with them! And here you'll encounter the first notable departure from the standard Laravel framework; neither the `database.php` configuration file nor for that matter the `config` directory exists by default in a Lumen application! Lumen does however *support* these files, so let's bring `database.php` into the picture by first creating the `config` directory inside your project's root directory:

```
1 $ mkdir config
```

Next, copy the `database.php` directory from `vendor/laravel/lumen-framework/config/`, placing the copy inside the newly created `config` directory:

```
1 $ cp vendor/laravel/lumen-framework/config/database.php config/
```

With the config directory and app.php file in place, Lumen will automatically begin incorporating the .env file's DB_* settings into your application!

Next you'll want to open bootstrap/app.php and uncomment the following line:

```
1 $app->withFacades();
```

Uncommenting this line will allow you to use the DB facade and therefore the Laravel Query Builder syntax. With these configuration changes in place we can create a new action and retrieve the list count. Add the following action to the Status controller:

```
1 public function lists()  
2 {  
3  
4     $result = \DB::select('select count(id) as `count` from todolists');  
5  
6     return response()->json(['count' => $result[0]->count]);  
7 }
```

Save these changes and then open the app/Http/routes.php file and add the following route definition:

```
1 $app->get('/lists/count', 'App\Http\Controllers\StatusController@lists');
```

Keep in mind you are certainly free to add multiple controllers to a Lumen application! Because this microservice example is intended to be very focused, I'm just being a bit naughty here and creating a convenience route (/lists/count) that obviously doesn't abide by standard route naming conventions. After saving these changes, return to your browser (or preferably Postman or a similar tool), and navigate to /lists/count. You should output similar to the following (formatted for readability):

```
1 {  
2     "count": 643  
3 }
```

Incidentally, it's also possible to use Eloquent, although you'll need to first uncomment the following line from bootstrap/app.php:

```
1 $app->withEloquent()
```

Additionally you'll have to create companion models within your Lumen application. I suggest sticking to the DB facade whenever possible as not only will the application perform slightly faster but you additionally won't have to deal with potentially redundant code (similar or identical to that found in a companion Laravel application).

Integrating the Lumen Application Into TODOParrot.com

After having created and deployed the microservice, it's time to incorporate it into the TODOParrot website. If you head over to the TODOParrot.com homepage, you'll see the homepage includes a reference to the total number of lists created (directly below the button found in the middle of the page). This information is retrieved in real-time from the microservice! I use a simple jQuery-driven AJAX call to talk to the Lumen `/lists/count` endpoint. Because this JavaScript should only execute on the home page, I added a `@yield` statement to the `layouts/master.blade.php` file:

```
1 @yield('footer_js')
```

Next, in `welcome.blade.php`, I added the following:

```
1 @section('footer_js')
2 <script>
3     $.ajax({
4         url: "http://status.todoparrot.com/lists/count",
5         dataType: 'jsonp',
6         crossDomain: true,
7         success: function(data) {
8             $('#list_count').html(data.count + " lists created!");
9         },
10        error: function(data) {
11            $('#list_count').html('Squawk!');
12        }
13    });
14 </script>
15 @endsection
```

Keep in mind for the purposes of this demonstration I've created a *cross-domain* Ajax request, and therefore the Lumen response varies slightly from that presented earlier in this chapter in order to accommodate the callback wrapper required by JSONP-oriented responses. This is trivial to add

within Laravel/Lumen applications, and you'll see how it's done in the example microservice source code found [on GitHub](#)¹⁵¹

If your microservice exposes potentially sensitive information, then obviously an approach such as the above isn't going to be secure. You'll need to research solutions for properly securing an API and how credentials are managed when working with JavaScript-based clients. Perhaps I'll expand upon these matters in a future iteration of this chapter but for now I just wanted to make some Lumen-related information available to readers!

Summary

Hopefully this brief introduction to the Lumen microframework served to get your mind racing regarding new ways to increase application performance, not to mention new microservice possibilities. The Laravel team has per usual put together a pretty [comprehensive set of documentation](#)¹⁵² so I suggest having a thorough look at it in order to gain an even deeper understanding of what's available via this amazingly fast new solution!

¹⁵¹<https://github.com/wjgilmore/status.todoparrot.com>

¹⁵²<http://lumen.laravel.com/docs>

Chapter 11. Introducing Events

Your project's success can be quantified by usage metrics such as new registrations, user activity, and conversions (purchases of upgrades or other products). In order to stay abreast of these metrics you might wish to receive an e-mail, update an analytics dashboard, or send a message to a group chat application such as [Slack](https://slack.com)¹⁵³.

How might you be notified of these milestones? Adding custom logic to the associated controller actions seems like the obvious solution, however such an approach violates the convention of limiting an action's responsibility to one specific task. To illustrate the sort of problems which might arise from such haphazard expansion of a controller action's responsibilities, suppose you wanted to be notified every time a particular milestone is met (such as a new user registration), and so modify the associated action to send an e-mail every time the action is executed. Over time your list of desired notification metrics grows to include active users and conversions, and with it the number of tweaks to other actions. Eventually, your continued success forces a frenetic refactoring of these various notifications to eliminate the constant inbox flooding and ensure more efficient analysis of these milestones.

Fortunately Laravel 5 offers a much more attractive alternative to this common dilemma. Known as *events*, you can create easily maintainable bits of logic which can be triggered to execute in conjunction with a specific occurrence. In this chapter I'll walk you through the steps necessary to incorporate a custom event into your application. You'll also learn about a new Laravel 5.1 feature that allows you to *broadcast* events to all of the users currently interacting with the web application.

Creating an Event

Laravel events are actually comprised of two parts:

- **Event Handler:** The event handler contains the information associated with the event. For instance, if your goal is to trigger an event associated with the creation of a list, then that `List` object would be made available (along with any other information you require) to the event handler. This information will subsequently be made available to the event listener, introduced next.
- **Event Listener:** The event listener "listens" for the event instance, and responds to it accordingly. It is here where you will implement the event implementation logic.

So in a nutshell, the event handler will be attached to the application logic you'd like to monitor, and the event listener will execute once the event handler is triggered (or *fired*, in Laravel nomenclature).

¹⁵³[http://slack.com](https://slack.com)

Both the event listener and event handler are defined within standard PHP classes, and are subsequently associated with one another using a simple declaration found in a configuration file. In this section I'll guide you through the process of creating a listener and handler, binding the two together, and then integrating the event into the TODOParrot application so that a log message is generated every time a new list is created. I'll keep the actual event logic simple because that's not really the point of this exercise; as you'll soon see it will be trivial to substitute this logic for a more practical outcome such as sending an e-mail or updating the database.

Defining the Event Handler

The event handler is responsible for initializing the event execution process, firing in conjunction with the application logic you desire to monitor. Let's create an event handler which will execute in conjunction with the creation of a new list. You can do so using Artisan's `event:generate` command:

```
1 $ php artisan make:event ListWasCreated
2 Event created successfully.
```

This command generated the event handler skeleton, which you'll find in `app/Events/ListWasCreated.php`. The file looks like this:

```
1 <?php
2
3 namespace Todoparrot\Events;
4
5 use Todoparrot\Events\Event;
6 use Illuminate\Queue\SerializesModels;
7 use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
8
9 class ListWasCreated extends Event
10 {
11     use SerializesModels;
12
13     /**
14      * Create a new event instance.
15      *
16      * @return void
17      */
18     public function __construct()
19     {
20         //
21     }
22 }
```

```

23     /**
24      * Get the channels the event should be broadcast on.
25      *
26      * @return array
27      */
28     public function broadcastOn()
29     {
30         return [];
31     }
32 }

```

As you can see, this is just a standard PHP class that extends Laravel’s `Event` class. The `SerializesModels` trait will help to more effectively manage any Eloquent models as they pass through the handler, and is likely not something you’ll need to worry about when getting started using events. The `broadcastOn` method is new to Laravel 5.1, and we’ll return to the purpose of this method later in the chapter.

The constructor is by default empty although at a minimum you’ll want to pass along the `List` object created within the logic you plan on monitoring:

```

1  use Todoparrot\Todolist;
2
3  class ListWasCreated extends Event
4  {
5
6      ...
7
8      public $list;
9
10     public function __construct(Todolist $list)
11     {
12         $this->list = $list;
13     }
14
15     ...
16
17 }

```

This `$list` attribute will in turn be exposed to the event listener. Let’s create that listener next.

Defining the Event Listener

The event listener contains the implementation logic which should be executed when the event is triggered. You can use Artisan to generate the listener skeleton, as demonstrated here:

```

1 $ php artisan make:listener LogMessageWhenListCreated \
2 > --event="ListWasCreated"

```

Note I've identified the name of the event handler created previously (`ListWasCreated`). This command generates the event listener skeleton, which you'll find in `app/Listeners/LogMessageWhenListCreated.php`. The file looks like this:

```

1 <?php
2
3 namespace Todoparrot\Listeners;
4
5 use Todoparrot\Events\ListWasCreated;
6 use Illuminate\Queue\InteractsWithQueue;
7 use Illuminate\Contracts\Queue\ShouldQueue;
8
9 class LogMessageWhenListCreated
10 {
11
12     public function __construct()
13     {
14         //
15     }
16
17     public function handle(ListWasCreated $event)
18     {
19         //
20     }
21 }

```

The `handle` method will contain the logic you'd like to execute when the event is triggered. Notice how the `ListWasCreated` event object is passed into the method by default. You'll use this object to access the `List` object assigned in the handler's constructor. Update the constructor to look like this:

```

1 public function handle(ListWasCreated $event)
2 {
3     \Log::info("LIST CREATED {$event->list->name}");
4 }

```

Again, we're just executing a trivial log message to demonstrate that indeed the `List` object is made available to the listener. You could instead send an e-mail, notify a web service, or do whatever else you please here.

Associating the Events

With the event and event listener created, it's time to associate them so Laravel knows to trigger the event logic when the listener detects the desired action. You'll do so within `app/Providers/EventServiceProvider.php`. Open this file and you'll find the following protected `$listen` attribute:

```
1 protected $listen = [  
2     'event.name' => [  
3         'EventListener',  
4     ]  
5 ];
```

The lone entry in this array is just an example, so you can go ahead and replace it with our new event binding:

```
1 protected $listen = [  
2     'Todoparrot\Events\ListWasCreated' => [  
3         'Todoparrot\Listeners\LogMessageWhenListCreated'  
4     ]  
5 ];
```

After saving these changes you're ready to integrate the event into your application.

Integrating the Event Into Your Application Flow

Triggering an event is incredibly easy, done using the `Event::fire` method. You'll pass a new instance of the event handler you'd like to execute into this method, and inside it pass the object you'd like its constructor to receive. Here's an example in which the `ListWasCreated` handler is executed within the `Lists` controller's `store` action:

```
1 use Event;  
2 use Todoparrot\Events\ListWasCreated;  
3  
4 ...  
5  
6 public function store(ListCreateFormRequest $request)  
7 {  
8  
9     ...  
10  
11     Event::fire(new ListWasCreated($list));  
12  
13 }
```

With the event handler in place, the next time a new list is created you'll find a log message similar to the following has been added to the log file:

```
1 [2015-06-29 18:59:31] local.INFO: LIST CREATED
2 Groceries for Dinner
```

Binding Multiple Listeners to an Event Handler

It's certainly possible that you might wish to bind multiple listeners to an event handler. For instance, you might wish to both receive an e-mail and write a log message whenever a new list is created. Doing so is trivial; just create all of the necessary event listeners and then bind them like so inside `EventServiceProvider.php`:

```
1 protected $listen = [
2     'Todoparrot\Events\ListWasCreated' => [
3         'Todoparrot\Listeners\LogMessageWhenListCreated',
4         'Todoparrot\Listeners\LogAnotherMessageWhenListCreated'
5     ]
6 ];
```

Binding Events to the Model Lifecycle

If your sole intent is to trigger an event in conjunction with the creation, update, or deletion of a model, there's an even more streamlined approach you might consider. Whenever Laravel creates, updates, saves, deletes, or restores a model, it fires an event. You can piggyback atop these events, triggering your own custom events in kind. This is done within the `app/Providers/AppServiceProvider.php`'s `boot` method:

```
1 class AppServiceProvider extends ServiceProvider {
2
3     public function boot()
4     {
5         //
6     }
7
8     ...
9
10 }
```

For instance, if you'd like to trigger the `ListWasCreated` event handler when a new list is created, you can update the `boot` method like so:

```
1 use Event;
2 use Todoparrot\Events>ListWasCreated;
3 use Todoparrot\Todolist;
4
5 ...
6
7 public function boot()
8 {
9     Todolist::created(function ($list) {
10         Event::fire(new ListWasCreated($list));
11     });
12 }
```

After saving the changes, create a new list and you'll see that indeed another message is appended to the log.

Broadcasting Events

Laravel 5.1 introduced the ability to *broadcast* events to all users currently interacting with the application. This is in my opinion one of the most interesting Laravel features as it opens up countless new ways to communicate with your users in a really compelling fashion. For instance, the below screenshot presents an example of a broadcast notification sent out to all users after a user created a new list.



Broadcasting events

In this section I'll show you how to refactor the above list creation event to notify all users whenever a new list is created.

Configuring Broadcasting

To begin, we'll need to make a few configuration-related changes. All Laravel 5.1+ applications include a new configuration file named `config/broadcasting.php`. The default file looks like this:

```
1 <?php
2
3 return [
4
5     'default' => env('BROADCAST_DRIVER', 'pusher'),
6
7     'connections' => [
8
9         'pusher' => [
10             'driver' => 'pusher',
11             'key' => env('PUSHER_KEY'),
12             'secret' => env('PUSHER_SECRET'),
13             'app_id' => env('PUSHER_APP_ID'),
14         ],
15
16         'redis' => [
17             'driver' => 'redis',
18             'connection' => 'default',
19         ],
20
21         'log' => [
22             'driver' => 'log',
23         ],
24
25     ],
26
27 ];
```

Laravel currently supports three broadcasting drivers, including [Pusher](https://pusher.com/)¹⁵⁴, [Redis](https://redis.io/)¹⁵⁵ (which allows you to use Redis' pubsub capabilities to interact with a solution such as [Socket.io](https://socket.io/)¹⁵⁶), and a logging driver for testing purposes. Because this is for demonstration purposes I'll be using Pusher's free Sandbox plan. If you'd like to experiment with broadcasting, I suggest doing the same as it is the easiest to configure. To do so, head over to <https://pusher.com/>¹⁵⁷ and create a free account. After doing so, sign into your account and create a new app via the dashboard. You'll be provided with an application ID, public token, and secret token. Next, add the `PUSHER_KEY`, `PUSHER_SECRET`, and `PUSHER_APP_ID` configuration variables to your local `.env` file, and assign each the appropriate value. Don't forget you'll additionally want to make these configuration settings available to your production environment prior to deployment.

¹⁵⁴<https://pusher.com/>

¹⁵⁵<https://redis.io/>

¹⁵⁶<https://socket.io/>

¹⁵⁷<https://pusher.com/>

Next you'll need to install two new Composer packages, including `pusher-php-server` and `redis`. Add the following two lines to your `composer.json` file's `require` key:

```
1 "require": {  
2     ...  
3     "pusher/pusher-php-server": "~2.0",  
4     "redis/redis": "~1.0"  
5 },
```

Next, you'll need to configure a *queue listener*, because broadcasting depends upon queues for performance reasons. I haven't yet touched upon the topic of queues in the book (this is slated for a forthcoming chapter), so I'm not keen on including an introduction to the matter here and for the time being am going to instead point you to the appropriate [Laravel documentation](http://laravel.com/docs/master/queues)¹⁵⁸.

To enable Laravel's queue support you'll need to update `config/queue.php`, identifying the queue driver you'd like to use. By default it is set to `sync`, which means Laravel will just bypass the queue altogether and execute the event normally. Several drivers are supported, although for the purposes of this example I'm going to use the database driver since it is arguably the easiest to configure. After setting the `queue.php` file's default key to `database`, open the terminal and execute the following Artisan command from within your project's root directory:

```
1 $ php artisan queue:table  
2 Migration created successfully!
```

This creates a new database table which will be used to manage the queues. Next you'll want to run the newly created migration:

```
1 $ php artisan migrate  
2 Migrated: 2015_06_30_161852_create_jobs_table
```

Finally, you'll need to start Laravel's queue listener. Open a new terminal tab and execute the following Artisan command from your project directory:

```
1 $ php artisan queue:listen
```

With your broadcasting driver configured and listener initiated, we're ready to broadcast events!

Broadcasting Events

Broadcasting events to your application users is a two-step process. First, you'll need to update the appropriate event handler's `broadcastOn` method. Returning to `app/Events/ListWasCreated.php`, you'll find this method at the bottom of the class:

¹⁵⁸<http://laravel.com/docs/master/queues>


```
1 public function broadcastOn()  
2 {  
3     return [];  
4 }
```

To enable broadcasting, you'll need to make two modifications to this class. First, you'll need to implement the `ShouldBroadcast` contract which is imported at the top of the class file. Second, you'll update the `broadcastOn` method to identify the Pusher *channel* which will broadcast the event information. Pusher channels are analogous to television channels; you'll broadcast whatever content you'd like via a channel, and then “tune in” to that channel on the client-side. We'll call our channel `list-updates`:

```
1 use Illuminate\Contracts\Broadcasting\ShouldBroadcast;  
2  
3 class ListWasCreated extends Event implements ShouldBroadcast  
4 {  
5  
6     ...  
7  
8     public function broadcastOn()  
9     {  
10         return ['list-updates'];  
11     }  
12 }
```

With the event handler changes in place, you'll need to add the necessary client-side code to the application. If you're using Pusher this is rather straightforward and is described in [their documentation](#)¹⁵⁹. You'll begin by referencing the Pusher JS library in your project layout:

```
1 <script src="//js.pusher.com/2.2/pusher.min.js"></script>
```

Next, you'll create the JavaScript responsible for connecting to Pusher and retrieving the latest broadcasts. Believe it or not all that is required is a few simple lines of code. The following jQuery-enhanced snippet will connect to Pusher using your public key, subscribe to the `list-updates` channel, bind to the `ListWasCreated` event, and update a DIV named `pusher` with information about the newly created list name:

¹⁵⁹https://pusher.com/docs/javascript_quick_start

```
1 $( document ).ready(function() {  
2  
3     var pusher = new Pusher('YOUR_PUBLIC_PUSHER_KEY');  
4     var channel = pusher.subscribe('list-updates');  
5  
6     channel.bind('Todoparrot\\Events\\ListWasCreated', function(data) {  
7         $('#pusher').html('New list created: ' + data.list.name);  
8     });  
9  
10 });
```

Of course, you'll want to additionally add the pusher DIV to an appropriate location within your application layout. With these pieces in place, you're ready to begin broadcasting messages out to your users!

Troubleshooting Pusher

If your broadcasts aren't working as expected, beyond confirming you're using the correct public API key and ensuring the Pusher JS library is loading properly, consider adding the following JS code to your JavaScript client to confirm you are indeed connecting to the Pusher server:

```
1 pusher.connection.bind('connecting', function() {  
2     alert('Connecting to Pusher...');  
3 });  
4  
5 pusher.connection.bind('connected', function() {  
6     alert('Pusher connection successful');  
7 });  
8  
9 pusher.connection.bind('failed', function() {  
10     alert('Pusher connection failed');  
11 });
```

Additionally, be sure to monitor the Debug Console within the Pusher dashboard; this is supremely useful for determining whether Pusher is indeed receiving the broadcasts emitted from your event.

Summary

Events offer an elegant solution for executing important tasks alongside your domain logic without risk of polluting the code base. Once you begin using this powerful feature within your Laravel applications you'll wonder how you ever got along without it!

Chapter 12. Introducing Vue.js

I'd imagine most readers have seen *The Hangover*, the smash-hit comedy film in which a group of best friends go to Las Vegas for a bachelor's party and wind up getting in a little more than they bargained for. A major character is played by Zach Galifianakis, who in his role as Alan Garner manages to simultaneously play both protagonist and antagonist, gleefully adding equal doses of entertainment and chaos to the trip. Although often reviled and the subject of scorn, by the end of the movie Alan's unique talents play a pivotal role in his companions' successful outcomes.

If his role were instead to be played by a programming language, JavaScript's agent would undoubtedly receive the first call. Wildly annoying yet indispensable to the modern web, it's practically inconceivable a successful web application could be implemented without at least a smidgen of the language showing up somewhere in the code base.

Yet despite, or perhaps because of, the language's sordid history, JavaScript adoption continues to hit new highs thanks to an incredible amount of effort being put into not only building even more palatable solutions, but even going so far as to make JavaScript the star of the show. These days you'll find JavaScript frameworks such as AngularJS and Ember playing key roles at companies such as Walgreens, The Weather Channel, Costco, Square, Twitch, and Netflix¹⁶⁰. The popular Node.js runtime has been embraced by The New York Times, PayPal, and LinkedIn¹⁶¹. More recently, Facebook released React, a JavaScript library which plays a major role in powering Facebook and Instagram, and even the native mobile applications.

While all of these JavaScript technologies can easily be integrated into a Laravel application, still another has become the de facto solution for the Laravel community. It's called **Vue.js**¹⁶², and like Laravel, Vue.js really seems to have been developed with the goal of maximizing users' productivity and efficiency while embracing no-nonsense, easily understandable syntax. In this chapter you'll learn all about Vue.js as we update TODOParrot to include several new dynamic features.



Throughout this chapter we'll modify the TODOParrot application to create a more fluid list management interface. These changes are found in the `vue-integration` branch on GitHub.

Installing Vue.js

Like most popular JavaScript libraries, Vue.js can be installed in several different fashions. When working on Vue.js features you'll want to download and locally host the development version,

¹⁶⁰See <https://www.madewithangular.com/> and <http://emberjs.com/ember-users/>

¹⁶¹See <https://www.quora.com/What-companies-are-using-Node-js-in-production>

¹⁶²<https://vuejs.org/>

which can be downloaded from <https://vuejs.org/js/vue.js>. Place this file in your project's `public/js` directory (create the `js` directory if it doesn't exist), and then reference the file just above your project layout's closing `<body>` tag:

```
1 ...  
2 <script src="/js/vue.js"></script>  
3 </body>
```

As you'll see later in the chapter, the development version will log useful debugging messages to your browser's JavaScript console, which will help to quickly resolve beginner's mistakes.

In production you can also host the library locally, however for performance reasons you'll want to download and reference the minified version (available at <https://vuejs.org/js/vue.min.js>), which omits all debugging messages. Alternatively, you can use a CDN (content distribution network) such as Cloudflare's CDNJS. To reference Vue.js via CDNJS, add the following line just above your layout's closing `<body>` tag:

```
1 <script  
2   src="https://cdnjs.cloudflare.com/ajax/libs/vue/1.0.21/vue.min.js">  
3 </script>  
4 </body>
```

If you plan on using Vue.js in conjunction with Laravel to build a SPA (single page app), then you'll want to install it using NPM. We'll be using Vue.js for a less ambitious integration, and so rather than discuss NPM-related matters here I'll instead point you to the installation documentation:

<https://vuejs.org/guide/installation.html>.

Your First Vue.js Example

With the development version of Vue.js installed, let's get acquainted with the syntax by integrating a very simple example. Create a new file in `public/js` named `mycode.js` (the name doesn't really matter), and add the following code to it:

```
1 window.onload = function () {
2     new Vue({
3         el: '#greeting',
4         data: {
5             message: 'Welcome to TODOParrot'
6         }
7     });
8 }
```

This example instantiates a new instance of the `Vue` class, identifying a DOM element ID (`#greeting`), and assigning a string (`Welcome to TODOParrot`) to a variable named `message`. We're going to display this message within a page by creating an element having the ID `greeting`, and specifying where within that element the `message` variable should be rendered.

Incidentally, we're encapsulating the Vue.js-specific JavaScript within the event handler because we don't want the JavaScript to execute until the page DOM has completely loaded into the browser. Lacking this precautionary step, the code would run before the element associated with the `greeting` ID was available, causing an error. Be sure to keep your browser's JavaScript console open while developing Vue.js-driven features so you can easily keep tabs on these warnings. For instance the following screenshot displays the error you would see were you to not wrap the Vue code in the `window.onload` handler:



A Vue.js warning

If you're using jQuery (in conjunction with Bootstrap for instance), you can take advantage of jQuery's streamlined handler:

```
1 $(function() {  
2     var greeting = new Vue({  
3         el: '#greeting',  
4         data: {  
5             message: 'Welcome to TODOParrot'  
6         }  
7     });  
8 });
```

Next, return to your layout and add the following line after the reference to `vue.js`:

```
1 <script src="/js/mycode.js"></script>
```

Finally, open up one of the project views (it doesn't really matter which one; I used `lists/index.blade.php`) and add the following:

```
1 <div id="greeting">  
2     @{{ message }}  
3 </div>
```

Reload the page associated with the modified view and you'll see the message `Welcome to TODOParrot` has been dynamically added!

By default Vue.js uses the curly brackets to identify variables, which as you are by now well aware is identical to Laravel Blade's approach. Therefore if you're using Vue.js within a Blade-enabled view, you'll need to prefix `@` to the opening brackets, which will cause Blade to ignore that particular bracket enclosure when rendering the view. If you're not using Vue.js within a Blade-enabled view, then you can forego the `@` prefix.

This example was intended simply to familiarize you with Vue.js syntax fundamentals. We're of course going to do something much cooler with this powerful library, radically improving the TODOParrot list tasks manager to create a much more streamlined interface.

Refactoring the List Tasks Manager

The current TODOParrot list tasks manager is fairly boilerplate in terms of your traditional CRUD web application. To create a new list task, a standard HTML form is presented (generated using `LaravelCollective/HTML`). The user completes and submits the form, and the `Tasks` controller's `store` method is executed. Presuming the `TaskCreateFormRequest` form request doesn't detect any validation errors, the new task is associated with the list and added to the database before returning the user to the list's task listing. Pretty standard stuff, in other words.

While the interface is functional, when adding a series of tasks I find it easy to forget which task I just added and wind up adding the same task twice. To resolve this issue we're going to refactor the view associated with this form, displaying the list's tasks like presented in the following screenshot:

The screenshot shows a web browser window with the URL `dev.todoparrot.com:8002/lists/4/tasks`. The page has a dark header with the text "TODOParrot" on the left and navigation links "Your Lists", "About", "Hi, Jason Gilmore", and "Sign Out" on the right. The main content area is split into two columns. The left column, titled "Create a Task (Write Book)", contains a "Task Name" input field with the placeholder text "Task Name", a "Due Date" input field with the value "2016-04-25", a checkbox labeled "Completed?", and a green "Create Task!" button. The right column, titled "Your Tasks", contains a list of three tasks: "Write Chapter 1", "Write Chapter 2", and "Write Chapter 3", each with a blue underline.

The Refactored List Creation View

Further, in the interests of productivity we're going to update the logic so the user can enter many list tasks in rapid fire fashion, submitting each task via Ajax and automatically updating the right-side list without ever reloading the page. This feature requires several different steps, so we're going to just focus on one step at a time until everything is in place, beginning with retrieving the tasks on page load using Vue.js and a bit of Ajax.

Retrieving the Tasks

While you certainly could modify the Tasks controller's `create` action to additionally retrieve any existing tasks associated with the target list, and then pass the collection into the view, we eventually want the convenience of being able to dynamically update this task list when the user submits a new task through the form found in the above screenshot. To do so, we're going to want to load these tasks using a Vue.js-driven Ajax call. However, the core Vue.js library does not include Ajax-related features, and so you're going to want to integrate a Vue.js-plugin called *Vue Resource* (<https://github.com/vuejs/vue-resource>) which can generate HTTP requests such as GET, POST, and DELETE (among other features). This is done in an identical fashion to that used to integrate Vue.js; just reference the CDNJS-hosted library below the Vue.js import and you're done:

```
1 <script
2 src="https://cdnjs.cloudflare.com/ajax/libs/vue-resource/0.7.0/vue-resource.js">
3 </script>
```

With that complete, we're going to want to create a new endpoint for retrieving the tasks associated with a given TODOLIST ID. You can add this wherever convenient, however as applied to TODOParrot I suggest adding the method in the Tasks controller:

```
1 use Request;
2
3 ...
4
5 public function tasks($listId) {
6
7     $list = Todolist::findOrFail($listId);
8
9     $user = User::find(\Auth::id());
10
11     if ($user->owns($listId)) {
12
13         if (Request::ajax()) {
14             return response()->json(['tasks' => $list->tasks]);
15         }
16
17     }
18
19 }
```

Nothing found in this method should be new at this point, except for the following lines:

```
1 if (Request::ajax()) {
2     return response()->json(['tasks' => $list->tasks]);
3 }
```

This will return a JSON-formatted collection of tasks if the incoming request was made using Ajax. Since the action was created expressly to respond to Ajax-based calls, you don't necessarily need to wrap the return statement in the conditional, however I thought it an opportune time to demonstrate this Laravel feature since we'll be using it in a more practical fashion later in the chapter. The returned JSON will look like this:


```

1  {
2    "tasks": [
3      {
4        "id": 14,
5        "todolist_id": 4,
6        "name": "Write Chapter 1",
7        "due": "2016-05-15",
8        "done": 0,
9        "created_at": "2016-04-26 00:20:54",
10       "updated_at": "2016-04-26 00:20:54"
11     },
12     {
13       "id": 15,
14       "todolist_id": 4,
15       "name": "Write Chapter 2",
16       "due": "2016-05-20",
17       "done": 0,
18       "created_at": "2016-04-26 00:22:12",
19       "updated_at": "2016-04-26 00:23:16"
20     }
21   ]
22 }

```

If you're not familiar with JSON and this syntax looks intimidating, not to worry because as you'll soon learn Laravel and JavaScript will take care of the monotonous parsing for you. With the new action created, add a route to the `routes.php` file:

```

1  Route::get('lists/{id}/tasks', 'ListsController@tasks');

```

Next, we'll update the `mycode.js` script, replacing the initial example with the following code. You can find the entire script in the `vue-integration` branch's `public/js/mycode.js`, so to facilitate instruction I'm going to break the script into smaller pieces with accompanying explanation:

```

1  $(function() {
2      var formCreate = new Vue({
3          el: '#tasks',
4          data: {
5              listId: '',
6              tasks: []
7          },

```

As before we've instantiated a new instance of the `Vue` class, and identified a target element ID of `tasks`. The data object literal contains two properties: `listId` and `tasks`. The `listId` property will be used to manage the target list's primary key, which as you'll see has been embedded into the page using an [HTML5 data attribute](#)¹⁶³. The `tasks` property will manage the array of tasks returned via the Ajax call to the `Lists` controller's `tasks` action.

```

1 ready: function() {
2
3     // Retrieve the list ID via the form's data-id attribute
4     var taskForm = document.getElementById('task-creation-form');
5     this.listId = taskForm.getAttribute('data-id');
6
7     this.retrieveTasks();
8
9 },

```

The `ready` function is special to Vue.js, executed at a specific point in the instance lifecycle and after the target element (the element associated with the `tasks` ID, in our case) has been inserted into the document. Inside this function you'll execute anything that you'd like to occur immediately. In this example we're retrieving the value of the `data-id` attribute embedded into the page, which identifies the target list's primary key. Because I use the `Laravel Collective` package for forms generation, that `data-id` attribute is passed into `Form::open` like so:

```

1 {!! Form::open(
2     [
3         'route' => ['lists.tasks.store', $list->id],
4         'class' => 'form',
5         'id'     => 'task-creation-form',
6         'v-on:submit.prevent' => 'submitTask',
7         'data-id' => $list->id
8     ])
9 !!}

```

When rendered to the browser, the opening form tag will look like this:

¹⁶³https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Using_data_attributes

```

1 <form
2   id="task-creation-form"
3   class="form"
4   data-id="4"
5   accept-charset="UTF-8"
6   action="http://www.todoparrot.com/lists/4/tasks"
7   method="POST">

```

Returning to the `mycode.js` explanation, after retrieving this ID the `retrieveTasks` method is executed. This method is responsible for generating the HTTP request used to talk to the `Lists` controller's `tasks` action. You'll find the `retrieveTasks` method in the next section of `mycode.js`:

```

1 methods: {
2
3   retrieveTasks: function(e) {
4
5       this.$http.get('http://www.todoparrot.com/lists/'
6         + this.listId + '/tasks')
7         .then(function(response) {
8             this.tasks = response.data.tasks;
9         })
10        .catch(function(error) {
11            document.getElementById('task-list').innerHTML =
12              'Could not retrieve tasks';
13        });
14
15    }
16
17 }

```

The `methods` object defines any methods that you'd like to execute in conjunction with this Vue instance. So far our instance only consists of the `retrieveTasks` function, which as stated previously will retrieve the list of tasks. Note how we're referring to the `data` object literal's `listId` property (via `this.listId`), and then in the `then` promise we're assigning the Ajax response (`response.data.tasks`) to the `data` object literal's `tasks` property (via `this.tasks`).



I'm unavoidably referring to some intermediate JavaScript concepts in this section, including features such as "promises". It's difficult to know where to draw the lines in terms of introducing fundamental concepts as a precursor to presenting these sorts of examples without radically expanding the chapter, so at least for this release I'll refer you to the excellent Mozilla Developer Network's [JavaScript Guide](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide)¹⁶⁴ for more information about this crucial JavaScript features.

¹⁶⁴<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>

With the Vue instance complete, it's time to update the view. Open `resources/views/tasks/create.blade.php` and we'll add the HTML and Vue.js-specific markup which will result in a list's retrieved tasks being rendered to the view. I've already discussed the changes to the `<form>` element in order to make the list ID available, so we'll now focus on the markup used to display the tasks:

```
1 <div class="col-md-6">
2 <h2>Your Tasks</h2>
3
4 <div id="task-list">
5
6   <p v-for="task in tasks">
7     <a
8       href="/lists/@{{ listId }}/tasks/@{{ task.id }}/edit">@{{ task.name }}
9     </a>
10   </p>
11
12 </div>
13
14 </div>
```

Much of the above is just regular HTML markup except for these crucial lines:

```
1 <p v-for="task in tasks">
2   <a
3     href="/lists/@{{ listId }}/tasks/@{{ task.id }}/edit">@{{ task.name }}
4   </a>
5 </p>
```

Recall we assigned the returned tasks in `mycode.js` to the Vue instance's data object literal's `tasks` property. This array is made available to the view, and can be iterated over using Vue.js' `v-for` directive. The `v-for` directive operates similarly to Laravel's `@foreach` directive, iterating over each item in the array until all elements are exhausted. The looping block is defined by the element in which it is defined, so in this case the `</p>` tag closes the looping block. Inside that block you can see we're creating a hyperlink, rendering the data literal's `listId` property, and the current array element's `id` and `name` properties.

Once rendered, the output HTML will look like this:

```

1 <p>
2   <a href="/lists/2/tasks/27/edit">Write Chapter 1</a>
3 </p><p>
4   <a href="/lists/2/tasks/28/edit">Play video games and procrastinate</a>
5 </p><p>
6   <a href="/lists/2/tasks/29/edit">Write Chapter 2</a>
7 </p>

```

With the revised Vue instance and updated HTML in place, you should be able to reload the `create.blade.php` view in your browser and see a list of tasks associated with the current list!

Submitting the Task Creation Form via Ajax

With the list task now dynamically rendered alongside the `create.blade.php` view form, let's update the form so all new tasks are added to the database via Ajax, eliminating the need for a page refresh. Let's start by modifying the existing Tasks controller's `store` method. Believe it or not the bulk of the existing method will work in conjunction with both Ajax- and non-Ajax driven requests! All we need to do is determine whether the incoming request is Ajax-based in order to return a JSON-based response instead of redirecting the user. To really illustrate just how little code requires modification, I've highlighted the relevant changes in the following snippet:

```

1 public function store($listId, TaskCreateFormRequest $request)
2 {
3
4     $user = User::find(\Auth::id());
5
6     if ($user->owns($listId)) {
7
8         $list = Todolist::findOrFail($listId);
9
10        $task = new Task([
11            'name' => $request->get('name'),
12            'due' => $request->get('due'),
13            'done' => true ? $request->get('done') == 'true' : false
14        ]);
15
16        $task = $list->tasks()->save($task);
17
18        if ($request->ajax()) {
19            return response()->json(['message' => 'New task added!']);
20        } else {
21            return \Redirect::route('lists.show', array($list->id))

```

```

22         ->with('message', 'Your task has been created!');
23     }
24
25     } else {
26
27         if ($request->ajax()) {
28             return response()->json(['message' => 'Authorization error']);
29         } else {
30             return \Redirect::route('home')
31                 ->with('message', 'Authorization error');
32         }
33
34     }
35
36 }

```

With this updated store method in place, let's return to `mycode.js` and add the JavaScript necessary to process the form. In order to minimize confusion, I'm only going to include the code necessary to submit the form via Vue.js, but bear in mind the final `mycode.js` implementation found in the `vue-integration` branch includes the code used to both retrieve the list tasks and submit the form. As before, I'll separate each part of the script with commentary:

```

1  $(function() {
2      var formCreate = new Vue({
3          el: '#tasks',
4          data: {
5              ajaxRequest: false,
6              taskInfo: {
7                  name: '',
8                  due: '',
9                  done: ''
10             },
11             listId: '',
12             tasks: [],
13             errors: []
14         },

```

The data literal's `ajaxRequest` property is used to determine the state of a user notification element found in `create.blade.php`. That element looks like this:

```
1 <span id="ajax-response" v-if="ajaxRequest">Please Wait...</span>
```

The `v-if` directive is a really convenient Vue.js feature which can be used to conditionally display data. When the `ajaxRequest` property is `false`, the element content (“Please Wait...”) will be hidden; when set to `true`, the element content will appear. You’ll see how the `ajaxRequest` property is toggled in just a moment.

Returning to `mycode.js`, the `taskInfo` property literal is used to manage a task object. The form contains three fields used to represent a task: `name`, `due`, and `done`. This `taskInfo` property will bind to those fields, and be conveniently passed into the Ajax request for transmission to the server.

The `errors` array will be populated with any errors returned from the Laravel form request. You’ll see how this is populated in just a moment.

```
1 ready: function() {
2
3     Vue.http.headers.common['X-CSRF-TOKEN'] =
4     document.querySelector('#_token').getAttribute('value');
5
6 },
```

Recall from Chapter 5 that Laravel requires a CSRF token to avoid cross-site scripting attacks. However when using Ajax we’re not going to use the standard request/response process and so will need to ensure the generated token is passed along via the HTTP headers. This is done in the `ready` function by retrieving the token via JavaScript’s `querySelector` method (or via a more convenient solution such as the selectors available through jQuery). Incidentally, you can ensure the CSRF token is available within the view by adding the following `meta` tag to your layout header:

```
1 <meta id="_token" value="{{ csrf_token() }}">
```

Next up is the `submitTask` function, which you’ll find in the Vue.js instance’s `methods` object:

```
1 submitTask: function(e) {
2
3     this.ajaxRequest = true;
4
5     this.$http.post('http://www.todoparrot.com/lists/'
6                   + this.listId + '/tasks', this.taskInfo)
7     .then(function(response) {
8         document.getElementById('ajax-response').innerHTML
9         = response.data.message;
10        this.tasks.push(this.taskInfo);
```

```

11         this.taskInfo = { name: '', due: '', done: '' };
12     })
13     .catch(function(error) {
14
15         this.errors = error.data;
16         document.getElementById('ajax-response').innerHTML = "Errors:";
17
18     });
19 },

```

The function begins by setting the `ajaxRequest` property to `true`. If you recall from the earlier discussion this will cause the `v-if` directive to display the “Please Wait...” message, something we’ll want to do in order to provide the user with visual indication that the application is working.

Next up the POST request is generated. The URL is constructed by passing the list ID into the URL, and the `taskInfo` payload is sent to the server. Remember we’re going to bind that `taskInfo` property to the form fields, which will result in the property *automatically* being populated with whatever values the user enters into the fields. This is truly one of the most beautiful aspects of Vue.js, because you can concentrate on building your application features rather than worrying about tedious plumbing.

When the response is returned, we’re going to update the element associated with the `ajax-response` ID with a status message (returned from the `Tasks` controller’s `store` method), add the task to the `tasks` property (which will cause the tasks list to *automatically* be updated), and then clear out the `tasks` property, which will *automatically* clear out the form fields.



Notice I’ve made regular use of the word *automatically* in recent paragraphs? This is because like Laravel, Vue.js removes a lot of the monotony otherwise incurred when building dynamic web applications!

If an error occurs while processing the data (such as the form request identifying an invalid form field value), the `errors` property will be populated and output to the view using the following syntax:

```

1 <ul v-for="(index, item) in errors">
2   <li>@{{ item }}</li>
3 </ul>

```

Finally, let’s update the form markup. In the interests of space I’m only going to present the relevant parts of the form, however as always you can find the entire form in the `vue-integration` branch’s `resources/views/tasks/create.blade.php` file. Let’s begin with the `Form::open` statement:


```

1  {!! Form::open(
2      [
3          'route' => ['lists.tasks.store', $list->id],
4          'class' => 'form',
5          'id'     => 'task-creation-form',
6          'v-on:submit.prevent' => 'submitTask',
7          'data-id' => $list->id
8      ])
9  !!}

```

We already discussed the `data-id` attribute (used to identify the target list's primary key). The other attributes are already quite familiar to you, except for the following:

```

1  'v-on:submit.prevent' => 'submitTask',

```

This Vue.js syntax will cause the `submitTask` method (found in `mycode.js`) to execute when the form is submitted. You can additionally prevent the standard form submission process from completing by canceling the event using the `.prevent` event modifier. This is the same as adding the following line to the `submitTask` function:

```

1  e.preventDefault();

```

Next up you'll find the Task Name, Due Date, and Completed fields. For the sake of space I'm only going to include the first, but the same concept should be applied to all three fields:

```

1  ...
2
3  <div class="form-group">
4      {!! Form::label('Task Name') !!}
5      {!! Form::text('name', null,
6          [
7              'v-model' => 'taskInfo.name',
8              'class' => 'form-control',
9              'placeholder' => 'Task Name'
10         ]) !!}
11  </div>
12
13  ...

```

The `v-model` attribute is assigned to `taskInfo.name`. Recall from the earlier discussion that we wanted to bind the data literal's `taskInfo` property to the form fields. This is precisely how that important step is accomplished! It's so incredibly powerful, yet easy to implement.

With the form updated, reload the page and try submitting a new task. The record should be added to the database, and the list of tasks found on the page updated!

Where to From Here?

Despite being barely two years old, Vue.js is *packed* with features and this short chapter hardly does the topic justice. I'll be rapidly expanding this material in the coming weeks, but if you'd rather not wait I suggest checking out the following resources:

- Vue.js Examples (<http://vuejs.org/examples/>): The official Vue.js website hosts about a dozen real-world examples, including a grid component, live Markdown editor, and a HackerNews clone.
- Full Stack Radio Evan You Interview (<http://www.fullstackradio.com/30>): Vue.js creator Evan You talks about the inspiration beyond Vue.js, and offers insights into how the library compares to other solutions such as React and AngularJS.

Summary

Vue.js offers an incredibly powerful solution for taking your application's user interface to the next level, yet in my experience allows you to do so by investing a fraction of the time and effort required of competing solutions.

Appendix A. Deploying Your Laravel Application to DreamHost



This appendix was released on August 10, 2015 and should be considered a beta release. I'm concerned Windows users in particular might run into a few issues that are not discussed here, therefore if you run into any issues whatsoever (whether on Windows or not), please do e-mail me at wj@wjgillmore.com so I can update the material accordingly.

Sooner or later, you're going to want to make your Laravel creation available to the world. Fortunately, there are hundreds, if not thousands of web hosting providers perfectly capable of hosting your Laravel application. I've successfully deployed Laravel projects to multiple hosting providers, including Heroku and DreamHost. In Chapter 8 I even explained how to deploy your project to Heroku. However, Heroku isn't for everybody, and it can get pretty expensive quickly if your application requires significant resources. So in this Appendix I thought I'd offer instructions regarding deploying your application to DreamHost.

DreamHost is without a doubt my favorite shared web hosting provider. I've been a customer for almost 10 years now and have rarely experienced a problem with the service, and never anything significant. It's also very inexpensive, costing just \$8.95 for an entry-level hosting solution that is perfectly suitable for hosting a Laravel application.

In this chapter I'll guide you through the steps required to deploy your project to DreamHost. We'll use the popular open source server automation and deployment software [Capistrano](http://capistranorb.com/)¹⁶⁵, which can be a bit confusing at first but once you understand how it works you'll wonder how you ever lived without it. In addition to configuring Capistrano we'll need to complete some one-time server-side configuration tasks. Incidentally, although these instructions are indeed specific to DreamHost, chances are they'll be easily adapted for other shared hosting providers, although as always keep in mind your mileage may vary. :-)



If you're not already a DreamHost subscriber, and would like to sign up, use the code `EASYLARAVELBOOK` when signing up and you'll receive \$50 off your purchase, meaning your first year monthly hosting fees will be only \$5.78!

¹⁶⁵<http://capistranorb.com/>

Deploying Your Project to DreamHost

Admittedly, configuring DreamHost (or frankly, many other shared hosting providers) to play with Capistrano can be a bit time-consuming the first time around, however many of these tasks only need to be completed once after which you'll be able to deploy subsequent projects in less than five minutes flat. I'll spend the rest of the chapter guiding you through both the one-time and repeating tasks, doing so in what I believe is the most natural order of completion. The tasks will be presented in this order:

1. **Managing Your Project Using Version Control:** Although this task could conceivably be presented later in the chapter, I wanted to discuss it immediately in case you aren't yet managing your project under version control and therefore needed to do some additional footwork (which I'll discuss in the later section).
2. **Configuring Your Domain:** Your application will of course be associated with some domain name. In this section I'll show you how to configure your DreamHost account to host that domain name and it's various files and other resources.
3. **Updating Your Domain Registrar DNS:** When users around the world navigate to your domain name, you'll want the DreamHost servers to respond in kind and serve up your application. In this section you'll learn how to configure your domain registrar's DNS settings to ensure this happens.
4. **Configuring Your DreamHost User Account:** Your default DreamHost user does not have shell access, however access can be easily enabled via the administration panel. In this section I'll show you how to do so.
5. **Updating the DreamHost PHP CLI Version:** When deploying project updates to the DreamHost server it's likely various Artisan commands will execute to carry out tasks such as database migrations. To do so you'll want your DreamHost account to have access to a recent PHP version for use on the command-line. In this section I'll show you how this is (easily) done.
6. **Installing Composer on DreamHost:** Each time your project deploys you'll want to ensure it has access to the latest dependencies. Because these dependencies are managed using Composer, you'll want to install Composer on DreamHost. As with updating the PHP CLI version this is easier than you think, and in this section you'll learn how.
7. **Configuring Passwordless Login:** Although not a requirement, being required to manually authenticate with DreamHost every time you'd like to deploy code updates is going to get tedious fast. By configuring passwordless login using SSH keys you can avoid this inconvenience altogether without compromising your account or server security. In this section I'll discuss the steps necessary to configure and test passwordless login.
8. **Installing and Configuring Capistrano:** With the server configured we'll install Capistrano and update your application's newly created Capistrano configuration files. In this section we'll work through the steps necessary to install and configure Capistrano for deployment purposes.

9. **Deploying Your Application:** The moment of glory has arrived! With the server and Capistrano configured, we'll deploy your application to the DreamHost servers. You'll also learn how to easily and seamlessly rollback a deployment to the previous version.

Managing Your Project Using Version Control

While not strictly required for deployment purposes, placing your project under version control is just an all around very wise idea. In fact, version control is so crucial to the success of your project that if you're not yet using it, I suggest putting off this chapter and instead learning more about the topic and the various version control solutions, including most notably [Git](#)¹⁶⁶. You can find a succinct explanation of the merits of version control on the [Tower website](#)¹⁶⁷.

Placing your project under version control has a secondary benefit in regards to Capistrano in that it will help you to deploy your projects much more quickly than other available solutions. When configured accordingly, Capistrano will clone the repository on the production server and then use Git's native capabilities to only retrieve the latest changes which are then incorporated into the server-side repository. Through some subsequent magic that I'll discuss later in the chapter, this updated repository is then exported (or "archived") to a directory which will be subsequently used as the new website version.

Git is the most popular version control solution today, what I use on a daily basis, and is what Capistrano supports natively, so I'll use it for the relevant examples and discussion. I'll presume you've already initialized your project as a Git repository (e.g. `git init`) and have made at least one commit representative of the project version you'd like to deploy.

Additionally, although this isn't strictly necessary, it makes a lot of sense to be managing your project centrally on a project hosting service such as [BitBucket](#)¹⁶⁸ or [GitHub](#)¹⁶⁹. If so, Capistrano can be configured to refer to the centrally hosted repository rather than your local repository, which can be very advantageous given multiple team members are probably committing changes to your project. If you would like to manage your project privately but would rather not incur a monthly expense just yet, keep in mind BitBucket offers a free account tier which includes private repositories. I happen to use both BitBucket and GitHub, and find both services to be indispensable. Later in the chapter I'll show you how to configure Capistrano to interact with your GitHub repository, although these instructions could easily be adapted for any other Git project hosting service.

Configuring Your Domain

Your deployed Laravel application will logically be accessible via a domain name. To do so, you'll need to identify this domain within the DreamHost panel, and then update your domain name registrar to point the domain to DreamHost's DNS servers. Let's begin with the former task. If you

¹⁶⁶<https://git-scm.com/>

¹⁶⁷<http://www.git-tower.com/learn/git/ebook/mac/basics/why-use-version-control>

¹⁶⁸<https://bitbucket.org/>

¹⁶⁹<http://github.com>

wanted to host your domain name `example.com`, you'll login to your DreamHost administration panel <http://panel.dreamhost.com>¹⁷⁰ and navigate to Domains > Manage Domains. There you'll find a button titled 'Add Hosting to a Domain / Sub-Domain'. Click that button and you'll be presented with the interface found in the following screenshot:

Fully Hosted
(Upload your site to our servers and we'll serve it up!)

Domain name

Domain to host:
sub-domains are okay!

Do you want the www in your URL? [\(what's this?\)](#)

☐ Leave it alone: Both `http://www.example.com/` and `http://example.com/` will work.

☒ Add WWW: Make `http://example.com/` redirect to `http://www.example.com/`

☐ Remove WWW: Make `http://www.example.com/` redirect to `http://example.com/`

Users, Files, and Paths

Run this domain under the user:

This affects which FTP account will have access to the domain, as well as what user PHP and CGI scripts will run as.

Web directory:

Logs directory:
(can't be changed)

Web Options

PHP mode: [\(what's this?\)](#)

Automatically upgrade PHP: ☒
Keeps your site up to date with DreamHost's recommended PHP version.

Extra Web Security? ☒
(highly recommended - [what's this?](#))

BETA! Page Speed Optimization? ☐
[\(what's this?\)](#)

PHP XCache Support: ☐ requires a VPS
[\(what's this?\)](#)

Passenger (Ruby/NodeJS/Python apps only): ☐
[\(what's this?\)](#)

Hosting a Domain

In order to properly deploy your application in the fashion I outline in this chapter, it is *very important* for you to properly complete this form in the manner I describe here. I'll define the purpose of each field, and explain how you should define the associated value:

- **Domain to host:** Here you'll identify the domain name. Note you'll just specify it as `example.com` and not `www.example.com`.
- **Do you want www in your URL?:** This decision is entirely up to you, however keep in mind that for search engine optimization purposes you'll want to refer to the site exclusively as `www.example.com` or `example.com`, so I would suggest either selecting Add WWW or Remove WWW, and sticking with whichever convention you choose.

¹⁷⁰<http://panel.dreamhost.com>

- **Run this domain under the user::** For security reasons you can create a new owner for each hosted domain. This account would own the hosted files, and you would use that account for SSH'ing and SFTP'ing into the server to manage the domain files. For reasons of convenience I tend to manage my sites using the same user, however keep in mind that in doing so if that account were to be compromised then the attacker would have control over all of your hosted domain files.
- **Web directory:** This field is particularly important so pay very close attention. When you identified the domain in the Domain to host field, this web directory path was automatically updated to read like `/home/username/example.com`. Change this to read `/home/username/example.com/current/public`. Triple-check that you typed it exactly as I specify (replacing `example.com` with your domain of course), because if it is not exactly as specified you will run into problems later in this chapter.
- **Logs directory:** This identifies the location of your log files. It can't be changed.
- **PHP mode:** At the time of this writing DreamHost will by default use PHP 5.4 FastCGI to run your PHP-powered websites. I suggest changing this to the very latest available FastCGI version, which again at the time of this writing is PHP 5.6 FastCGI.
- **Automatically upgrade PHP:** I suggest enabling this option so you can always take advantage of the latest PHP features.
- **Extra web security:** This option enables the Apache `mod_security` module, which can protect your website from common exploits such as cross-site scripting and remote execution.
- **Page speed optimization:** This option enables the Apache `mod_pagespeed` module, which automatically applies optimization techniques such as image compression and CSS/JavaScript concatenation and minification. I recommend enabling this feature.
- **PHP XCache Support:** This feature can improve the performance of your PHP applications by caching compiled PHP code. It is however only available to DreamHost's Virtual Private Server customers.
- **Passenger (Ruby/NodeJS/Python apps only):** This option only applies to Ruby, NodeJS, and Python applications so leave it disabled.

You can safely disregard the options found below the above, although I certainly encourage you to investigate their utility. Once you've completed the aforementioned form fields, press the `Fully host this domain` button to complete the process.

Updating Your Domain Registrar DNS

After pressing the `Fully host this domain` button the ensuing page will instruct you to update your domain registrar's DNS settings to point the domain name to DreamHost's DNS servers. The interface for doing so varies according to the registrar, but they are all pretty straightforward. For instance, the below screenshot presents an example of Namecheap's DNS editor for one of my domains:

Modify Domain: easyreactbook.com

[Related Help](#)
[Related Video](#)

CHANGE EXISTING DOMAIN NAME SERVER INFORMATION

You can change the existing domain name server (DNS) information below. This option is typically used when you change your web hosting company etc. Please note that it will take up to 24 hours for the changes to take effect.

- ☐ Use Namecheap Hosting DNS Servers
☒ Specify Custom DNS Servers (Your own DNS Servers)

- | | | |
|----|--|---|
| 1. | <input type="text" value="ns1.dreamhost.com"/> | * |
| 2. | <input type="text" value="ns2.dreamhost.com"/> | * |
| 3. | <input type="text" value="ns3.dreamhost.com"/> | |
| 4. | <input type="text"/> | |
| 5. | <input type="text"/> | |

[Add More Nameservers](#)

Please note that you are also free to [Transfer the DNS back to us](#) * to take advantage of our free features like e-mail & url forwarding, dynamic dns etc.

[Save Changes](#)

Namecheap's DNS Editor

After saving these changes, it will take anywhere from a few minutes to a few hours for the updates to *propagate* to the world's root DNS servers. Once this happens, requests made to your domain will be ultimately handled by DreamHost. Of course, because you haven't yet deployed the application, should these changes propagate before you complete the remaining steps discussed in this chapter then users will be greeted with a default DreamHost splash page.

Incidentally, If you haven't yet purchased a domain name, I've been using [Namecheap](https://www.namecheap.com/)¹⁷¹ for recent domain name purchases. They seem much more sane than some of the other domain name registrars out there, and offer a really nice management interface. If you search for "namecheap coupons" you'll find they offer a special coupon good every month which will save you some money on the purchase.

Configuring Your DreamHost User Account

Capistrano will interact with your server by way of a user account and (preferably) passwordless login (discussed in the later section, "Configuring Passwordless Login"). When configuring your domain you were prompted to identify a user when setting the field Run this domain under the user. Whether you used the default user account or created a new account, you'll need to update this user to enable shell access. This is done by signing into the DreamHost panel and navigating to

¹⁷¹<https://www.namecheap.com/>

Users > Manage Users, clicking the Edit link associated with the desired user, and then setting the User Type field to Shell user (see below screenshot).

Editing User: [redacted] (on [redacted])

Full Name: [redacted]
[learn more](#)

Home Directory: [redacted]

Enhanced security? ☒
[learn more](#)

User Type: ([Learn more about enabling shell access](#))
☐ FTP user - allows login via FTP for file transfers only.
☐ SFTP user - allows login via SFTP (SSH file transfer) for file transfers only.
☒ Shell user - allows login via SSH (secure shell) for command-line access, as well as SFTP.
Shell Type:
Disallow FTP?: ☐

CPU Reporting: ☐
Enable user CPU usage statistics
(unavailable on DreamHost PS)
[learn more](#)

New Password:
leave blank for no change (8-31 characters)

New Password Again:

Pick a password for me ☐
revealed on next page

[Save Changes](#)

Updating a User Account

Once done, you'll be able to SSH into your DreamHost server by supplying your account username and the domain name you had configured over the course of the previous two sections:

```
1 $ ssh wjgilmore@example.com
2 wjgilmore@example.com's password:
3
4 No mail.
5 Last login: Mon Aug 10 13:05:04 2015 from 70.60.34.46
```

With this step complete, it's time to move on to the next section!

Updating the DreamHost PHP CLI Version

Recall from the earlier section, “Configuring Your Domain”, you update the domain’s PHP mode option use the latest available PHP version. Indeed this will cause the web server to use that version for running your application, however the PHP CLI (command-line interface) will confusingly continue to refer to the default PHP version (5.3.6 at the time of this writing). You’ll want to update the CLI to instead use the same version you specified via the PHP mode setting. This is incredibly easy to do, requiring just a few steps. Begin by SSH’ing into your DreamHost account and creating a directory named bin inside your account home directory:

```
1 $ mkdir -p ~/bin
```

Next you'll create a symbolic link inside this directory which points to the same binary version you referenced in the PHP mode setting. I used PHP 5.6 above and so will refer to that version in the below example:

```
1 $ ln -s /usr/local/bin/php-5.6 ~/bin/php
```

Next we want to be able to just execute php anywhere within the account directory tree and result in the linked PHP binary being executed. To do so, add this newly created directory to your system path and make sure that updated path is available for both login and non-login shells:

```
1 $ echo "export PATH=~/bin/:$PATH" >> ~/.bash_profile
2 $ echo "export PATH=~/bin/:$PATH" >> ~/.bashrc
```

Next you'll want to enable PHP's phar (PHP Archive) extension. Composer is distributed in Phar format, which if you've ever used Java is akin to Java's JAR format. It is just a way to easily distribute PHP-powered applications such as Composer. Don't worry too much about the details here; you just need to carry out a few simple steps to make Phar available to your PHP CLI version. Begin by creating a directory for housing a custom configuration file which will be used by your PHP CLI. Again I'm using a version-specific directory name here, which you might want to change if you're using a PHP version newer than 5.6:

```
1 $ mkdir -p ~/.php/5.6
```

Next, we'll use a command-line convenience to create a file named phprc inside this newly created directory, and add the line `extension = phar.so` to it. This tells the PHP CLI to load the `phar.so` extension:

```
1 $ echo "extension = phar.so" >> ~/.php/5.6/phprc
```

Finally, you can confirm Phar is enabled by running the following command:

```
1 $ php -m | grep Phar
2 Phar
```

With Phar installed, it's time to install Composer!

Installing Composer on DreamHost

Just as you use Composer locally to install and manage your project dependencies, so will you need it available on our DreamHost for the same purposes. Fortunately, once Phar has been enabled it is very easy to install and run Composer on your DreamHost server. Begin by downloading the Composer installer:

```
1 $ curl -s https://getcomposer.org/installer | php -- --install-dir=~/.bin
```

Once complete, all you need to do is change the permissions of the `composer.phar` file which has been placed in the `bin` directory you created when updating the PHP CLI version:

```
1 $ chmod u+x ~/.bin/composer.phar
```

For convenience reasons I prefer to remove the unnecessary `phar` extension from the file:

```
1 $ mv ~/.bin/composer.phar ~/.bin/composer
```

Once complete you should be able to run Composer in the very same way you do on your development machine:

```
1 $ composer --version
2 Composer version 1.0-dev
```

Configuring Passwordless Login

Next up we'll configure passwordless login. This means you'll be able to SSH into your DreamHost account without supplying a password, like this:

```
1 $ ssh wjgilmore@www.wjgilmore.com
2
3 No mail.
4 Last login: Mon Aug 10 12:56:48 2015 from 1.2.3.4
5 [~]:
```

Notice how I wasn't prompted for a password? This can be incredibly convenient, particularly when you are regularly SSHing into several different servers on a regular basis, each with a distinct account password. Additionally, it's useful for Capistrano deployments because it saves you the hassle of having to enter your DreamHost account username and password every time you deploy. You can do so by providing DreamHost with your public SSH key, and then use key-based authentication when connecting to the server.

If you're not familiar with key-based authentication but seem to recall having generated a pair of public and private SSH key pair at some point, you can easily check by opening a terminal and navigating to your local home directory. List your directory contents, looking for a directory named `.ssh`. If it exists, look inside this directory for files that look like this:

```
1 id_rsa.pub
2 id_rsa
```

If these exist, then indeed you have already created a key pair. If this `.ssh` directory does not exist, or if the directory is empty, you'll need to generate the keys. If you're on Mac OSX or Linux, this is incredibly easy. Just execute the `ssh-keygen` command as demonstrated below (replacing the e-mail address placeholder with your actual address), and then follow along with the subsequent prompts:

```
1 $ ssh-keygen -t rsa -b 4096 -C "YOUR_EMAIL_ADDRESS_HERE"
2 Generating public/private rsa key pair.
3 Enter file in which to save the key
4 (/Users/wjgilmore/.ssh/id_rsa):
5 Enter passphrase (empty for no passphrase):
6 Enter same passphrase again:
7 Your identification has been saved in
8 /Users/wjgilmore/.ssh/id_rsa.
9 Your public key has been saved in
10 /Users/wjgilmore/.ssh/id_rsa.pub.
```

Note you'll be prompted to supply what's referred to as a *passphrase*. This is just a fancy word for password. This passphrase can be as long as you desire, and if you plan on using passwordless login to any degree then I suggest choosing a sufficiently long and complicated passphrase. This is because even if a third-party were to obtain your private key (`id_rsa` in the above example), they will not be able to masquerade as you without knowing that passphrase! This is because every time the keys are used, you'll be required to enter the passphrase.

At first glance it seems like this really isn't any more convenient than manually signing in via SSH in the first place, however you can avoid that by making your keys available to what's referred to as the *SSH agent*. This is a program that will manage and unlock your keys for you, requiring you to only enter your passphrase once per login. These days, operating systems such as OSX even go the extra step of optionally adding the passphrase to your system *keychain*, meaning you won't ever be bothered with having to enter the passphrase again (although you still must keep the passphrase in a safe place should you ever wish to use the SSH keys on another machine). To add your keys to the agent, execute the `ssh-add` command:

```
1 $ ssh-add ~/.ssh/id_rsa
2 Enter passphrase for ~/.ssh/id_rsa:
```

Once OSX / Linux readers have completed the above step, skip ahead to the section, "Configuring DreamHost for Passwordless Login".

If you're on Windows, per usual things aren't so straightforward but is nonetheless easily achievable. I am however going to take the rare easy way out of further explanation and point you to this

excellent set of instructions on the [Joyent website](http://bit.ly/1LhUhPj)¹⁷² because I haven't used Windows in years and would therefore prefer readers have access to what seems to be a regularly updated set of instructions. Once complete, return to the book and go to the section, "Configuring DreamHost for Passwordless Login".

Configuring DreamHost for Passwordless Login

Once you've configured your SSH key pair, you'll need to provide DreamHost with your public key. Your public key is found in the file having the extension `.pub` in your `.ssh` directory. If you open this file you'll see that it contains an absurdly long random-looking string like this:

```
1  ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCAQCg4QD5XAvxFy9KgAf815gnpLYW
2  Z13c1eKDM2goER9wC7KgNQuXv7WRP12gnPzHHMaFSMKFfL0Qyes6v491hBVmXYvQM
3  Yr1cAAEdTdJ2A1pMkrnTGqr5SeNZ0XdI6CFLm3bSBpXF8zedC/ng5Q0fiitcGJ/2oS1
4  Ee8NWvqInWpmVjs0mAfk6L7Y1H5sAjAqALRrQw9E01XgCPkOCxrC72880vmvqn4MDGxSWS
5  8KGf7s6xz/7ZZqd4/yzbdTGScTKLWm2sOMPIkJ1j1MdkTSLniBagAwBKxmhi3CxB4yRg
6  hZCJFqIoLRfs0vIrZdsZ82YcfUmQyX2CJ/S+bcIjS90XIa01q4nXsrzWCYD9uh7PYk560Wq8
7  TXrFVZ0PuI14GykB8wSSG6dE74bB9Dr/zxrmh05YadrS96v9q+y/aoEW74DWWXuRgCEi
8  CqVaSrKv5i29hAqIvFtA+/99Jo19jp9k0X4T/cWPWS+YBQfz6JXImCxLHH6k4wPWKNX1Y
9  xktR6FXhrdSoA8n9YppkVx332rq7BkK5/XKS4AeJqPkJIbXGS8i8c/VyVVqKNRMASZf1Vu1r
10 ZwXi1eDb9bLvwNcw/Nv4fFTDghux85j63rKyGUUfIp10ZUxrL+RbviZWxY4x+Ft6jlyGn/9q
11 u6oBy3eKPYcFhnkUBuhxY506R/3DoIaw==your_email@example.com
```

Copy these key onto your clipboard, and then SSH into your DreamHost server:

```
1  $ ssh wjgilmore@example.com
```

Next, create a directory named `.ssh` inside your DreamHost account's home directory:

```
1  $ mkdir .ssh
```

Next, enter the `.ssh` directory, and create a file named `authorized_keys`. You can use terminal editors such as `vim` or `nano` to create this file. Alternatively you can create the file locally, and then use SFTP to upload this file to the `.ssh` directory. In any case, you'll then want to paste your *public key* (not your private key) into the `authorized_keys` directory. Once complete, exit the DreamHost server, and then try SSHing in anew:

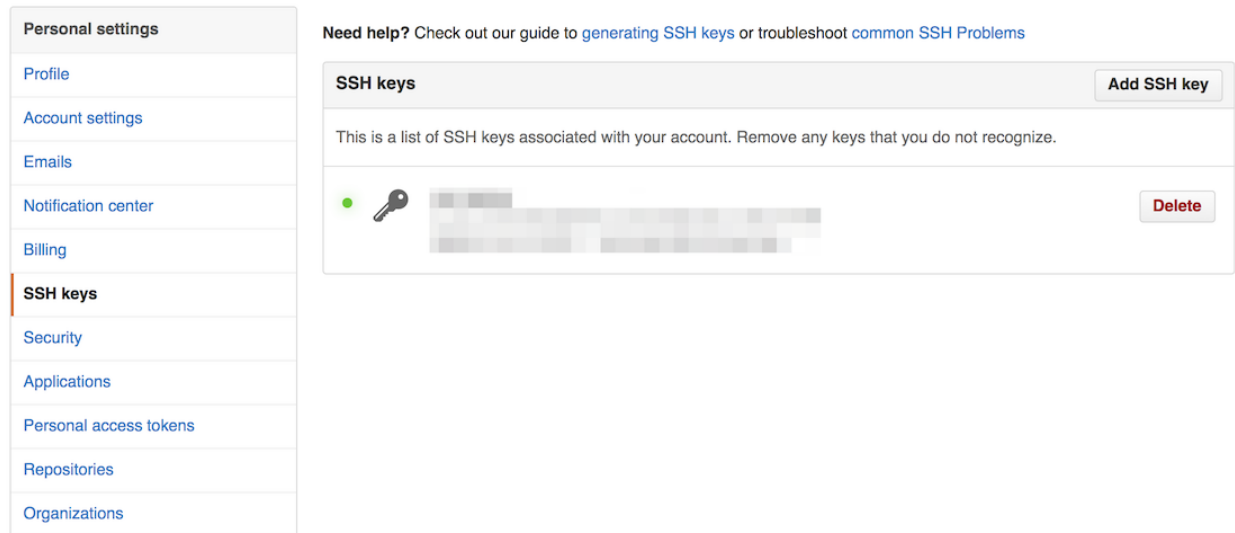
```
1  $ ssh wjgilmore@example.com
```

This time you should immediately enter the server without having to type your password!

¹⁷²<http://bit.ly/1LhUhPj>

Adding Your Public Key to GitHub

In order for the DreamHost server to conveniently talk to GitHub, you'll need to add this newly generated public key to your GitHub account. To do so, sign into your GitHub account, and navigate to Settings > SSH Keys and press the Add SSH Key button. Paste the `id_rsa.pub` contents into the text area, assign the key some easily recognizable name (such as `my_laptop`) and save the changes.



Associating a Public Key with Your GitHub Account

Installing and Configuring Capistrano

With your domain configured and the registrar's DNS updated, it's time to complete the few remaining deployment-specific tasks. Unless you plan on using a different user in conjunction with each domain hosted on DreamHost, the instructions found here are the only ones you'll have to repeat on a per-domain basis when deploying with Capistrano (other than installing Capistrano on your local machine, which you'll only have to do once)!

Let's begin by installing Capistrano. Capistrano is distributed as a Ruby gem, meaning you'll need to install Ruby on your local machine. Keep in mind you will *not* have to learn the Ruby language or anything like that to begin using Capistrano! You just need to install the Ruby interpreter, since Capistrano happens to have been written in Ruby. If you're running OSX or Linux, then chances are Ruby is already installed. You can confirm this by opening a terminal prompt and executing the following:

```
1 $ ruby -v
```

Presuming a version number is installed, then you're ready to go unless you're running a version older than 1.9.3 which is the minimum Capistrano requirement, in which case you'll need to use

your package manager to upgrade. If you're running Windows then you can use the officially recommended [Ruby Installer](#)¹⁷³.

Once Ruby has been installed, install Capistrano by executing the following command from a terminal:

```
1 $ gem install capistrano
```

Once Capistrano has been installed, enter the root directory of the project you'd like to deploy to DreamHost, and execute the following command:

```
1 $ cap install
```

This command will create several directories and files inside your project root directory, including:

- `Capfile`: This file is used by Capistrano to load the core library files and other optional extensions. You won't need to do anything with this file to deploy a Laravel project as described in this chapter, other than leaving the file in place.
- `config`: This directory will be created for reason of housing a directory named `deploy` and a file named `deploy.rb`. The `deploy` directory will in turn house two files named `production.rb` and `staging.rb`. These files contain *target-specific* deployment instructions. For instance, you could define different sets of deployment instructions for different servers, which would be useful if you wanted to manage a beta and production server. For the purposes of this chapter I'll be discussing solely the `production.rb` file. The `deploy.rb` file contains a general set of deployment settings which will be inherited by `production.rb` and `staging.rb` unless one of these latter files expressly overrides those defaults. We'll return to `deploy.rb` and `production.rb` in just a moment, so don't worry about them too much more now. Of course, Laravel applications already have a `config` directory, so Capistrano won't try to recreate it or anything like that. Instead it will just create the aforementioned `deploy` directory (along with the two files) and `deploy.rb` file inside it
- `lib`: This directory houses any custom Capistrano behavior you create for reason of managing your deployments. You won't need to do anything with this directory, so just leave it in place and don't worry about it for now.

With the necessary directories and files in place, we'll need to modify the `deploy.rb` and `production.rb` files. Let's begin with the `deploy.rb` file. Open it up, marvel at the many options available to you, and then delete everything in the file, replacing it with the following. Please carefully read the comments found throughout this file (prefixed with `#`) as it is crucial for you to understand the various settings found herein:

¹⁷³<http://rubyinstaller.org/>

```
1  # This should match whatever Capistrano gem version
2  # Execute cap -V to find this out.
3  lock '3.2.1'
4
5  # Where does your project GitHub repository reside?
6  set :repo_url, 'git@github.com:wjgilmore/example.git'
7
8  # This is where the files will reside. This should
9  # match the directory you defined when configuring
10 # your domain directory, but WITHOUT the current/public
11 # path
12 set :deploy_to, '/home/YOUR_USERNAME/example.com'
13
14 # Use this directory for housing the cloned repository
15 # NOTE: you will need to create this directory inside
16 # your DreamHost account home directory.
17 set :tmp_dir, '/home/YOUR_USERNAME/tmp'
18
19 # We're deploying a Git repository
20 set :scm, :git
21
22 # Set the logging to debug so we can see what is going on
23 set :log_level, :debug
24
25 # Maintain three releases on the server
26 set :keep_releases, 3
27
28 namespace :deploy do
29
30   # After deployment is complete, execute the
31   # following commands. Notably, we are going to
32   # run composer, make the artisan script executable,
33   # create a symbolic link to a production-specific
34   # .env file, and then run any outstanding migrations.
35   desc "Build"
36   after :updated, :build do
37     on roles(:app) do
38       within release_path do
39         execute :composer, "install --no-dev --quiet"
40         execute :chmod, "u+x artisan"
41         execute "ln -s #{shared_path}/.env #{release_path}/.env"
42         execute :php, "artisan migrate --force" # run migrations
```



```
43         end
44     end
45 end
46
47 end
```

The comments found in the above code should be suffice to help you understand what's happening, however take special note of the post-deployment step in which we're creating a symbolic link to a production-specific `.env` file. You're going to want to host an `.env` file that is almost certainly from that found locally, since logically you're going to want your application to connection to a production-specific database, perhaps use different mail server connection parameters, and so forth. Therefore you'll want to upload a production-specific `.env` file to DreamHost. It's common convention to place these "shared" files inside a directory named `shared`, and so go ahead and login to your DreamHost server now and create that directory:

```
1 $ ssh wjgilmore@example.com
```

After logging in, enter the domain directory that was automatically created when you configured the domain:

```
1 $ cd example.com
```

This directory will contain a directory named `current`, and inside it a directory named `public`. This is because you identified the document root as such when configuring the domain. However, we're going to let Capistrano manage this path (exactly why we want to do this will become clear in a moment), so delete this `current` directory (and everything inside it):

```
1 $ rm -rf current
```

Next, remaining inside the domain directory, create a directory named `shared`:

```
1 $ mkdir shared
```

Copy the contents of your project's `.env` file, enter this newly created `shared` directory, and paste the contents into a file also named `.env`. Update the configuration parameters to reflect those used for your production server (such as the DreamHost MySQL hostname, username, database name, and password). Once this file is in place, Capistrano will create a symbolic link from this file to the newly deployed project root directory by executing the following line within the post-deployment block found in `deploy.rb`:

```
1 execute "ln -nfs #{shared_path}/.env #{release_path}/.env"
```

Incidentally, there are somewhat more automated ways to handle shared files in Capistrano, however this first time around I thought it would be useful for you to manually create the shared directory and create the `.env` file, because otherwise the deployment process tends to work like magic, and magic can be dangerous when you're trying to figure out what went wrong at some point down the road.

Next, open `config/deploy/production.rb` and replace the contents with the following:

```
1 role :app, %w{wjgilmore@example.com}
2
3 set :ssh_options, {
4     forward_agent: true,
5     auth_methods: %w(publickey),
6     user: 'wjgilmore'
7 }
```

All we are doing here is telling Capistrano that when deploying to the production server, we're going to SSH in using the username `wjgilmore` and the domain address `example.com`, and as defined by the SSH options we'll be authenticating using the user `wjgilmore`'s public key. The `forward_agent` option is a convenient solution for allowing your DreamHost account user to furnish your *local* SSH keys to GitHub when it's time to access the hosted repository. In order for this to work, keep in mind you will need to add your public key to your GitHub account as described in the earlier section, "Configuring DreamHost for Passwordless Login".

Deploying Your Application

Whew, that was a pretty long process but we are done. All that remains for you to do is deploy! After having committed your changes and pushed them to GitHub (or Bitbucket or anywhere else you please), you can deploy your project to DreamHost by executing the following command:

```
1 $ capistrano production deploy
```

This tells Capistrano to use the `production.rb` file settings to carry the deployment process. Combined with the information found in `deploy.rb`, Capistrano will have everything it needs to complete the process! When executing this command, you'll be treated to a rather exhaustive bit of output pertaining to everything Capistrano is doing to complete the deployment. This is because we set `log_level` to debug in `deploy.rb`. You can tone down the verbosity if desired; see the Capistrano documentation for more information.

Once deployed, return to your DreamHost server and enter the domain directory. You'll now see the following directory contents:

- **current:** This is a *symbolic link* pointing to the *most recently deployed* version of your code. Read that twice. For instance this link will point to a directory such as `/home/wjgilmore/example.com/releases/20150810233629`. Notice how the link points to a directory found in `releases`. Capistrano gives you the ability to actually rollback your deployment to a previous version, doing so by simply removing the symbolic link to the latest deployment and then linking to the second-newest deployment (as determined by the timestamp)! I'll show you how to rollback in just a moment.
- **releases:** This directory contains a predetermined number of current and previous deployments, the number of which being determined by the `keep_releases` setting in your `deploy.rb` file. For instance, if `keep_releases` is set to 3, then a total of three deployments will be maintained (the current deployment and the previous two).
- **repo:** This contains the cloned repository.
- **shared:** This is the directory we created prior to deploying; it contains the production `.env` file.

Each time you deploy a new version of your project, Capistrano will retrieve the latest changes, archive them to a timestamped directory found in `releases`, remove the symbolic link pointing to the now “old” deployment, and then create a new symbolic link pointing to the new deployment. It's an incredibly easy yet powerful solution! Among other things this gives you the ability to roll back your changes using the `deploy:rollback` command:

```
1 $ cap production deploy:rollback
```

After deploying for the first time, make a change to your project (add some text to the home page or similar), deploy anew, confirm the changes are on the production site, and then run `deploy:rollback`. Once the command completes, your production site will have reverted to the previous version! Then, undo and commit the local changes, deploy once more, and the production site will no longer include the changed text!

Summary

As you can see, if you require a custom deployment solution using a tool such as Capistrano, it can be a bit of a chore to configure the first time around. Once in place though, it is a solution you'll return to repeatedly for all of your new projects. I've been using Capistrano for several years now and honestly don't know what I'd do without it. And to think we've hardly scratched the surface in terms of what it can do! Be sure to check out the Capistrano documentation for more information.

Appendix B. Feature Implementation Cheat Sheets

The book provides occasionally exhaustive explanations pertaining to the implementation of key Laravel features such as controllers, migrations, models and views. However, once you understand the fundamentals it isn't really practical to repeatedly reread parts of the book just to for instance recall how to create a model with a corresponding migration or seed the database. So I thought it might be useful to provide an appendix which offered a succinct overview of the steps necessary to carry out key tasks. In this appendix you'll find the following cheat sheets:

- Creating a Model
- Creating a Migration
- Seeding the Database
- Creating a Resource Controller
- Creating and Validating a Form

Of course, in future releases I'll continue adding new cheat sheets. If you have a suggestion for a cheat sheet e-mail me at wj@wjgilmore.com!

Creating a Model

You can create a model using the Artisan `make:model` command:

```
1 $ php artisan make:model TodoList
2 Model created successfully.
```

You'll find the newly generated `TodoList` model inside `app/TodoList.php`. Keep in mind however that as of Laravel 5.1 the `make:model` command will not generate a corresponding migration by default. To generate the corresponding migration you'll need to supply the `-m` option:

```
1 $ php artisan make:model Todolist -m
2 Model created successfully.
3 Created Migration: 2016_10_11_031538_create_todolists_table
```

After generating the model and corresponding migration (which you'll find in `database/migrations`) you will need to update the migration file to reflect the desired columns, and the execute the migration:

```
1 $ php artisan migrate
```

Creating a Migration

You'll occasionally wish to create a migration without a corresponding model. This is useful for instance when you need to create a pivot table when implementing a many-to-many relationship. For instance to create a pivot table used to manage a many-to-many relationship between a blog post and category model, you might create a pivot table called `category_post`:

```
1 $ php artisan make:migration create_category_todo_list_table \  
2 --create=category_post  
3 Created Migration: 2016_10_11_032021_create_category_...
```

Next you'll want to open the newly created migration (found inside `database/migrations`) and define the appropriate columns inside the `up` method. Once complete, save your changes and execute the migration:

```
1 $ php artisan migrate
```

Should you make a mistake in the migration and want to correct something (e.g. incorrect datatype, missing option), you can undo the migration by *rolling it back*:

```
1 $ php artisan migrate:rollback
```

It is often useful to determine your migration status, because it can be easy to forget which migrations you've already executed:

```
1 $ php artisan migrate:status
```

Seeding the Database

Most applications require a starter set of data in order to function properly (such as a list of categories or a product catalog). While you could deploy the application and then painstakingly use an administration form to enter this data, a much more efficient approach involves *seeding the database*.

To seed the database you'll create a seed file. You can use Artisan's `make:seeder` command to create the file skeleton for you:

```
1 $ php artisan make:seeder CategorySeeder
```

This will create a file named `CategorySeeder.php` and place it in the `database/seeds` directory. From there you can use Eloquent syntax to create the records. For instance, here is a seed file used by the TODOparrot application to seed the categories table:

```
1 <?php namespace App;
2
3 use Illuminate\Database\Seeder;
4 use Illuminate\Database\Eloquent\Model;
5
6 class CategoryTableSeeder extends Seeder {
7
8     public function run()
9     {
10
11         Category::create([
12             'name' => 'Leisure'
13         ]);
14
15         Category::create([
16             'name' => 'Work'
17         ]);
18
19         Category::create([
20             'name' => 'Shopping'
21         ]);
22
23     }
24
25 }
```

After creating your seed file, open `database/seeds/DatabaseSeeder.php` and add the following line to the `run` method:

```
1 $this->call('App\CategoryTableSeeder');
```

Notice how I've prefixed the seed file class name with the namespace. This is a common point of confusion when using Laravel's seed feature. Also, and this is very important, I've had to run the following command to get Laravel to recognize the new seed file class:

```
1 $ composer dump-autoload
```

During development your seed files will be in a constant state of flux, so you'll often want to rebuild the tables from scratch. You can just delete the table contents prior to reseeding the table as demonstrated here:

```
1 public function run()  
2 {  
3     Model::unguard();  
4  
5     DB::table('categories')->delete();  
6  
7     $this->call('App\CategoryTableSeeder');  
8  
9     Model::reguard();  
10  
11 }
```

Creating a Resource Controller

Laravel controllers are just plain old PHP classes, however it can be nice to just sit back and let Artisan do the tedious work of creating the class skeleton for you. The utility of doing so is particularly apparent when creating a *resourceful controller*, because Artisan will create all seven resource methods for you (index, show, create, store, edit, update, destroy). To create a resourceful controller execute the following command:

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

A new controller file named `ListsController.php` will be placed in `app/Http/Controllers`. Keep in mind you'll need to add the route declaration to `app/Http/routes.php`:

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

You'll also need to create the necessary action views in `resources/views`.

Creating and Validating a Form

Advance warning: this cheatsheet is far longer than any others found in this appendix, however the reality is that creating and validating a Laravel form involves numerous moving parts, all of which must be properly implemented in order for a form to be created, validated, and processed. Even so, this succinct guide should do wonders in terms of helping you quickly understand (or remember) how these parts are implemented and integrated.

In the bad old days of web development it was common practice to hand-code an HTML form, and then write some custom code to ensure the user input was valid. At some point PHP got fancy and added some standard [validation and input functions](#)¹⁷⁴ however it was still up to you to define a *convention* for validating the input and subsequently processing the valid data, or notify the user of the invalid data and present the form anew. Laravel removes all of this hassle by providing you with both the code and convention required to implement all of these tasks.

Let's work through an example involving adding a category to the TODOParrot application's `categories` table. Begin by creating the `Category` model and corresponding migration.

Creating the Model and Migration

To keep things simple let's presume the `categories` table consists of a single field for managing the category name (in addition to the usual ID primary key and create/update timestamps). Let's generate the `Category` model:

```
1 $ php artisan make:model Category -m
2 Model created successfully.
3 Created Migration: 2016_10_17_152001_create_categories_table
```

Next, open the migration file (inside `database/migrations`) and update the `up` method to look like this:

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

Next, run the migration:

¹⁷⁴<http://php.net/manual/en/book.filter.php>


```
1 $ php artisan migrate
2 Migrated: 2016_10_17_152001_create_categories_table
```

With the model and corresponding table in place, let's next create a controller which will serve and subsequently process the form.

Creating the Resourceful Controller

Use Artisan's `make:controller` command to create a controller which (among other things) we'll use to present and process the form:

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

The `make:controller` command will by default create a *resourceful controller* containing the seven customary RESTful methods (`index`, `show`, `create`, `store`, `edit`, `update`, `destroy`). However, Laravel does not yet know how to respond to these methods, because their associated routes need to be defined inside `app/Http/routes.php`. Open this file and add the following line:

```
1 Route::resource('categories', 'CategoriesController');
```

Once defined, you'll be able to navigate to for instance `/categories` to access the `Categories` controller's `index` action, and `categories/create` to access the `Categories` controller's `create` action, because Laravel will map each route to an appropriate method. At this point both routes will present blank pages because we haven't yet created the views. Let's do so next.

Creating the Form View

For the purposes of this tutorial we're only interested in `create` and `store`, which is responsible for presenting the form to the user, and `store`, which saves the form data (presuming it passes validation; more on this in a moment). Currently these actions are empty:

```
1 ...
2
3 public function create()
4 {
5     //
6 }
7
8 public function store(Request $request)
9 {
10    //
11 }
12
13 ...
```

Update the create action to look like this:

```
1 public function create()
2 {
3     return view('categories.create');
4 }
```

This tells Laravel to serve a view named `create.blade.php` residing in the directory `resources/views/categories`. Because neither the file nor the directory have been created yet, do so now, beginning with the directory:

```
1 $ cd resources/views
2 $ mkdir categories
```

Next, create a file named `create.blade.php`, saving it to `resources/views/categories`. For the moment just add the following HTML to the file:

```
1 <h1>Create a Category</h1>
```

Save the changes and navigate to `/categories/create` and you should see the above H1 header. While a nice start, we of course want the user to be presented with a form. While you could tediously create this form yourself, that's really no fun so let's instead use the fantastic laravelcollective/html¹⁷⁵ package to do the hard work for us. Add the package to your project by executing the following command from inside your project root directory:

¹⁷⁵<http://laravelcollective.com/docs/5.1/html>

```
1 $ composer require laravelcollective/html
```

Next, open `config/app.php` and add the following line to the bottom of the providers array:

```
1 'providers' => [
2     ...
3     Collective\Html\HtmlServiceProvider::class,
4 ]
```

Continue scrolling down and add the following line to the bottom of the aliases array:

```
1 'aliases' => [
2     ...
3     'Form' => Collective\Html\FormFacade::class
4 ]
```

Save the changes and the package is now installed and configured! Return to `create.blade.php` and add the following markup:

```
1 <h1>Create a Category</h1>
2
3 {!! Form::open(
4     [
5         'route' => 'categories.store',
6         'class' => 'form'
7     ]
8 ) !!}
9
10 @if (count($errors) > 0)
11 <div class="alert alert-danger">
12     There were some problems adding the category.<br />
13     <ul>
14         @foreach ($errors->all() as $error)
15             <li>{{ $error }}</li>
16         @endforeach
17     </ul>
18 </div>
19 @endif
20
21 <div class="form-group">
22     {!! Form::label('Category') !!}
```

```

23     {!! Form::text('name', null,
24     [
25         'class'=>'form-control',
26         'placeholder'=>'List Name'
27     ]) !!}
28 </div>
29
30 <div class="form-group">
31     {!! Form::submit('Create Category!',
32     ['class'=>'btn btn-primary']
33     ) !!}
34 </div>
35 {!! Form::close() !!}
36 </div>

```

The `laravelcollective/html` package's syntax is so intuitive that I'd imagine you understand everything found in this markup, however I'd like to nonetheless highlight a few key items:

- The form route points to `categories/store`, because the `Categories` controller's `store` method will process the form.
- `Form::open` uses the `POST` method by default, so there's no need to reference the method explicitly.
- The `@if...@endif` block is used to present information about validation errors to the user. We'll return to this matter in a moment.
- For demonstration purposes I'm passing along [Bootstrap](http://getbootstrap.com/)¹⁷⁶ attributes to the form elements. Obviously the form will still work without these stylistic attributes.

With the form in place, return to `categories/create` and you should see the form. If you input a category and submit the form, you'll be taken `categories/store` however of course nothing will actually happen because we haven't written that code yet. Let's do so next.

Processing the Form

Return to `CategoriesController.php` and scroll down to the `store` method. This is where the user is taken after submitting the form. Update the `store` method to look like this:

¹⁷⁶<http://getbootstrap.com/>

```
1 use App\Category;
2
3 ...
4
5 public function store(Request $request)
6 {
7
8     $category = new Category;
9
10    $category->name = $request->get('name');
11
12    $category->save();
13
14    return \Redirect::route('categories.show',
15        [$category->id])
16        ->with('message', 'Your category has been created!');
17
18 }
```

It really doesn't get much simpler than this. We instantiate a new `Category` model object, assign the submitted category name, save the record, and then redirect the user to `categories/show` (which I'll leave to you as an exercise to implement). If you resubmit your form with a category name, sure enough you'll see that it has been added to the database. But wait a second, what about validation? Sure enough, you can currently submit a *blank* category, and an empty value will be saved to the database! This is of course not acceptable, so let's use a *form request* to validate the data.

Validating the Form

Laravel 5 offers a great new feature known as a *form request* which allows you to easily validate your form input without polluting the controller. The validation logic is managed in a separate file, and you can generate this file using Artisan's `make:request` feature:

```
1 $ php artisan make:request CreateCategoryFormRequest
2 Request created successfully.
```

This creates a file named `CreateCategoryFormRequest.php`, placing it inside `app/Http/Requests`. The file currently looks like this:

```
1 <?php
2
3 namespace App\Http\Requests;
4
5 use Illuminate\Foundation\Http\FormRequest;
6
7 class CreateCategoryFormRequest extends FormRequest
8 {
9
10     public function authorize()
11     {
12         return false;
13     }
14
15     public function rules()
16     {
17         return [
18             //
19         ];
20     }
21 }
```

Modify the `authorize` method to look like this:

```
1 public function authorize()
2 {
3     return true;
4 }
```

The `authorize` method can be used to ensure the user is authorized to submit this form. For instance you might only want to make the form available to logged-in users. Frankly I don't particularly understand why this is useful since one could instead use middleware to accomplish the same task, but to each his own I guess. Anyway, you'll want to set this to `true` because when `false` is returned the form will no longer be processed.

Next, modify the `rules` method to look like this:

```
1 public function rules()  
2 {  
3     return [  
4         'category' => 'required'  
5     ];  
6 }
```

This uses Laravel's [validation](#)¹⁷⁷ syntax to declare the form's category field as required. Note the array key matches the name of the corresponding form field. Of course, you're free to add additional form fields and corresponding validators. You can also combine validators like so:

```
1 'category' => 'required|alpha'
```

With the form request complete, you need to integrate the request into the `Categories` controller's store action. Add the following line to the top of `CategoriesController.php`:

```
1 use App\Http\Requests\CreateCategoryFormRequest;
```

Next, update the store action to accept a `$request` object of type `CreateCategoryFormRequest`:

```
1 public function store(CreateCategoryFormRequest $request)  
2 {  
3  
4     ...  
5  
6 }
```

With the store action updated to use the custom request, Laravel will *automatically* validate the submitted form data, and if invalid, will *automatically* return the user to the form with one or several validation-related error messages. That's right; there is nothing else you need to do to the existing store action code!

With the model, table, controller, view, and form request in place, categories can be safely added to the database!

¹⁷⁷<http://laravel.com/docs/master/validation>