CODE SMART

THE LARAVEL FRAMEWORK *VERSION 5* FOR BEGINNERS

DAYLE REES

Laravel: Code Smart

The Laravel Framework Version 5 for Beginners

Dayle Rees

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

This version was published on 2016-05-22

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.

Tweet This Book!

Please help Dayle Rees by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm learning @laravelphp with #codesmart by @daylerees. Get it at https://leanpub.com/codesmart

The suggested hashtag for this book is #codesmart.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#codesmart

Also By Dayle Rees

Laravel: Code Happy

Laravel: Code Happy (ES)

Laravel: Code Happy (JP)

Laravel: Code Bright

Code Happy (ITA)

Laravel: Code Bright (ES)

Laravel: Code Bright (SR)

Laravel: Code Bright (JP)

Laravel: Code Bright (IT)

Laravel: Code Bright (TR) Türkçe

Laravel: Code Bright (PT-BR)

PHP Pandas (PHP7!)

Laravel: Code Bright (RU)

PHP Pandas (ES)

PHP Pandas (IT)

PHP Pandas (FR)

PHP Pandas (TR)

PHP: Composer

Contents

Acknowledgements

First of all, I would like to thank my girlfriend Emma, for not only putting up with all my nerdy ventures but also for taking the amazing red panda shots for the books! Love you, Emma!

Thanks to my parents, who have been supporting my nerdy efforts for close to thirty-two years! Also, thanks for buying a billion or so copies of the first few books for family members!

Taylor Otwell, the journey with Laravel has been incredible. Thank you for giving me the opportunity to be part of the team and for your continued friendship. Thanks for making a framework that's a real pleasure to use, makes our code read like poetry, and for putting so much time and passion into its development.

Thank you to everyone who bought my previous books and to all of the Laravel community. Without your support, futures titles would not have been possible.

Errata

While this may be my fourth book, and my writing is steadily improving, I assure you that there will be many, many errors. I don't have a publisher, a review team, or an English degree. I do the best that I can to help others learn about Laravel. I ask that you, please be patient with my mistakes. You can support the title by sending an email with any errors you might have found to me@daylerees.com[1] along with the section title.

Errors will be fixed as they are discovered. Fixes will be released in future editions of the book.

[1] mailto:me@daylerees.com

Feedback

Likewise, you can send any feedback you may have about the content of the book or otherwise by sending an email to me@daylerees.com[2]. You can also send a tweet to @daylerees[3]. I will endeavour to reply to all mail that I receive.

[2] mailto:me@daylerees.com
[3] https://twitter.com/daylerees

Translations

If you would like to translate Code Smart into your language, please send an email to me@daylerees.com[4] with your intentions. I will offer a 50/50 split of the profits from the translated copy, which will be priced at the same as the English copy.

Please note that the book is written in markdown format, and is versioned on Github[5].

[4] mailto:me@daylerees.com
[5] http://github.com

How to read this book

Start with the letters, then the words, then the sentences. Just kidding!

Beginners

If you're new to Laravel, you'll want to read through the book as you would any other. Each chapter will lead into the next, and they have been ordered in a fashion that will make the learning process as simple as possible.

You don't need any prior Laravel knowledge, but you should have an understanding of the PHP language. If you've never used PHP, I'd suggest reading my other book 'PHP Pandas' first. It's available on Leanpub.com[6].

Experienced

Experienced developers will find Code Smart useful as a reference book. The chapters have been named in a format to make 'skimming' the book for content much more simple than in previous versions. Simply find the topic in the index, navigate to the chapter, and skim the examples until you find what you need.

As always, you'll find much more advanced topics at the end of the book. New topics will be added as we move forward, see the next chapter.

Updates

My books are hosted on Leanpub, and I do this for good reason. It's not that I prefer to self-publish, but instead, the flexibility that Leanpub give. Leanpub allows for books to be published while in progress. What this means is that you can release either partial content, or publish additional chapters at a later date.

Laravel, as a framework, will evolve. I believe that the book should evolve with it. For that reason, it's important that you download the latest copy of the book at a regular interval. If you want to stay informed as new topics and changes are released, then you'll find that I post most of my update on my Twitter profile[7]. You'll also receive an email whenever a new version of the book is published.

[6] https://leanpub.com/php-pandas
[7] https://twitter.com/daylerees

New topics will be added to the book at regular intervals. At launch, the book will contain all of content that was present in Code Bright, refreshed and updated for the new framework version, plus a bunch of additional new chapters. I'm committing to delivering *at least* one chapter per week, expanding the book until it's the 'Encyclopedia Laravelica'.

If you'd like to suggest a chapter to cover next, think a topic is missing, or would like something explained in further detail, then please contact me at me@daylerees.com[8].

Let's build this masterpiece together!

[8] mailto:me@daylerees.com

Changes

Developers love to keep change logs, don't they? They contain everything that they've changed in their code, date by date. Since this is a book that is in constant evolution, I have decided to do the same with Code Smart. In this chapter, you'll find all of the changes made to the book, including new content that has been released, and where to find it.

27th April 2016

Code Smart was released. The book includes updated (and refreshed) content from Code Bright, and some chapters that are unique to Laravel 5. The preparatory process has been redesigned from the ground up to make the learning process more simple. Enjoy!

21st May 2016

Added a bunch of fixes, and new chapters on Valet, Middleware and Service Providers. Have fun!

22nd May 2016

Added a big chunky chapter on the Container. It's worth a read!

1. Introduction

Why hello there! Pleased to meet you.

My name is Dayle, and I'm a developer. I've just started my fourth decade on this planet, and I hail from the pleasant land of Wales. Many years ago, I was one of the first to use a brand new framework for the PHP programming language called 'Laravel.' From the first read of the documentation, I knew that I had to start building applications with it. It was clean, concise and beautiful. I saw great potential in the framework, and I think I was right!

Since then, I've been a core contributor, a conference speaker, a consultant, and an author of several books based on the Laravel framework. The books have received an astounding reaction, and have thousands of fans worldwide! It's been an incredible experience. If you've read one of my books before, then thank you so much! If not, well, let me take a moment to explain how this works.

You see, I'm not a traditional author. I don't like making things complicated. I don't like using posh words to make myself sound smart, and I write my books as if we're sharing an adventure of discovery. I've said it before, and I'll say it again; I like to write my books as if we're two friends, sitting in a bar, sharing a drink and a conversation. Just so long as long as you're getting a round in!

This title is the third in my Laravel series, and it covers version 5.x of the Laravel framework. I've learned a little more about writing technical books with every one that I've released, and so, I hope you find reading this book an enjoyable experience. If there's anything you don't like about the book or if there's a chapter or concept that's not as clear as it could be, then please send me an email! I'm very responsive. In fact, I pride myself on it! For this book to be as perfect as it can be, I'm going to need your feedback. Let's write this masterpiece together!

Right then, you're probably excited to get started aren't you? It's a big framework, but I promise that if we take it step by step, you'll be building great apps in no time at all! Are you prepared? Go ahead, flip the page.

2. Installation

I'm sure you're eager to get started with Laravel. Since it's a web framework, it's important that we create an environment with a web-server. In a production environment, this would take a long time to configure, and would rely on a unix-based platform, so let's build a homestead instead.

> This is no time for home-making!

Hah! You might be right, but that's not quite what we mean in this instance. A 'Homestead' is a virtualised web application environment powered by a piece of software called 'Vagrant.' It can get us up and running on Laravel in a jiffy!

Homestead is my preferred option of running Laravel. It ships with all the software required to launch the framework, however, it is one extra piece of software to learn. You'll need to remember to SSH into the virtual machine and run your commands from there. For this reason, the next chapter will introduce 'Valet'. Valet is a binary that will allow you to run web applications using a version of PHP installed on your computer. Right now, it's only available on Mac. If you're not using a Mac, or you'd like to take my advice, then please continue reading. Otherwise, feel free to skip to the next chapter to use Valet instead.

Install Software Dependencies

Before we can get our homestead running, we're going to need to install a few dependencies. Here's the full list.

- PHP http://www.php.net/[1]
- Git https://git-scm.com/[2]
- Composer https://getcomposer.org/[3]
- Virtualbox https://www.virtualbox.org/[4]
- Vagrant https://www.vagrantup.com/[5]

[1] http://www.php.net/
[2] https://git-scm.com/
[3] https://getcomposer.org/
[4] https://www.virtualbox.org/
[5] https://www.vagrantup.com/

I'm not going to provide specific installation instructions because they change from week to week. Instead, head over to the URLs above and you'll find sections of each site that will cover download and installation instructions for your operating system.

Let's take a look at what each piece of software offers.

We're going to need '**PHP**'. It's the language used by the Laravel framework. There's no way of getting out of this one! Download the latest version that's available.

We'll use '**Git**' for version control. It's also a great way of downloading a base copy of Laravel. It's worth having since it's a day-to-day part of any experienced software developer's workflow.

We'll use '**Composer**' to manage all the libraries that we use with PHP. Go ahead and grab it. It's one of my favorite pieces of software!

Next, we have '**Virtualbox**'. It's software that will allow operating system virtualisation. Essentially, this will allow you to run 'pretend' computers on your main computer. Since web development setups run best on unix environments, we'll be using a virtual unix environment so that we can develop on any platform!

Finally, we have '**Vagrant**'. Vagrant is used to provision virtual environments, and to configure them the way that we want them. It's a command line program, so we'll be using it in the terminal. Vagrant will be making use of Virtualbox to create project environments.

Once we've got all five pieces of software installed, we're ready to move to the next section.

Create a Laravel Project

Before we build our homestead, we'll need to have a Laravel project ready for it.

First, we'll need to decide on a disk location for your projects. I'm going to be creating a 'Projects' directory in my home folder. This is where all my Laravel projects will live.

Let's start by navigating to our projects folder, shall we?

Example 01: Navigate to projects directory.

```
1 cd ~/Projects
```

That wasn't so hard, was it? Next, we can use the 'Composer' software that we installed earlier to create a new Laravel project. Let's type the following command and see what happens. You can replace the word 'example' with your project name if you'd prefer something different.

Example 02: Create a Laravel project.

```
composer create-project laravel/laravel example
```

You'll see a lot of output. This is where Composer is downloading all of the libraries that support Laravel. We call these 'package dependencies'. Once complete, you've installed Laravel. Continue to the next section to add a homestead.

Install Homestead

First, let's navigate to the directory that our Laravel project is contained within.

Example 03: Navigate to the project directory.

```
cd ~/Projects/example
```

Next, we're going to install the homestead composer package, so that we can create a homestead virtual environment. Don't worry; it's simple. Just type the following command.

Example 04: Add homestead to composer dependencies.

```
composer require laravel/homestead --dev
```

You'll see a few more composer packages installed in the output of the command. Don't worry. That's what we want!

Next, we'll run the 'make' command, to configure our project for homestead. Here's the full command.

Example 05: Install homestead.

```
php vendor/bin/homestead make
```

Well, at least that one was super quick!

Next, it's time for the long one. Let's use the following command to boot up a virtual machine. If it's the first time that you're running this command, it's going to take a while to download the virtual machine image. It's around 600mb or so. Go and make yourself a coffee!

Example 06: Boot the virtual machine.

```
vagrant up
```

Once the virtual machine has come online, there's one final thing you're going to need to do. We'll add the homestead app domain to our hosts file so that we can conveniently access our web application in the browser.

Use your favorite editor to edit the file at `/etc/hosts`. You're going to need admin/root privileges to access this file. You'll want to add the following line.

Example 07: Add a local DNS entry.

```
192.168.10.10  homestead.app
```

To check whether everything is working, let's visit `http://homestead.app` in our browser. You should see the text 'Laravel 5'. That's our Laravel application!

Mastering Vagrant

Your brand new web development environment exists on a virtual machine, and that machine is draining the resources of your host machine. For example, it will share your available RAM and processor time.

From time to time, it might be useful to be able to 'halt' and restore our environment to save system resources. It's also noting that when we restart our host computer, our virtual environment will not boot automatically.

For this reason, let's go over a few basic commands to maintain our Vagrant environments. We'll be using these commands in the directory containing the `Vagrantfile` file.

The first command is used to start your virtual machine. You've already used it. Here's an example.

Example 08: Boot the virtual machine.

```
vagrant up
```

The second command is the exact opposite. It's used to halt your virtual machine. It will no longer be using system resources. You're going to want to use the following.

Example 09: Halt the virtual machine.

```
vagrant halt
```

If you decide to change the configuration of your Vagrant box by updating the `Vagrantfile`, you're going to want to use 'provision' to apply the changes to the virtual machine.

Example 10: Update the virtual machine settings.

```
vagrant provision
```

In some circumstances, you may want to run commands on the guest virtual machine. You can easily access this machine by using the Vagrant SSH command. This will allow you sudo access to the guest. For example.

Example 11: SSH into the virtual machine.

```
vagrant ssh
```

Finally, if we're done with our project, and don't want the virtual machine image hogging our disk space, we can ditch it! Just use the 'destroy' command!

Example 12: Destroy the virtual machine.

```
vagrant destroy
```

In the next chapter, we'll take a look at an alternative way of running a Laravel application.

3. Valet

Laravel is a piece of software that exists within the Laravel ecosystem to allow you to run web applications using the stack that is already installed on your development machine. For example, you might already have PHP installed locally, or a copy of MariaDB to use as a database. Homestead would run a VM containing many types of software that can be used with Laravel, but with Valet; you install only what you need.

Personally, I would recommend using Homestead, as it will allow you to version your running environment alongside your code. This is an incredibly useful advantage, especially when working within a team. However, it is understood that Homestead is more complicated than Valet.

If you're not comfortable with Homestead, the continue reading to learn how to be up and running with Valet in no time at all.

Installation

Since Valet uses the software on your computer to run its stack, then you'll first need to install *at least* PHP. Remember, Valet will only work on a Mac, so if you're using a Windows or Linux machine, then I'd just skip to the next chapter at this point.

On a Mac, PHP can be installed using the 'Homebrew' software. It would be best to check the Homebrew website[1] for instructions on how to install the software. Once it's installed, you can install PHP using the following command:

Example 01: Install PHP with Homebrew.

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

Once you've got your copy of PHP installed, we'll continue to install the Valet binary. This can be achieved using Composer. If you'd like to find out how to install Composer, then please check the previous chapter. It would be pointless to repeat this information, wouldn't it?! We are programmers after-all. We hate repeating ourselves!

We'll install Valet globally by running the following command:

[1] http://brew.sh/

Example 02: Install Valet using Composer.

```
composer global require laravel/valet
```

You may need to add Composer's binary path to your system PATH variable. You'll be able to do that for the current session using the following command.

Example 03: Add Composer to system path.

```
export PATH="$PATH:$HOME/.composer/vendor/bin"
```

Next, install the Valet service with the following command.

Example 04: Install the Valet service.

```
$ valet install

WARNING: Improper use of the sudo command could lead to data loss
or the deletion of important system files. Please double-check your
typing when using sudo. Type "man sudo" for more information.

To proceed, enter your password, or type Ctrl-C to abort.

Password:
[dnsmasq] is not installed, installing it now via Brew...

Valet installed successfully!
```

Great! Valet is now installed. Let's start using it, shall we?

Adding Sites

Valet is fully equipped to serve multiple applications, so first we'll need to 'park' our application directory. Let's say that we're storing our projects in `~/projects`. We'll navigate to the projects folder and create a new Laravel project using Composer. We'll call our project 'blog' for example purposes.

Example 05: Create a new Laravel project.

```
$ composer create-project laravel/laravel blog
```

Next, we'll navigate to and park our applications directory.

Example 06: Park our project directory.

```
$ cd ~/projects
$ valet park
This directory has been added to Valet's paths.
```

That's it; you're done! Honestly!

You see, any project within your parked directory will be served under the hostname `projectname.dev`. For example, the blog we created above will be accessible at `blog.dev`. If you visit the URL, you'll see the Laravel welcome page.

Don't forget to install some database software if you'd like to store information later. Do you see why I prefer homestead yet? Well, each to their own I suppose!

Valet Commands

If you'd like to start, stop or restart the valet daemon that is serving your applications, then the following commands are available.

Example 07: Valet service commands.

```
$ valet stop
$ valet start
$ valet restart
```

Sharing

Valet offers a useful shortcut to be able to share your applications with others. Please note that this is a tool for collaboration and that you shouldn't use this to serve your applications for the long-term! Simply use the `share` command as follows:

Example 08: Share a Valet application.

```
$ cd ~/projects/blog
$ valet share
```

On the screen that appears, you'll find a 'Forwarding' URL that can be used to share your applications with others. You'll also see the requests as they pour into your application.

If you'd like to use this sharing feature with Homestead, or any other stack, then take a look at the 'ngrok' website[2].

We've now got everything we need to start working with Laravel. It looks like it's time to move on to something new. In the next chapter, we'll take a look at the lifecycle of a modern PHP application. Do your best superhero pose and flip that page.

[2] https://ngrok.com/

4. Lifecycle

If you've not used a PHP framework before, then you'll likely be used to having a bunch of different PHP files in your web directory. The person using your web application will request each script individually.

Laravel uses a front-controller and router combination. What this means is that there's only a single PHP file in your web root directory, and all requests to your application will be made through this script. This file is called `index.php` and you'll find it in the `public` directory, which is the only directory that should be accessible on the web.

> I can't make a web application with one page!

Don't worry. We've got a solution to that. You see, Laravel uses some different techniques to serve different content based on the state of the web-browser request. In fact, here's a diagram to display the lifecycle of a single request to a Laravel framework application.

```
1  Request > Services > Routing > Logic > Response
```

In a way, a request to a webserver is an input/output loop. In honesty, there are a few more steps involved, but this isn't the place to discuss them! Let's step through each section of the process, shall we?

Request

Every request made by a web browser to your application has a broad range of information attached to it. For example, the URL, the HTTP method, request data, request headers, and information about the source of the request.

It's up to Laravel, and your application to interrogate the information within the request, to decide which action to perform. Using Laravel, the information for the current request is stored in an instance of the class `Illuminate\Http\Request`, which extends from the Symfony Framework `Symfony\Component\HttpFoundation\Request` class.

You see, the Symfony Framework has a fantastic implementation of the HTTP protocol in its 'HTTP Foundation' package. Laravel makes use of this package to avoid re-inventing a wheel.

We now have a request, and so we have all the information we need for our application to decide on an appropriate action. So, what's next? Let's take a look.

Services

The next step in the process is the bootstrapping of the framework. Laravel ships with a bunch of services and features that make our lives as web developers considerably easier! Before we can make use of these services, they need to bootstrapped.

The framework will load all defined services and configuration, and ensure that it has everything it needs to support our code. We'll take a closer look at how services are loaded within the 'Service Providers' chapter.

The framework is now ready for our code, but which piece of code should it run? Let's find out!

Routing

As we have discovered previously, there's only a single script that's accessible when using Laravel. How do we show different pages, or make different actions? That's where the router makes an appearance!

The router's sole purpose is to match up a request to the appropriate piece of code to execute. For example, the router will know that it should run the code to display a users profile page when the request includes a HTTP verb of 'GET' and a URI of '/user/profile'.

Laravel has a nice way of defining these routes, and we'll be looking at that very soon. For now, let's take a look at what happens next.

Logic

Next, we have our logic segment. This section can best be described as *your* code. It's where you'll be talking to a database, validating forms, or showing pages.

In Laravel, we've got some different ways of defining your logic, but we'll be looking at this in a later chapter. For now, let's take a look at the final section, shall we? I know our application users will be eager for this one!

Response

The response is created at the end of your logic. It might be a HTML template. It might be some JSON data; it might just be a string. Hey, it might be nothing at all. In some sad circumstances, it might be an error screen or a 404 page! It's good to have options, isn't it?

The response is what your application users will see. It's the part that they are most excited about!

Let's summarize, shall we?

The web browser sends a request. Laravel bootstraps its services, interrogates the request, and performs routing operations. Our code is executed and delivers a response to the user. It's a wonderful smooth operation, isn't it?

It sounds like a very simple process, but I promise that keeping these "flows" in your mind, will make you a better web developer.

Let's turn up the music, and keep this party going in the next chapter! We're going to be looking at the MVC (Model View Controller) software architecture pattern. It's used by plenty of Laravel apps, so I think it's worth reading!

5. Namespaces

In version 5.3 of PHP, a new feature known as name-spacing was added to the language. Many modern languages already had this feature for some time, but PHP was a little late to the scene. None the less, every new feature has a purpose. Let's find out why PHP namespaces can benefit our application.

You can't have two classes that share the same name. They have to be unique. The issue with this restriction is that if you are using a third party library which has a class named `User`, then you can't create your own class also called `User`. This is a real shame because that's a pretty convenient class name, right?

PHP namespaces allow us to circumvent this issue. In fact, we can have as many `User` classes as we like. Not only that, but we can use namespaces to contain our similar code into neat little packages, or even to show ownership.

Let's take a look at a normal class. Yes. I know you have used them before. Just trust me on this one, okay?

Global Namespace

Here's a simple class.

Example 01: A simple class.

```
1  <?php
2
3  // app/Eddard.php
4
5  class Eddard
6  {
7
8  }
```

There's nothing special about it. If we want to use it then, we can do this.

Example 02: Using the class.

```php
<?php

// app/Http/routes.php

$eddard = new Eddard;
```

> Dayle, I know some PHP...

Okay, okay. Sorry.

We can think of this class as being in the 'global' namespace. I don't know if that's the right term for it, but it sounds quite fitting to me. It essentially means that the class exists without a namespace. It's just a normal class.

Simple Name-spacing

Let's create another class alongside the original, global Eddard.

Example 03: Name-spaced class.

```php
<?php

namespace Stark;

// app/another.php

class Eddard
{

}
```

Here we have another Eddard class, with one minor change. The addition of the `namespace` directive. The line `namespace Stark;` informs PHP that everything we do is relative to the `Stark` namespace. It also means that any classes created within this file will live inside the 'Stark' namespace.

Let's try to use the 'Eddard' class once again.

Example 04: Using a class.

```
<?php

// app/Http/routes.php

$eddard = new Eddard;
```

Once again, we get an instance of the first class we created in the last section. Not the one within the 'Stark' namespace. Let's try to create an instance of the 'Eddard' within the 'Stark' namespace.

Example 05: Using a class in Stark.

```
<?php

// app/Http/routes.php

$eddard = new Stark\Eddard;
```

We can instantiate a class within a namespace, by prefixing it with the name of the namespace, and separating the two with a backward (\) slash. Now we have an instance of the 'Eddard' class within the 'Stark' namespace. Aren't we magical?!

You should know that namespaces can have as many levels of hierarchy as they need to. For example.

Example 06: A horrible namespace.

```
This\Namespace\And\Class\Combination\Is\Silly\But\Works
```

The Theory of Relativity

Remember how I told you that PHP always reacts **relative** to the current namespace? Well, let's take a look at this in action.

Example 07: Instantiate within namespace.

```php
<?php

namespace Stark;

// app/Http/routes.php

$eddard = new Eddard;
```

By adding the namespace directive to the instantiation example, we have moved the execution of the PHP script into the 'Stark' namespace. Since we are inside the same namespace as the one we put 'Eddard' into, we receive the name-spaced 'Eddard' class. See how it's all relative?

Now that we have changed the active namespace, we have created a little problem. Can you guess what it is? How do we instantiate the original 'Eddard' class? The one not in the namespace.

Fortunately, PHP has a trick for referring to classes that are located in the global namespace. We simply prefix them with a backward (\) slash.

Example 08: Instantiate class from root namespace.

```php
<?php

// app/Http/routes.php

namespace Stark;

$eddard = new \Eddard;
```

With the leading backward (\) slash, PHP knows that we are referring to the 'Eddard' in the global namespace, and instantiates that one.

Use your imagination a little, like how Barney showed you. Imagine that we have another namespaced class called `Tully\Edmure`. Now we want to use this class from within the 'Stark' framework. How do we do that?

Example 09: Instantiate class in other namespace.

```php
<?php

namespace Stark;

// app/Http/routes.php

$edmure = new \Tully\Edmure;
```

One again, we need the prefixing backward slash to bring us back to the global namespace before instantiating a class from the 'Tully' namespace.

It could get tiring referring to classes within other namespaces by their full hierarchy each time. Luckily, there's a nice little shortcut we can `use`. Let's see it in action.

Example 10: Use a foreign class.

```php
<?php

namespace Stark;

use Tully\Edmure;

// app/Http/routes.php

$edmure = new Edmure;
```

Using the `use` statement, we can bring one class from another namespace into the current namespace. This will allow us to instantiate it by name only.

We can give our imported classes little nicknames like we used to in the PHP playground. Let me show you.

Example 11: Class alias.

```php
<?php

namespace Stark;

use Tully\Brynden as Blackfish;

// app/Http/routes.php

$edmure = new Blackfish;
```

By using the ‘as‘ keyword, we have given our ‘Tully/Brynden’ class the ‘Blackfish’ nickname. This will allow us to use the new nickname to identify it within the current namespace. Neat trick, right? It’s also really handy if you need to use two similarly named classes within the same namespace. For example.

Example 12: The mother of twin classes.

```php
<?php

namespace Targaryen;

use Dothraki\Daenerys as Khaleesi;

// app/Http/routes.php

class Daenerys
{

}

// Targaryen\Daenerys
$daenerys = new Daenerys;

// Dothraki\Daenerys
$khaleesi = new Khaleesi;
```

By giving the ‘Daenerys’ within the ‘Dothraki’ namespace a nickname of ‘Khaleesi’, we can use two ‘Daenerys’ classes by name only. Handy, right? The game is all about avoiding conflicts and grouping things by purpose or faction.

You can `use` as many classes as you need to.

Example 13: Using multiple classes.

```
<?php

namespace Targaryen;

use Stark\Eddard;
use Lannister\Tyrion;
use Dothraki\Daenerys;
use Snow\Jon as Bastard;
```

Structure

Namespaces aren't just about avoiding conflicts. We can also use them for organization and ownership. Let me explain with another example.

Let's say I want to create an open source library. I'd love for others to use my code; it would be great! The trouble is, I don't want to cause any problematic class name conflicts for the person using my code. That would be terribly inconvenient. Here's how I could avoid causing hassle for the wonderful, open source embracing, individual.

Example 14: Vendor prefixing.

```
Dayle\Blog\Content\Post
Dayle\Blog\Content\Page
Dayle\Blog\Tag
```

Here we have used my name to show that I created the original code, and to separate my code from that of the person using my library. Inside the base namespace, I have created some sub-namespaces to organize my application by its internal structure.

In the composer section, you will learn how to use namespaces to simplify the act of loading class definitions. I **strongly** suggest you take a look at this useful mechanism.

Limitations

In truth, I feel a little guilty for calling this sub-heading 'Limitations.' What I'm about to talk about isn't a bug.

You see, in other languages, namespaces are implemented in a similar way. Those other languages provide an additional feature when interacting with namespaces.

In Java, for example, you can import some classes into the current namespace by using the import statement with a wildcard. In Java, 'import' is equivalent to 'use', and it uses dots to separate the nested namespaces (*or packages*). Here's an example.

Example 15: Java imports.

```
import dayle.blog.*;
```

The above would import all of the classes that are located within the 'dayle.blog' package.

In PHP, you simply can't do that. You have to import each class individually. Sorry about that. Actually, why am I saying sorry? Go and complain to the PHP internals team instead. Only, be gentle. They have given us a lot of cool stuff recently.

Here's a neat trick you can use. Imagine that we have this namespace and class structure, as in the previous example.

Example 16: My blog.

```
Dayle\Blog\Content\Post
Dayle\Blog\Content\Page
Dayle\Blog\Tag
```

We can give a sub-namespace a nickname, to use it's child classes. Here's an example.

Example 17: Nickname a sub-namespace.

```
<?php

namespace Baratheon;

use Dayle\Blog as Cms;

// app/Http/routes.php

$post = new Cms\Content\Post;
$page = new Cms\Content\Page;
$tag  = new Cms\Tag;
```

This should prove useful if you need to use many classes within the same namespace. Enjoy!

Next, we will learn about Jason. No. Not Jason Lewis the Aussie scoundrel, but JSON strings. Just turn the page and you'll see what I mean!

6. JSON

What is JSON?

For the next chapter on Composer, we're going to need a basic understanding of JSON. Trust me; it doesn't hurt to learn this stuff. JSON is used by a bunch of different technologies.

JSON stands for JavaScript Object Notation. It was named this way because JavaScript was the first language to take advantage of the format.

Essentially, JSON is a human readable method of storing arrays and objects with values as strings. It is used primarily for data transfer and is a lot less verbose than some of the other options such as XML.

Commonly, it is used when the front-end part of your application requires some data from the back-end without a page reload. This is commonly achieved using JavaScript with an AJAX or 'XHR' request.

Many software APIs also serve content using this file format. Twitter's own is an excellent example of such an API.

Since version 5.2.0, PHP has been able to serialise objects and arrays to JSON. This is something I have personally abused a billion or so times and was a great addition to the language.

If you have been working with PHP for a while, you may have already used its `serialize()` method to represent a PHP object as a string. You are then able to use the `unserialize()` method to transform the string into a new instance containing the original value.

It's what we will be using JSON for. However, the advantage is that JSON can be parsed by a variety of different languages, where `serialize()`d strings can only be parsed by PHP. The additional advantage is that we (as humans and pandas) can read JSON strings, but PHP serialised strings will look like garbage.

Enough back story. Let's dive right in and have a look at some JSON.

JSON Syntax

Example 01: Sample JSON.

```
{"name":"Lushui","species":"Panda","diet":"Green Things","age":7,"colours":["red\
","brown","white"]}
```

Yay! JSON! Okay. When I said it was readable to humans, I may have forgotten to mention something. By default, JSON is stored without any white space between its values. This might make it a little more difficult to read.

This is normally used to save on bandwidth when transferring the data. Without all the extra whitespace, the JSON string will be much shorter and thus be fewer bytes to transfer.

The good news is that JSON doesn't care about whitespace or line breaks between its keys and values. Go wild! Throw all the white space you want in there to make it readable.

Sure, we could do this by hand (let's not), but there are plenty of tools out there on the web to beautify JSON. I won't choose one for you. Go hunting! You might even find extensions for your web browser to allow you to read JSON responses from web servers more easily. I highly recommend finding one of these!

Let's add some whitespace to the JSON from before to make it easier to read. (Honestly, I did do this by hand. Don't try this at home folks.)

Example 02: Beautiful JSON.

```
{
    "name":         "Lushui",
    "species":      "Panda",
    "diet":         "Green Things",
    "age":          7,
    "colours":      ["red", "brown", "white"]
}
```

Aha! There we go. We now have a JSON string representing the red panda that lives on the cover of my books. Lushui safely guards your Laravel knowledge against prying eyes.

As you can see from the example, we have some key-value pairs. Within one of the key-value pairs, we have an array. If you have used some JavaScript before you may wonder what has changed here? In fact, let's have a look at how this would be represented in JavaScript.

Example 03: A JavaScript object.

```
var lushui = {
    name:       'Lushui',
    species:    'Panda',
    diet:       'Green Things',
    age:        7,
    colours:    ['red', 'brown', 'white']
};
```

Hopefully, you will have spotted some similarities between the JavaScript and JSON snippets.

The JavaScript snippet is assigning an object literal to a variable, like this:

Example 04: Object assignment.

```
var lushui = { .. };
```

JSON is a data transfer format and not a language. It has no concept of variables. This is why you don't need the assignment within the JSON snippet.

The method of representing an object literal is very similar. This is no coincidence! As I said before, JSON was originally invented for use with JavaScript.

In both JSON and JavaScript, objects are contained within `{` two curly braces `}` and consist of key-value pairs of data. In the JavaScript variant, the keys didn't require quotes because they represent variables, but we know that JSON doesn't have variables. This is fine since we can use strings as keys, and that's what JSON does to work around this problem.

You may also have noticed that I used single quotes around the JavaScript values. This is a trap! That behaviour is perfectly acceptable with JavaScript, but JSON strings **must** be contained within double quotes. You must remember this, young padawan!

In both JavaScript and JSON, key-value pairs must be separated with a colon (`:`), and sets of key-value pairs must be separated by a comma (`,`).

JSON will support strings and numerical types. You can see that the value for Lushui's age is set to an integer value of seven.

Example 05: Wise old Lushui.

```
age: 7,
```

JSON will allow the following value types.

- Double
- Float
- String
- Boolean
- Array
- Object
- Null

Numeric values are represented without quotes. Be careful when choosing whether to quote a value or not. For example, US zip codes consist of five numbers. However, if you were to omit the quotes from a zip code, then `07702` would act as an integer and be truncated to `7702`. This has been a mistake that has taken many a life of a web-faring adventurer.

Booleans are represented by the words `true` and `false`, both without quotes much like PHP itself. As I said previously, string values are contained within double quotes **and not single quotes**.

The null value works in a similar way to PHP and is represented by the word `null` without quotes. This should be easy to remember!

We have seen objects. Much like the main JSON object itself, they are wrapped with curly braces and can contain all sorts of value types.

Arrays look very similar to their JavaScript counterpart.

Example 06: A JavaScript Array.

```
1 // JavaScript
2 ['red', 'brown', 'white']
```

Example 07: A JSON array.

```
1 ["red", "brown", "white"]
```

You will notice that I didn't add an inline comment for the JSON snippet in the above example. That's because JSON doesn't support commenting since it's used for data transfer. Keep that in mind!

As you can see, the arrays for both are wrapped within `[` square braces `]`, and contain a list of comma (`,`) separated values with no indexes. The only difference is that double quotes must be used for strings within JSON. Are you getting bored of me saying that yet?

As I mentioned above, the values that can be contained within JSON include both objects and arrays. The clever chaps amongst my readers (aka all of you) may have realised that JSON can support nested objects and arrays. Let's have a look at that in action!

Example 08: Nesting.

```
{
    "an_object": {
        "an_array_of_objects": [
            { "The": "secret" },
            { "is": "that" },
            { "I": "still" },
            { "love": "shoes!" }
        ]
    }
}
```

Okay. Take a deep breath. Here we have a JSON object containing an object containing an array of objects. This is perfectly fine, and will allow JSON to represent some complex data collections.

JSON and PHP

As mentioned previously since version 5.2.0 PHP has provided support for serialising and unserializing data to and from the JSON format. Let's go ahead and have a look at that in action.

Serialise a PHP array to JSON

To serialise a PHP value we need only use the `json_encode()` method.

Example 09: Serialise an array.

```
<?php

$truth = ['panda' => 'Awesome!'];
echo json_encode($truth);
```

The result of this snippet of code would be a JSON string containing the following value.

Example 10: Output.

```
{"panda":"Awesome!"}
```

Great! That's more like it. Let's make sure that we can convert this data back into a format that can be understood by PHP.

Unserialise a PHP array from JSON

For this we will use the `json_decode()` method. I bet you didn't see that one coming?

Example 11: Unserialise a JSON string.

```php
<?php

$truth = json_decode('{"panda":"Awesome!"}');
echo $truth['panda'];
```

Awesome! We go... wait. What?

Example 12: Error.

```
Fatal error: Cannot use object of type stdClass as array in ...
```

You see, the `json_decode()` method doesn't return our JSON as a PHP array; it uses a `stdClass` object to represent our data. Let's instead access our object key as an object attribute.

Example 13: Object access.

```php
<?php

$truth = json_decode('{"panda":"Awesome!"}');
echo $truth->panda;

// Awesome!
```

Great! That's what we wanted. If we wanted an array instead, then PHP provides some ways to convert this object into one, but fortunately `json_decode()` has another trick up its sleeve!

If you provide `true` as the second parameter to the function, we will receive our PHP array exactly as expected. Thanks `json_decode()`!

Example 14: Array access.

```php
<?php

$truth = json_decode('{"panda":"Awesome!”}', true);
echo $truth['panda'];

// Awesome!
```

Huzzah!

So you might be wondering why I just wrote a gigantic chapter on JSON within a Laravel book. Furthermore, you are probably asking why I'm choosing to answer this question at the end of the chapter!?

It's just more fun that way.

In the next section, we will be taking a look at the package manager for PHP. When we start looking at Composer, you will understand why a knowledge of JSON is so important.

7. Composer

In this chapter, we're going to take a pause from Laravel to examine another tool in the PHP world. Don't worry. Laravel does make use of this tool, so if you stick it out, it will make you a better Laravel developer.

Composer is something special in the world of PHP. It has changed the way we handle application dependencies and quelled the tears of many PHP developers.

You see, in the olden days, when you wanted to build an application that relied on third party dependencies you would have to install them with PEAR or PECL. These two dependency managers both have a very limited set of outdated dependencies and have been a thorn in the side of PHP developers for a long time.

When a package is finally available, you could download a particular version, and it would be installed on your system. However, the dependency is linked to PHP rather than your application itself. What this means is that if you had two applications that required different versions of the same dependencies... well, you're gonna have a bad time.

Enter Composer, king of the package managers. First, let's think about packages. What are they?

Let's forget about the concept of 'applications' and 'projects' for now. The tool that you are building is called a package. Imagine a little box containing everything needed to run your application.

This box requires only one piece of paper (file) inside for it to be registered as a package.

Configuration

You have learned about JSON in the last chapter, right? You are ready for this now! Remember that I told you JSON was used for data transfer between web applications? Well, I lied. It's not that I'm nasty. It just made it easier to teach the topic with a smaller scope of its ability. I do this a lot. Expect many lies!

Do you remember how JSON represents a complex piece of data? Why can't we use it within flat files to provide configuration? That's precisely what the Composer guys thought. Who are we to argue with them?

JSON files use the `.json` extension. Composer expects its configuration to live at the root of your package using the filename `composer.json`. Remember this! Laravel will use this file often.

Let's open it up and start entering some information about our package.

Example 01: Composer.json.

```
{
    "name":             "marvel/xmen",
    "description":      "Mutants saving the world for people who hate them.",
    "keywords":         ["mutant", "superhero", "bald", "guy"],
    "homepage":         "http://marvel.com/xmen",
    "time":             "1963-09-01",
    "license":          "MIT",
    "authors": [
        {
            "name":         "Stan Lee",
            "email":        "stan@marvel.com",
            "homepage":     "http://marvel.com",
            "role":         "Genius"
        }
    ]
}
```

Right. Here we have a `composer.json` file at the root of a package for the X-Men. Why the X-Men? They are awesome. That's why.

Truth be told, all of the **options** (keys) in this file are optional. You would usually provide the above information if you intended to redistribute the package or release it into the wild.

To be quite honest with you, I normally go ahead and fill in this information. It doesn't do any harm. The configuration above is used to identify the package. I have omitted a few keys that I felt were reserved for special circumstances. If you are curious about any additional config, then I would recommend checking out the Composer website[1] which contains a wealth of information and documentation.

I also found this handy cheat sheet[2] online, which may be useful for newcomers to Composer when creating new packages. Mouse over each line to discover more about the configuration items.

Let's have a closer look at the configuration we have created for the X-Men package.

[1] http://getcomposer.org/
[2] http://composer.json.jolicode.com/

Example 02: Package name.

```
1 {"name": "marvel/xmen"}
```

This is the package name. If you have used Github[3] then the name format will be familiar to you, but I'm going to explain it anyway.

The package name consists of two words separated by a forward slash (/). The part before the forward slash represents the owner of the package. In most circumstances, developers will use their Github username as the owner, and I fully agree with this notion. You can, however, use any name you like. Be sure to keep it consistent across all packages that belong to you.

The second part of the name string is the package name. Keep it straightforward and descriptive. Many developers choose to use the repository name for the package when hosted on Github, and once again I fully agree with this system.

Example 03: Package description.

```
1 {"description": "Mutants saving the world for people who hate them."}
```

Provide a brief description of the functionality of the package. Remember to keep it simple. If the package is intended for open source, then you can go into detail within the `README` file for your repository. If you want to keep some personal documentation, then this isn't the place for it. Maybe get it tattooed on your back and keep a mirror handy? That makes the most sense to me. Sticky notes will also work well, though.

Example 04: Keywords.

```
1 {"keywords": ["mutant", "superhero", "bald", "guy"]}
```

These keywords are an array of strings used to represent your package. They are similar to tags within a blogging platform and essentially serve the same purpose. The tags will provide useful search metadata for when your package is listed within a repository.

Example 05: Homepage.

```
1 {"homepage": "http://marvel.com/xmen"}
```

The homepage configuration is useful for packages due to be open-sourced. You could use the homepage for the project, or maybe the Github repository URL? Whichever you feel is more informative.

Once again, I must remind you that all of these configuration options are optional. Feel free to omit them if they don't make sense for your package.

[3] http://github.com

Example 06: Release date.

```
{"time": "1963-09-01"}
```

This is one of those options that I don't see very often. According to the cheat sheet, it represents the release date of your application or library. I'd imagine that it's not required in most circumstances because most packages are hosted on Github or some other version control site. These sites normally date each commit, each tag, and other useful events.

Formats accepted for the time configuration are `YYYY-MM-DD` and `YYYY-MM-DD HH:MM:SS`. Go ahead and provide these values if you feel like it!

Example 07: License information.

```
{"license": "MIT"}
```

If your package is due to be redistributed, then you will want to provide a license with it. Without a license, many developers will not be able to use the package at all due to legal restrictions. Choose a license that suits your requirements, but isn't too restrictive to those hoping to use your code. The Laravel project uses the MIT license which offers lots of freedom.

Most licenses require you to keep a copy of the license within the source repository, but if you also provide this configuration entry within the `composer.json` file then the package repository will be able to list the package by its license.

The authors section of the configuration provides information about the package authors and can be useful for package users wishing to make contact.

Note that the authors section will allow an array of authors for collaborative packages. Let's have a look at the options given.

Example 08: The author object.

```
{
    "authors": [
        {
            "name":         "Stan Lee",
            "email":        "stan@marvel.com",
            "homepage":     "http://marvel.com",
            "role":         "Genius"
        }
    ]
}
```

Use an object to represent each author. Our example only has one author. Let's take a look at Stan Lee. Not only does he have a cameo in every Marvel movie, but he's also managed to make it into my book. What a cheeky old sausage!

Example 09: Author name.

```
1  {"name": "Stan Lee"}
```

I don't know how to simplify this line. If you are having trouble understanding it, then you might want to consider closing this book, and instead pursue a career in sock puppetry.

Example 10: Author email.

```
1  {"email": "stan@marvel.com"}
```

Be sure to provide a valid email address so that you can be contacted if the package is broken.

Example 11: Author homepage.

```
1  {"homepage": "http://marvel.com"}
```

This time, a personal homepage can be provided, go ahead and leech some hits!

Example 12: Author role.

```
1  {"role": "Genius"}
```

The role option defines the author's role within the project. For example, developer, designer, or even sock puppetry artist. If you can't think of something accurate, then put something funny.

That's all you need to describe your package. Let's look at something more interesting. Dependency management!

Dependency Management

You have a box that will contain the X-Men. There aren't a lot of mutants in that box yet, are there? To build a great superhero team (*application*) you will need to enlist the support of other mutants (*3rd party dependencies*). Let's take a look at how Composer will help us accomplish this.

Example 13: The require block.

```
{
    "name":             "marvel/xmen",
    "description":      "Mutants saving the world for people who hate them.",
    "keywords":         ["mutant", "superhero", "bald", "guy"],
    "homepage":         "http://marvel.com/xmen",
    "time":             "1963-09-01",
    "license":          "MIT",
    "authors": [
        {
            "name":         "Stan Lee",
            "email":        "stan@marvel.com",
            "homepage":     "http://marvel.com",
            "role":         "Genius"
        }
    ],
    "require": {

    }
}
```

We now have a new section within our `composer.json` called 'require'. This will be used to list our dependenc... mutants. From now on I'll be omitting the rest of the configuration, and just showing the require block to shorten the examples. Make sure you know where it lives!

We know that the X-Men will depend on:

- Wolverine
- Cyclops
- Storm
- Gambit

There are loads of others, but these guys are cool. We will stick with them for now. You see, we could copy the source files for these guys into our application directly, but then we would have to update them ourselves with any upstream changes. That could get boring. Let's add them to the `require` section so that Composer will manage them for us.

Example 14: Require packages.

```
{
    "require": {
        "xmen/wolverine":   "1.0.0",
        "xmen/cyclops":     "1.0.1",
        "xmen/storm":       "1.2.0",
        "xmen/gambit":      "1.0.0"
    }
}
```

Here we are listing the packages for our mutant dependencies and the versions that we would like to use. In this example, they all belong to the same owner as the X-Men package, but they could just as easily belong to another person.

Most redistributable packages are hosted on a version control website such as Github[4] or Bitbucket[5]. Version control repositories often have a tagging system where we can define stable versions of our application. For example with git, we can use the following command.

Example 15: Git tag.

```
git tag -a 1.0.0 -m 'First version.'
```

We have created version `1.0.0` of our application. This is a stable release that people can depend on.

Let's have a closer look at the Gambit dependency.

Example 16: Gambit.

```
{"xmen/gambit": "1.0.0"}
```

You should know by now that Composer package names consist of an owner and a package nickname separated by a forward slash (`/`) character. With this information, we know that this is the `gambit` package written by the `xmen` owner.

Within the `require` section, the key for each item is the package name, and the value represents the required version.

In the case of Gambit, the version number matches up to the tag available on Github where the code is versioned. Do you see how the versions of dependencies are now specific to our application, and not the whole system?

[4] http://github.com
[5] http://bitbucket.org

You can add as many dependencies as you like to your project. Go ahead, add a billion! Prove me wrong.

Listen, do you want to know a secret? Do you promise not to tell? Woah, oh-oh. Closer, let me whisper in your ear. Say the words you long to hear...

> Your dependencies can have their own dependencies.

That's right! Your dependencies are also Composer packages. They have their own `composer.json` files. This means that they have their own `require` section with a list of dependencies, and those dependencies might even have more dependencies. And that dep... well. You get the idea!

Composer will manage and install these nested dependencies for you. How fantastic is that? Wolverine might need `tools/claws`, `tools/yellow-mask`, and `power/regeneration` but you don't have to worry about that. As long as you `require` the `xmen/wolverine` package, then Composer will take care of the rest.

As for dependency versions, they can assume some different forms. For example, you might not care about minor updates to a component. In which case, you could use a wildcard within the version, like this:

Example 17: Wildcard version.

```
1 {"xmen/gambit": "1.0.*"}
```

Now Composer will install the latest version that starts with `1.0`. For example, if Gambit had versions `1.0.0` and `1.0.1`, then `1.0.1` would be installed.

Your package might also have a minimum or maximum boundary for package versions. This can be defined using the `greater-than` and `less-than` operators.

Example 18: Greather than.

```
1 {"xmen/gambit": ">1.0.0"}
```

The above example would be satisfied by any versions of the `xmen/gambit` package that have a greater version number than `1.0.0`.

Example 19: Less than.

```
1 {"xmen/gambit": "<1.0.0"}
```

Similarly, the `less-than` operator is satisfiable by packages less than the version `1.0.0`. This will allow your package to specify a maximum version dependency.

Example 20: Or equal to.

```
{
    "xmen/gambit": "=>1.0.0",
    "xmen/gambit": "=<1.0.0"
}
```

Including an equals sign `=` along with a comparison operator will result in the comparative version being added to the list of versions which satisfy the version constraint.

Occasionally, you may wish to enter more than one version, or provide a range value for a package version. More than one version constraint can be added by separating each constraint with a comma (`,`). For example:

Example 21: Multiple version constraints.

```
{"xmen/gambit": ">1.0.0,<1.0.2"}
```

The above example would be satisfied by the `1.0.1` version.

If you don't want to install stable dependencies, for example, you might be the type that enjoys bungee jumping or skydiving; then you might want to use bleeding edge versions. Composer can target `branches` of a repository using the following syntax.

Example 22: Branches.

```
{"xmen/gambit": "dev-branchname"}
```

For example, if you wanted to use the current codebase from the develop branch of the Gambit project on Github, then you would use the `dev-develop` version constraint.

Example 23: Develop branch.

```
{"xmen/gambit": "dev-develop"}
```

These development version constraints will not work unless you have a correct minimum stability setting for your package. By default, Composer uses the `stable` minimum compatibility flag, which will restrict its dependency versions to stable, tagged releases.

If you would like to override this option, simply change the `minumum-stability` configuration option within your `composer.json` file.

Example 24: Stability.

```
{
    "require": {
        "xmen/gambit": "dev-master"
    },
    "minimum-stability": "dev"
}
```

There are other values available for the minimum stability setting, but explaining those would involve delving into the depths of version stability tags. I don't want to overcomplicate this chapter by looking at those. Instead, I'd suggest looking at the Composer documentation for package versions[6] to find additional information on the topic.

Sometimes, you may find yourself needing to use dependencies that only relate to the development of your application. These dependencies might not be required for the day-to-day use of your application in a production environment.

Composer has got you covered in this situation thanks to its `require-dev` section. Let's imagine for a moment that our application will require the Codeception testing framework[7] to provide acceptance tests. These tests won't be any use in our production environment, so let's add them to the `require-dev` section of our `composer.json`.

Example 25: Development dependencies.

```
{
    "require": {
        "xmen/gambit": "dev-master"
    },
    "require-dev": {
        "codeception/codeception": "1.6.0.3"
    }
}
```

The `codeception/codeception` package will now only be installed if we use the `--dev` switch with Composer. There will be more on this topic in the installation and usage section.

As you can see, the `require-dev` section uses the same format as the `require` section. In fact, there are other sections which use the same format. Let's have a quick look at what's available.

[6] http://getcomposer.org/doc/01-basic-usage.md#package-versions
[7] http://codeception.com/

Example 26: Conflicting packages.

```
{
    "conflict": {
        "marvel/spiderman": "1.0.0"
    }
}
```

The `conflict` section contains a list of packages that would not work happily alongside our package. Composer will not let you install these packages side by side.

Example 27: Replacements.

```
{
    "replace": {
        "xmen/gambit": "1.0.0"
    }
}
```

The replace section informs you that this package can be used as a replacement for another package. This is useful for packages that have been forked from another but provide the same functionality.

Example 28: Provisions.

```
{
    "provide": {
        "xmen/gambit": "1.0.0"
    }
}
```

This section indicates packages that have been provided alongside your package. If the Gambit packages source were included within our main package, then it would be of little use to install it again. Use this section to let Composer know which packages have been embedded within your primary package. Remember, you need not list your package dependencies here. Anything found in `require` doesn't count.

Example 29: Suggestions.

```
{
    "suggest": {
        "xmen/gambit": "1.0.0"
    }
}
```

Your package might have some extra packages that enhance its functionality but aren't strictly required. Why not add them to the `suggest` section? Composer will mention any packages in this section as suggestions to install when running the Composer install command.

That's all I have on dependencies. Let's take a look at the next piece of Composer magic. Autoloading!

Auto Loading

By now we have the knowledge to enable Composer to retrieve our package dependencies for us, but how do we go about using them? We could `require()` the source files ourselves within PHP, but that requires knowing exactly where they live.

Ain't nobody got time fo' dat. Composer will handle this for us. If we tell Composer where our classes are located, and what method can be used to load them, then it will generate its autoloader. This can be used by our application to load class definitions.

Actions speak louder than words, so let's dive right in with an example.

Example 30: Autoload.

```
{
    "autoload": {

    }
}
```

This is the section in which all of our autoloading configurations will be contained. Simple, right? Great! No sock puppetry for you.

Let's have a look at the most simple of loading mechanisms, the `files` method.

Example 31: File loading.

```
{
    "autoload": {
        "files": [
            "path/to/my/firstfile.php",
            "path/to/my/secondfile.php"
        ]
    }
}
```

The `files` loading mechanism provides an array of files which will be loaded when the Composer autoloader component is loaded within your application. The file paths are considered relative to your project's root folder. This loading method is effective, but not very convenient. You won't want to add every single file manually for a large project. Let's take a look at some better methods of loading more significant amounts of files.

Example 32: Class mapping.

```
{
    "autoload": {
        "classmap": [
            "src/Models",
            "src/Controllers"
        ]
    }
}
```

The `classmap` is another loading mechanism which accepts an array. This time, the array consists of a number of directories which are relative to the root of the project.

When generating its autoloader code, Composer will iterate through the directories looking for files which contain PHP classes. These files will be added to a collection which maps a file path to a class name. When an application is using the Composer autoloader and attempts to instantiate a class that doesn't exist, Composer will step in and load the required class definition using the information stored in its map.

There is, however, a downside to using this loading mechanism. You will need to use the `composer dump-autoload` command to rebuild the class map every time you add a new file. Fortunately, there is another loading mechanism which is intelligent enough not to require a map. Let's learn about PSR-0 class loading.

PSR-0 class loading was first described in the PSR-0 PHP standard and provides an easy way of mapping PHP namespaced classes to the files that they are contained in.

Know that if you add a `namespace` declaration to a file containing your class, like this:

Example 33: A namespaced class.

```php
<?php

namespace Xmen;

class Wolverine
{
    // ...
}
```

Then the class becomes `Xmen\Wolverine`, and as far as PHP is concerned, it is an entirely different animal to the `Wolverine` class.

Using `PSR-0` autoloading, the `Xmen\Wolverine` class would be located in the file `Xmen/-Wolverine.php`.

See how the namespace matches up with the directory that the class is contained within? The `Xmen` name-spaced `Wolverine` class is located within the `Xmen` directory.

You should also note that the filename matches the class name, including the uppercase character. Having the filename match the class name is essential for `PSR-0` autoloading to function correctly.

Namespaces may have several levels. For example, consider the following class.

Example 34: Multiple levels.

```php
<?php

namespace Super\Happy\Fun;

class Time
{
    // ...
}
```

The `Time` class is located within the `Super\Happy\Fun` namespace. PHP will recognize it as `Super\Happy\Fun\Time` and not `Time`.

This class would be located at the following file path.

Example 35: File path.

```
Super/Happy/Fun/Time.php
```

Once again, see how the directory structure matches the namespace? Also, you will notice that the file is named the same as the class.

That's all there is to `PSR-0` autoloading. It's quite simple really! Let's have a look at how it can be used with Composer to simplify our class loading.

Example 36: PSR-0 Autoloading.

```
{
    "autoload": {
        "psr-0": {
            "Super\\Happy\\Fun\\Time": "src/"
        }
    }
}
```

This time, our `psr-0` autoloading block is an object rather than an array. This is because it requires both a key and value.

The key for each value in the object represents a namespace. Don't worry about the double backward slashes. They are used because a single slash would represent an escape character within JSON. Remember this when mapping namespaces in JSON files!

The second value is the directory in which the namespace is mapped. I have found that you don't need the trailing slash, but many examples like to use it to denote a directory.

This next bit is crucial and is a serious 'gotcha' for a lot of people. Please read it carefully.

The second parameter is not the directory in which classes for that namespace are located. Instead, it is the directory which ***begins*** the namespace to directory mapping. Let's take a look at the previous example to illustrate this.

Remember the super happy fun time class? Let's have another look at it.

Example 37: Here we go again.

```
<?php

namespace Super\Happy\Fun;

class Time
{
    // ...
}
```

We now know that this class would be located in the `Super/Happy/Fun/Time.php` file. With that in mind, consider the following autoload snippet.

Example 38: Autoloading.

```
{
    "autoload": {
        "psr-0": {
            "Super\\Happy\\Fun\\Time": "src/"
        }
    }
}
```

You might expect Composer to look in `src/Time.php` for the class. This would be **incorrect**, and the class would not be found.

Instead, the directory structure should exist in the following format.

Example 39: Directory structure.

```
src/Super/Happy/Fun/Time.php
```

This is something that catches so many people out when first using Composer. I can't stress enough how important this fact is to remember.

If we were to run an installation of Composer now, and later add a new class `Life.php` to the same namespace, then we would not have to regenerate the autoloader. Composer knows exactly where classes with that namespace exist, and how to load them. Great!

There's one other useful loading mechanism that was added much later into Composer's lifecycle. It's called `PSR-4` namespacing and is a little more convenient than `PSR-0` loading.

Here's an example require block for PSR-4 name-spacing.

Example 40: PSR-4 Autoloading.

```
{
    "autoload": {
        "psr-4": {
            "Dayle\\Blog\\": "src"
        }
    }
}
```

Here, we've mapped the `Dayle\Blog` namespace to the `src` directory. Don't forget to suffix the namespace with two slashes! This means that we don't have to create the `Dayle/Blog` directory structure, as all classes will be relative to the `src` directory. For example, consider the following class.

Example 41: A Class.

```
Dayle\Blog\Content\Post
```

It would be located at the following location on disk.

Example 42: Class location.

```
src/Content/Post
```

We've specified the namespace prefix in our autoloader, so we don't need to create the `Dayle` and `Blog` subfolders. This makes it much easier to see our project in the tree view of our text editor.

You might wonder why I put my namespaced files in an `src` folder? This is a common convention when writing Composer based libraries. In fact, here is a common directory/file structure for a Composer package.

Example 43: A typical package.

```
src/            (Classes.)
tests/          (Unit/Acceptance tests.)
docs/           (Documentation.)
composer.json
```

Feel free to keep to this standard, or do whatever makes you happy. Laravel provides locations for its classes which we will examine in a later chapter.

Now that you have learned how to define your autoload mechanisms, it's time that we looked at how to install and use Composer so that you can start taking advantage of its autoloader.

Installation

The Homestead machine that we configured in the installation chapter will have composer pre-installed. You'll need to SSH into the machine to use it.

First, make, sure that the machine is running by using `vagrant up`. Next, you'll use `vagrant ssh` to create an SSH connection to the machine. You're now ready to begin using Composer.

That was easy, wasn't it?

Usage

Let's assume that we have created the following `composer.json` file in our package directory.

Example 44: Example composer.json.

```
{
    "name":              "marvel/xmen",
    "description":       "Mutants saving the world for people who hate them.",
    "keywords":          ["mutant", "superhero", "bald", "guy"],
    "homepage":          "http://marvel.com/xmen",
    "time":              "1963-09-01",
    "license":           "MIT",
    "authors": [
        {
            "name":          "Stan Lee",
            "email":         "stan@marvel.com",
            "homepage":      "http://marvel.com",
            "role":          "Genius"
        }
    ],
    "require": {
        "xmen/wolverine":    "1.0.0",
        "xmen/cyclops":      "1.0.1",
        "xmen/storm":        "1.2.0",
        "xmen/gambit":       "1.0.0"
    },
    "autoload": {
        "classmap": [
            "src/Xmen"
        ]
```

```
    }
}
```

Let's go ahead and use the `install` command to install all of our package dependencies and generate our autoloader.

Example 45: Composer install.

```
composer install
```

The output we get from Composer will be similar to:

Example 46: Example output.

```
Loading composer repositories with package information
Installing dependencies

  - Installing tools/claws (1.1.0)
    Cloning bc0e1f0cc285127a38c232132132121a2fd53e94

  - Installing tools/yellow-mask (1.1.0)
    Cloning bc0e1f0cc285127a38c6c12312325dba2fd53e95

  - Installing power/regeneration (1.0.0)
    Cloning bc0e1f0cc2851213313128ea88bc5dba2fd53e94

  - Installing xmen/wolverine (1.0.0)
    Cloning bc0e1f0cc285127a38c6c8ea88bc523523523535

  - Installing xmen/cyclops (1.0.1)
    Cloning bc0e1f0cc2851272343248ea88bc5dba2fd54353

  - Installing xmen/storm (1.2.0)
    Cloning bc0e1f0cc285127a38c6c8ea88bc5dba2fd53343

  - Installing xmen/gambit (1.0.0)
    Cloning bc0e1f0cc285127a38c6c8ea88bc5dba2fd56642

Writing lock file
Generating autoload files
```

Remember that these are fake packages used as an example. Downloading them won't work! They are however more fun since they are X-men! Yay!

Why are there seven packages installed when I only listed four? Well, you forget that Composer automatically manages the dependencies of dependencies. The three extra packages are dependencies of the `xmen/wolverine` package.

I'd imagine you are probably wondering where these packages have been installed? Composer creates a `vendor` directory in the root of your project to contain your package's source files.

The package `xmen/wolverine` can be found at `vendor/xmen/wolverine`, where you will find its source files along with its own `composer.json`.

Composer also stores some of its files relating to the autoload system in the `vendor/composer` directory. Don't worry about it. You will never have to edit it directly.

So how do we take advantage of the awesome autoloading abilities? Well, the answer to that is even simpler than setting up the autoloading itself. Simply `require()` or `include()` the `vendor/autoload.php` file within your application. For example:

Example 47: Require the autoloader.

```
1  <?php
2
3  require 'vendor/autoload.php';
4
5  // Your awesome application bootstrap here!
```

Great! Now you can instantiate a class belonging to one of your dependencies. For example:

Example 48: Instantiate a class.

```
1  <?php
2
3  $gambit = new \Xmen\Gambit;
```

Composer will do all the magic and autoload the class definition for you. How fantastic is that? No more littering your source files with thousands of `include()` statements.

If you have added a file in a class mapped directory, you will need to run a command before Composer can load it.

Example 49: Rebuild class map.

```
composer dump-autoload
```

The above command will rebuild all mappings and create a new `autoload.php` for you.

What if we want to add another dependency to our project? Let's add `xmen/beast` to our `composer.json` file.

Example 50: Add another dependency.

```
{
    "name":             "marvel/xmen",
    "description":      "Mutants saving the world for people who hate them.",
    "keywords":         ["mutant", "superhero", "bald", "guy"],
    "homepage":         "http://marvel.com/xmen",
    "time":             "1963-09-01",
    "license":          "MIT",
    "authors": [
        {
            "name":         "Stan Lee",
            "email":        "stan@marvel.com",
            "homepage":     "http://marvel.com",
            "role":         "Genius"
        }
    ],
    "require": {
        "xmen/wolverine":   "1.0.0",
        "xmen/cyclops":     "1.0.1",
        "xmen/storm":       "1.2.0",
        "xmen/gambit":      "1.0.0",
        "xmen/beast":       "1.0.0"
    },
    "autoload": {
        "classmap": [
            "src/Xmen"
        ]
    }
}
```

Now we need to run `composer install` again so that Composer can install our newly added package.

Example 51: Install the new dependency.

```
Loading composer repositories with package information
Installing dependencies

  - Installing xmen/beast (1.1.0)
    Cloning bc0e1f0c34343347a38c232132132121a2fd53e94

Writing lock file
Generating autoload files
```

Now `xmen/beast` has been installed and we can use it right away. Smashing!

You may have noticed the following line in the output from the `composer install` command.

Example 52: The lock file.

```
Writing lock file
```

You might also have noticed that Composer has created a file called `composer.lock` at the root of your application. What's that for I hear you cry?

The `composer.lock` file contains the information about your package at the time that the last `composer install` or `composer update` was performed. It also contains a list of the **exact** version of each dependency that has been installed.

Why is that? It's simple. Whenever you use `composer install` when a `composer.lock` file is present in the directory; it will use the version numbers contained within the file instead of pulling down fresh versions of each dependency.

This means that if you version your `composer.lock` file along with your application source (and I highly recommend this) when you deploy to your production environment, it will be using the *exact* same versions of dependencies that have been tried and tested in your local development environment. This means you can be sure that Composer won't install any dependency versions that might break your application.

Note that you should never edit the `composer.lock` file manually.

While we are on the topic of dependency versions, let's learn how to update them. For example, if we had the following requirement in our `composer.json` file.

Example 53: Dependency version.

```
{"xmen/gambit": "1.0.*"}
```

Composer might install version `1.0.0` for us. However, what if the package was updated to `1.0.1` a few days later?

We can use the `composer update` command to update all of our dependencies to their latest versions. Let's have a look at the output.

Example 54: Composer update.

```
$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)

      - Installing xmen/gambit (1.0.1)
        Cloning bc0e1f0cc285127a38c6c8ea88bc5dba2fd56642

Generating autoload files
```

Great! The `xmen/gambit` package has been updated to its latest version, and our `composer.lock` file has been updated.

If we wanted to update only that one dependency rather than all of them, we could specify the package name when using the `update` command. For example:

Example 55: Update a single dependency.

```
composer update xmen/gambit
```

Wait, what's that `(including require-dev)` bit mean? Remember the `require-dev` section in the `composer.json` file where we list our development only dependencies? Composer expects the update command to be only executed within a secure development or testing environment. For this reason, it assumes that you will want your development dependencies, and downloads them for you.

If you don't want it to install development dependencies, then you can use the following switch.

Example 56: Do not install developer dependencies.

```
composer update --no-dev
```

Also, if you would like to install developer dependencies when using the `install` command, simply use the following switch.

Example 57: Force install developer dependencies.

```
composer install --dev
```

The last thing you should know is that you can use the `composer self-update` command to update the Composer binary itself. Be sure to use `sudo` if you have installed it globally.

Example 58: Update the Composer binary.

```
sudo composer self-update
```

That just about covers all the Composer knowledge we will need when working with Laravel. It has been a lot of information to take in, and a long chapter for these weary fingers, but I hope you have gained something from it.

If you feel that a particular topic needs expanding, then please be sure to let me know! In the next chapter, we're going to learn how to configure our Laravel application. Get your configuration gloves on!

8. Configuration

Configuration is an important part of any application. We won't always want our application to have the same settings in every environment. If we decide to give our application to other users, they may wish to configure it for themselves. What if they are using a different database server to us?

Let's take a look at how Laravel will allow us to specify our configuration values.

Configuration Files

We'll start by taking a look at the `config` directory at the root of our project.

Example 01: Configuration files.

```
app.php
auth.php
broadcasting.php
cache.php
compile.php
database.php
filesystems.php
mail.php
queue.php
services.php
session.php
view.php
```

You'll find a bunch of PHP files. Many of them are named after the framework's components. These are the default configuration files that ship with a fresh installation of Laravel. Now pick a file. Any file. Pick your favourite, and open it in your editor.

What's inside? Why just normal PHP arrays! You see, Laravel uses PHP arrays for its configuration. The array keys represent the configuration name, and the values are, of course, our setting values. With Laravel, we can access these settings using the configuration facade or `config()` helper. Here's an example.

Example 02: Accessing configuration values.

```
// Using a facade.
$debug = Config::get('app.debug', false);

// Using a helper function.
$debug = config('app.debug', false);
```

The signature to both techniques is identical. The first parameter is the name of the config key that we wish to retrieve. The configuration key is made up of the filename of the configuration file that we would like to check, followed by a period character, and then the array key of the setting using 'dot' notation. Here we're requesting the 'debug' array key from the 'app.php' file.

Don't forget to add `use Config;` to the top of your class if it's namespaced. You haven't forgotten the namespacing chapter already, have you?

The second parameter to both functions is the default value. Should a configuration value be missing, then we'll receive the default value instead.

In a later chapter, we will learn how to inject the configuration component into our classes to prevent having to rely on global functions or facades.

If you'd like to create your own configuration files, then simply drop them in the `config` directory. Laravel will load them automatically and will allow to you use them in the same way as its own configuration files. When writing applications for others, be sure to add as much configuration as possible. It's great when you can get an application to work *your* way.

Environmental Variables

While rooting around in the configuration files, you might have discovered a little function called `env()`. The `env()` function is the new way of making different configuration possible in different environments. For example, you won't want your development application to be using the same database as the live website!

Since the launch of version 5, Laravel no longer uses environment-based subfolders for configuration. Instead, it makes use of the 'DotEnv' library, to allow for configuration values to be loaded from environmental variables. Setting environmental variables in your shell can be a little cumbersome. For example, we wouldn't want to be doing this all the time.

Example 03: Set an environmental variable.

```
export DB_HOST="127.0.0.1"
```

Trust me; it's even uglier on windows. Environmental variables are the method of configuration that is preferred by system administrators since it's easy to script into newly provisioned web servers. However, if we are working with configuration values on a day to day basis, then we probably don't want to be exporting them each time.

Instead, we have access to a `.env` file in the root of our project. Let's take a look at that file.

Example 04: Our .env file.

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=base64:a0hRMNEdGdWW6EUVCn3vIu8VFQiJc113CMciFkJ+pcw=
APP_URL=http://localhost

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

CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

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

The `.env` file is very simple. Keys on the left (usually in capitals) and values on the right. Here we see a bunch of default values that are provided by the framework. You

can see our database, mail, and cache server settings. If we want our configuration values to be based on our environment, then it's best to add them to this file.

The `env()` function can be used in our PHP configuration files to retrieve values from our `.env` file. This makes our configuration layer a two-step process, but will allow us to use a different `.env` in multiple environments. The `env()` function will accept the same parameters as the `config()` function. Here's an example.

Example 05: The env() function.

```
1 $host = env('DB_HOST', '127.0.0.1')
```

Here we are requesting the host database setting, but providing a default value of localhost for when it isn't present in our `.env` file or environmental variables. It's worth noting that environmental variables will always take priority over values placed in the `.env` file.

People don't usually store their `.env` file under version control. Instead, they prefer that users provide their own configuration values. Still, it's useful to know what `.env` values are available, isn't it? To solve this problem, many application developers will version a file called `.env.dist` with their application, which will contain all of the available `.env` keys, and some universal configuration values. Anyone working on your application can then copy `.env.dist` to `.env` to start working on the project. Pretty handy, right?

> So why not just use `env()` everywhere instead of `config()`?

Great question, and it shows that you're listening intently! If you wanted to, you could certainly stick to `env()`, however, this way you won't be able to make use of the caching functionality of Laravel's configuration layer. You see, Laravel can cache your configuration files to make the framework more performant. However, it can't cache values from environmental variables, so if you choose to use `env()` outside of the PHP configuration files then you might end up getting yourself in a caching mess. We don't want that, do we?

> I guess not!

Great! Then the rule of thumb to use is that `env()` should only be seen in the `config` directory. That's not too difficult, is it?

If a third party package comes with its own configuration files, simply use `php artisan vendor:publish` to copy them into your configuration directory.

Configuration Caching

As mentioned in a previous section, Laravel can cache our configuration values to allow it to load much more quickly. This is a feature that's best used in a production environment.

To cache our configuration, first, we need to head to the root of our project (this is `/vagrant` in the homestead box) and run an Artisan CLI command.

```
1 php artisan config:cache
```

Our configuration will be cached, and the configuration files themselves will be bypassed. If we've added some new configuration and would like to clear our cache, we need only run the `config:clear` command.

```
1 php artisan config:clear
```

You're probably not going to need this functionality for your local development, and not while learning the framework, but it's good to know that it's there if we need it, isn't it?

In the next chapter, we'll start writing some code. Are you excited? Then flip the page to learn about routing!

9. Basic Routing

Let's start by taking a look at a request made to the Laravel framework.

Example 01: A URL

```
http://homestead.app/my/page
```

In this example, we are using the 'HTTP' protocol (used by most web browsers) to access your Laravel application hosted on `homestead.app`. The `my/page` portion of the URL is what we will use to route web requests to the appropriate logic.

I'll go ahead and throw you in at the deep end. Routes are defined in the `app/Http/routes.php` file, so let's go ahead and create a route that will listen for the request we mentioned above.

Defining Routes

Example 02: Our first route.

```
<?php

// app/Http/routes.php

Route::get('my/page', function () {
    return 'Hello world!';
});
```

Next, enter `http://homestead.app/my/page` into your web browser, replacing `homestead.app` with the address to your Laravel application.

If everything has been configured correctly, you will now see the words 'Hello world!' in wonderful Times New Roman! Why don't we take a closer look at the route declaration to see how it works?

Routes are always declared using the `Route` class. That's the bit right at the start and before the `::`. The `get` part is a method on the route class that is used to 'catch' requests that are made using the HTTP verb 'GET' to a certain URL.

You see, all requests made by a web browser contain a verb. Most of the time, the verb will be `GET`, which is used to request a web page. A `GET` request will get sent every time you type a new web address into your web browser.

It's not the only request, though. There is also `POST`, which is used to make a request and supply a little bit of data with it. These are normally the result of submitting a form, where data must be sent to the web server without displaying it in the URL.

There are other HTTP verbs available as well. Here are some of the methods that the routing class has available to you:

Example 03: Routing methods.

```
<?php

// app/Http/routes.php

Route::get();
Route::post();
Route::put();
Route::patch();
Route::delete();
Route::any();
```

All of these methods will accept the same parameters, so feel free to use whatever HTTP verb is correct in the given situation. This is known as `RESTful routing`. We will go into more detail on this topic later. For now, all you need to know is that `GET` is used to make requests, and that `POST` is used when you need to send additional data with the request.

The `Route::any()` method is used to match any HTTP verb. However, I would recommend using the correct verb for the correct situation to make our application code more transparent.

Let's get back to the example at hand. Here it is again to refresh your memory:

Example 04: Our first route, again.

```
<?php

// app/Http/routes.php

Route::get('my/page', function () {
    return 'Hello world!';
});
```

The next portion of the snippet is the first parameter to the `get()` (or any other verb) method. This parameter defines the URI that you wish the URL to match. In this case, we are matching the URI `my/page`.

The final parameter is used in this instance to provide the application logic to handle the request. Here we are using a `Closure`, which is also known as an `anonymous function`. Closures are simply functions without a name that can be assigned to variables, as with other simple types.

For example, the snippet above could also be written as:

Example 05: Separate closure.

```
<?php

// app/Http/routes.php

$logic = function () {
    return 'Hello world!';
};

Route::get('my/page', $logic);
```

Here we are storing the `Closure` within the `$logic` variable, and later passing it to the `Route::get()` method.

In this instance, Laravel will execute the Closure only when the current request is using the HTTP verb `GET` and matches the URI `my/page`. Under these conditions, the return statement will be processed and the "Hello world!" string will be handed to the browser.

You can define as many routes as you like, for example:

Example 06: Multiple routes.

```
<?php

// app/Http/routes.php

Route::get('first/page', function () {
    return 'First!';
});

Route::get('second/page', function () {
    return 'Second!';
});

Route::get('third/page', function () {
    return 'Potato!';
});
```

Try navigating to following URLs to see how our application behaves.

Example 07: Multiple URLs.

```
http://homestead.app/first/page
http://homestead.app/second/page
http://homestead.app/third/page
```

You will likely want to map the root of your web application. For example.

Example 08: No path.

```
http://homestead.app
```

Normally, this would be used to house your application's home page. Let's create a route to match this.

Example 09: Route with no path.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return 'In soviet Russia, function defines you.';
});
```

> Hey, wait a minute! We don't have a forward slash in our URI?!

Well spotted! You won't need to worry about that, though.

You see, a route containing only a forward slash will match the URL of the website, whether it has a trailing backslash or not. The route above will respond to either of the following URLs.

Example 10: With or without a slash.

```
http://homestead.app
http://homestead.app/
```

URLs can have as many segments (the parts between the slashes) as you like. You can use this to build a site hierarchy.

Consider the following site structure:

Example 11: Imaginary website.

```
/
/books
    /fiction
    /science
    /romance
/magazines
    /celebrity
    /technology
```

It's a fairly minimal site, but a great example of structure you will often find on the web. Let's recreate this using Laravel routes.

For clarity, the handling Closure of each route has been truncated.

Example 12: Route our imaginary site.

```
<?php

// app/Http/routes.php

// home page
Route::get('/', function () {});


// routes for the books section
Route::get('/books', function () {});
Route::get('/books/fiction', function () {});
Route::get('/books/science', function () {});
Route::get('/books/romance', function () {});

// routes for the magazines section
Route::get('/magazines', function () {});
Route::get('/magazines/celebrity', function () {});
Route::get('/magazines/technology', function () {});
```

With this collection of routes, we have easily created a site hierarchy. However, you may have noticed a certain amount of repetition. Let's find a way to minimize this repetition, and thus, adhere to DRY (Don't Repeat Yourself) principles.

Route Parameters

Route parameters can be used to insert placeholders into your route definition. This will effectively create a pattern in which URI segments can be collected and passed to the application's logic handler.

This might sound a little confusing, but when you see it in action, everything will fall into place. Here we go...

Example 13: Route parameters.

```
<?php

// app/Http/routes.php

// routes for the books section
Route::get('/books', function () {
    return 'Books index.';
});

Route::get('/books/{genre}', function ($genre) {
    return "Books in the {$genre} category.";
});
```

In this example, we have eliminated the need for all of the book genre routes by including a route placeholder. The `{genre}` placeholder will map anything that is provided after the `/books/` URI. This will pass its value into the Closure's `$genre` parameter, which will allow us to make use of this information within our logic portion.

For example, if you were to visit the following URL:

Example 14: Parameter in the URL.

```
http://homestead.app/books/crime
```

You would be greeted with this text response:

Example 15: Output.

```
Books in the crime category.
```

We could also remove the requirement for the book's index route by using an optional parameter. A parameter can be made optional by adding a question mark (?) to the end of its name. For example:

Example 16: Optional parameter.

```
<?php

// app/Http/routes.php

// routes for the books section
Route::get('/books/{genre?}', function ($genre = null) {
    if ($genre == null) {
         return 'Books index.';
    }
    return "Books in the {$genre} category.";
});
```

If a genre isn't provided with the URL, then the value of the `$genre` variable will be equal to `null`, and the message `Books index.` will be displayed.

If we don't want the value of a route parameter to be `null` by default, we can specify an alternative using assignment. For example:

Example 17: Default parameter values.

```
<?php

// app/Http/routes.php

// routes for the books section
Route::get('/books/{genre?}', function ($genre = 'Crime') {
    return "Books in the {$genre} category.";
});
```

If we visit the following URL:

Example 18: URL with missing parameter.

```
http://homestead.app/books
```

We will receive this response:

Example 19: Output.

```
Books in the Crime category.
```

Hopefully, you are starting to see how routes are used to direct requests to your site, and as the 'code glue' that is used to tie your application together.

There's a lot more to routing. Before we come back to that, let's cover more of the basics. In the next chapter, we will look at the types of responses that Laravel has to offer.

10. Responses

When someone asks you a question, unless you are in a mood or the question doesn't make sense, you will most likely give them a response. I guess other exceptions are those question-like greetings like when someone says, 'Alright?'. Even then, you should at least give them the nod in return. Some form of response.

Requests made to a web server are no different to the guy that said 'Alright?'. They will hope to get something back. In a previous chapter, our responses took the shape of strings returned from routed closures.

Something like this:

Example 01: A string response.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return 'Yeh am alright guv.';
});
```

Here we have the string "Yeh am alright guv." sent back to the browser. The string is our response and is always `return`ed by a routed closure or a Controller action (which we will cover later).

It would be fair to assume that we would like to send some HTML as our response. This is often the case when developing web applications.

I guess we could enclose the HTML in the response string?

Example 02: String HTML response.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return '<!doctype html>
            <html lang="en">
                <head>
```

```
                    <meta charset="UTF-8">
                    <title>Alright!</title>
                </head>
                <body>
                    <p>This is the perfect response!</p>
                </body>
            </html>';
});
```

Awesome! Now you see the power and grace of Laravel... just kidding. We don't want to serve HTML this way. Embedding logic would get annoying and, more importantly, it's just plain wrong!

Luckily for us, Laravel has some `Response` objects that make returning a meaningful reply a whole lot easier. Let's check out the `View` response since that's the most exciting one!

Views

Views are the visual part of your application. Within the `Model View Controller` pattern, they make up the `View` portion. That's why they are called views. It's not rocket science. Rocket science will be covered in a later chapter.

A view is just a plain text template that can be returned to the browser, though it's likely that your views will contain HTML. Views use the `.php` extension and are normally located within the `resources/views` directory. This means that PHP code can also be parsed within your views.

Let's just create a very simple view for now.

Example 03: Our first view.

```
<!-- resources/views/simple.php -->

<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Views!</title>
</head>
<body>
    <p>Oh yeah! VIEWS!</p>
</body>
</html>
```

Great! A tiny bit of HTML stored in `resources/views/simple.php`. Now let's make a view.

> Didn't we just do that?

Haha! Yes, you did little one, but I didn't mean 'Let's make another view file.' Instead, let's `make()` a new view response object based upon the view file we just created. The files representing views can be called templates. This may help you distinguish between the `View` objects and the HTML templates.

Example 04: A view response.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('simple');
});
```

> Oh, I see! You were talking about the `make()` method.

Indeed, I was little buddy! Using the `View::make()` method we can create a new instance of a `View` response object. The first parameter we hand to the `make()` method is the view template that we wish to use. You will notice that we didn't use the full path `resources/views/simple.php`. This is because Laravel is clever. It will by default assume that your views are located in `resources/views` and will look for a file with an appropriate view extension.

You can use the global function `view()` instead of `View::make()` to simplify your applications.

If you look a little closer at the `Closure`, you will see that the `View` object we have created is being returned. This is very important since Laravel will serve the result of our Closure to the web browser.

Go ahead and try hitting the `/` URI for your application. Great, that's the template we wrote!

Later in the book, you will learn how to make different types of templates that work with the `View` response to make our lives easier. For now, we will stick to the basics to get a good grasp on the fundamental Laravel concepts.

View Data

Being able to show templates is awesome. It really is. What if we want to use some data from our Closure, though? In an earlier chapter, we learned how to use Route parameters. Maybe we want to be able to refer to these parameters in the View? Let's take a look at how this can be done.

Example 05: View data.

```
<?php

// app/Http/routes.php

Route::get('/{squirrel}', function ($squirrel) {
    $data['squirrel'] = $squirrel;

    return View::make('simple', $data);
});
```

In the above route, we take the `$squirrel` parameter and add it to our view data array. You see, the second parameter of the `make()` method accepts an array that is passed to the view template. I normally call my array `$data` to indicate that it is my view **data** array, but you may use any name you like!

Be sure to check out the `compact()` PHP native function. It's useful for easily creating small view data arrays.

Before we start using this data, let's talk a little more about the view data array. When the array is passed to the view, the array keys are 'extracted' into variables that have the array key as their name and the given value. It's a little confusing to explain without an example so let me simplify it for you.

Here we have an array to be handed to `View::make()` as view data.

Example 06: Our view data array.

```
<?php
['name' => 'Taylor Otwell', 'status' => 'Code Guru']
```

In the resulting view template we can access these values by key:

Example 07: Using data in a view.

```
<?php echo $name;           // gives 'Taylor Otwell' ?>

<?php echo $status;         // gives 'Code Guru' ?>
```

So our `name` array key becomes the `$name` variable within the template. You can store multi-dimensional arrays as deep as you like in your view data array. Feel free to experiment!

Let's use the `$squirrel` variable in the simple template we created earlier.

Example 08: View data nested in HTML.

```
<!-- resources/views/simple.php -->

<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Squirrels</title>
</head>
<body>
    <p>I wish I were a <?php echo $squirrel; ?> squirrel!</p>
</body>
</html>
```

Now if we visit the URI `/gray` we receive a page stating 'I wish I were a gray squirrel!'.

Well that was simple wasn't it? Using views, you will no longer have to return strings from your Closures!

Earlier, I mentioned that there are different types of response objects. In some circumstances, you may wish to redirect the flow of your application to another route or logic portion. In such a circumstance the `Redirect` response object will be useful. See, Laravel's got your back!

Redirects

A `Redirect` is a special type of response object which redirects the flow of the application to another route. Let's create a couple of sample routes so that I can explain in more detail.

Example 09: Two string responses.

```
<?php

// app/Http/routes.php

Route::get('first', function () {
    return 'First route.';
});

Route::get('second', function () {
    return 'Second route.';
});
```

Having added the above routes, you will receive 'First route.' upon visiting the `/first` URI and 'Second route.' upon visiting the `/second` URI.

Excellent, that's exactly what we expected to happen. Now let's completely shift the flow of the application by adding a redirect to the first routed closure.

Example 10: A redirect response.

```
<?php

// app/Http/routes.php

Route::get('first', function () {
    return Redirect::to('second');
});

Route::get('second', function () {
    return 'Second route.';
});
```

In the first route we are now returning the result of the `Redirect::to()` method and passing the URI of the target location. In this case, we are passing the URI for the second route `second` as the location.

If you'd like, you can use the `redirect()` global function instead of `Redirect::to()`.

If you now visit the `/first` URI, you will be greeted with the text 'Second route.'. This is because upon receiving the returned Redirect object, Laravel has shifted the flow

of our application to the target destination. In this case, the flow has been shifted to the closure of the second route.

This can be useful when a condition of some kind has failed, and you need to redirect the user to a more useful location. Here's an example using the authentication system (which we will cover later) that will provide another use case.

Example 11: Redirect after failed authentication.

```
<?php

// app/Http/routes.php

Route::get('books', function () {
    if (Auth::guest()) {
        return Redirect::to('login');
    }
    // Show books to only logged in users.
});
```

In this example, if a user who has not yet logged into the system visits the `/books` URI then they are considered a guest and would be redirected to the login page.

Later you will find a better way to limit access when we discuss route middleware, so don't read too much into the above example. Instead, just consider that we could redirect the user to a more sensible destination if our conditions are not met.

Custom Responses

Both `View` and `Redirect` produce instances of Laravel `Response` object. The response object is an instance of a class that can be handed back to Laravel as the result of a routed closure or a controller action to enable the framework to serve the right kind of response to the browser.

Response objects contain a body, a status code, HTTP headers, and other useful information. For example, the body segment of the View would be its HTML content. The status code for a Redirect would be a `301`. Laravel uses this information to construct a sensible result that can be returned to the browser.

We aren't merely limited to using `View` and `Redirect`. We could also create our own response object to suit our needs. Let's have a look at how this can be done.

Example 12: Create a response with status code.

```
<?php

// app/Http/routes.php

Route::get('custom/response', function () {
    return Response::make('Hello world!', 200);
});
```

In the above example, we use the `Response::make()` method to create a new response object. The first parameter is the content or body of the response and the second parameter is the HTTP status code that will be served with the response.

Once again, you can use the `response()` global function instead of `Response::make()`. Decide which you think looks the best!

If you haven't seen HTTP status codes before, then think of them as status notifications for the web browser receiving your page. For example, if all goes well, a standard response may contain a `200` status code, which is web-server speak for 'A-OK.' A `302` status code indicates that a redirect has been performed.

In fact, I bet you have already come across the now infamous `404` not found page. The 404 part is the status code received by the browser when a requested resource could not be found.

Simply put, the above response will serve the content 'Hello world!' with a HTTP status code of `200` to let the browser know that its request was a success.

HTTP headers are a collection of key-value pairs of data which represent useful information to the web browser that is receiving the response. Normally they are used to indicate the format of the result or how long a piece of content should be cached for. However, we can define custom headers as we please. Let's have a look at how we can set some header values.

Example 13: Change headers on a response.

```
<?php

// app/Http/routes.php

Route::get('custom/response', function () {
    $response = Response::make('Hello world!', 200);
    $response->headers->set('our key', 'our value');
    return $response;
});
```

We have created a sample response object just as we did in the previous example. However, this time, we have also set a custom header.

The collection of HTTP headers can be accessed as the `headers` property of the response object.

Example 14: View headers.

```
<?php

var_dump($response->headers);
```

Using the `set()` method on this collection we can add our own header to the collection by providing a key as a first parameter and the associated value as the second.

Once our header has been added we simply return the response object as we have done previously. The browser will receive the headers along with the response and can use this information however it wishes.

Let's think of a more useful example. Hmm. Let's imagine that we want our application to serve markdown responses instead of HTML. We know that a web browser would not be able to parse a markdown response, but we might have another desktop application that could.

To indicate that the content is markdown and not HTML we will modify the `Content-Type` header. The `Content-Type` is a common header key used by browsers to distinguish between the various formats of content that are sent to them. Don't be confused! Let's have an example.

Example 15: Return a markdown response.

```
<?php

// app/Http/routes.php

Route::get('markdown/response', function () {
    $response = Response::make('***some bold text***', 200);
    $response->headers->set('Content-Type', 'text/x-markdown');
    return $response;
});
```

Having set the body of the response object to some sample markdown, and the `Content-Type` header to the mime type for the Markdown plain text format, we have served a response that can be identified as Markdown.

Our desktop application can now make a request to the `/markdown/response` URI, examine the `Content-Type` header, and in receiving the `text/x-markdown` value it will know to use a markdown transformer to handle the body.

Because we are all friends here, I'm going to share a secret with you. Come closer. Get in here. Let's have a huddle. The response object doesn't belong to Laravel.

> TREACHERY? WHAT MADNESS IS THIS?

Now don't get too worked up. You see, to avoid a lot of 're-inventing the wheel,' Laravel has used some of the more robust components that belong to the Symfony 2 project. The Laravel response object inherits most of its content from the `Response` object belonging to the 'Symfony HTTP Foundation' component.

What this means is that if we take a look at the API for the Symfony response object, suddenly we have access to a whole heap of extra methods that aren't covered in the Laravel docs! Holy smokes! Now that I have given away this secret, there's nothing to stop you from becoming a Laravel master!

The API documentation for the Symfony Response object can be found here[1]. If you look at the page, you will notice that the class has an attribute called `$headers`. That's right! That's the collection of headers we were using only a minute ago.

Since the Laravel response object inherits from this one, feel free to use any of the methods you see in the Symfony API documentation. For example, let's have a look at the `setTtl()` method. What does the API say?

[1] http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html"TheSymfonyResponseObject"

---

public Response setTtl(**int $seconds**)

Sets the response's time-to-live for shared caches. This method adjusts the Cache-Control/s-maxage directive.

Parameters:

int $seconds Number of seconds

Return Value:

Response

Right, so this method sets the time-to-live value for shared caches. I'm no expert on this kind of thing, but a time to live suggests how long a piece of information is considered useful before it is discarded. In this instance, the TTL relates to the content caching.

Let's give it a value for funsies. Having looked at the method signature, we see that it accepts an integer representing the time-to-live value in seconds. Let's give this response 60 seconds to live. Like some cruel James Bond villain.

Example 16: Set response time to live.

```
<?php

// app/Http/routes.php

Route::get('our/response', function () {
    $response = Response::make('Bond, James Bond.', 200);
    $response->setTtl(60);
    return $response;
});
```

Now when our response is served, it will have a time to live value of 60 seconds. You see, by interrogating the underlying Symfony component, we have a wealth of advanced options that we can use to modify our application responses to suit our needs.

Don't feel overwhelmed by the amount of complexity contained within the base Response object. For the most part, you will be happy using Laravel's View and Response classes to serve simple HTTP responses. The above example simply serves as a good starting point for advanced users looking to tweak their applications for specific scenarios.

JSON Responses

Often within our application, we will have some data that we wish to serve as JSON. It could be a simple object or an array of values.

Laravel provides a `Response::json()` method that will configure the response object with some details that are specific to JSON results. For example an appropriate HTTP Content-Type header.

You can optionally use the `response()->json()` function chain instead of `Response::json()`.

We could set these up manually, but why bother when Laravel will do it for us? Let's have a look at this method in action.

Example 17: A JSON response.

```
1  <?php
2
3  // app/Http/routes.php
4
5  Route::get('markdown/response', function () {
6      $data = ['iron', 'man', 'rocks'];
7      return Response::json($data);
8  });
```

By handing an array to the `Response::json()` method it has been converted to a JSON string and set as the body of our new `Response` object. Appropriate headers have been set to explain that the provided data is, in fact, a JSON string. The web browser will receive the following body:

Example 18: JSON array.

```
1  ["iron","man","rocks"]
```

Which is both compact and true. Enjoy using this shortcut to build clean yet functional JSON APIs!

Download Responses

Serving download streams for files directly requires certain headers to be set. Fortunately, Laravel takes care of this for you using the `Response::download()` shortcut. Let's see this in action.

Example 19: A download response.

```
<?php

// app/Http/routes.php

Route::get('file/download', function () {
    $file = 'path_to_my_file.pdf';
    return Response::download($file);
});
```

If we navigate to the `/file/download` URI the browser will initiate a download instead of displaying a response. The `Response::download()` method will receive a path to a file which will be served when the response is returned.

You can also provide optional second and third parameters to configure a custom file name and an array of headers. For example:

Example 20: Download a file with status and headers.

```
<?php

// app/Http/routes.php

Route::get('file/download', function () {
    $file = 'path_to_my_file.pdf';
    return Response::download($file, 'my_file.pdf', ['iron', 'man']);
});
```

Here we will serve our file with the name 'my_file.pdf' and a header value of `iron=man`.

This chapter was a lot longer than I originally anticipated, but I'm sure you will see that returning appropriate response objects can be a lot more valuable than returning simple strings.

11. Blade Templates

In this chapter, we will learn how to master the Blade. You will need it. To claim your rightful place as a PHP artisan, you will need to challenge and defeat High Lord Otwell in armed combat.

It's a rite of passage. Only then will you be able to take your rightful place amongst the Laravel council, and earn a spot at the Grand drinking table.

Once a month, us Council members ride out to the PHP battleground sat atop our fearsome Laravel riding pandas to do battle with the developers of other frameworks. If you want to ride into battle alongside us and fight for the honor of Laravel, then you **must** learn to master the Blade.

Well, *must* is a strong word I suppose. I mean, you could also master Blade templating. It's not quite as extravagant, and there are no fierce riding pandas involved. It is pretty handy, though, and perhaps more suited to coder types than hardcore battle?

Yeah, that's settled then, let's have a look at Blade templating.

You might be wondering about the name 'Blade'? So was I as I wrote this chapter, so I decided to ask Taylor. It turns out that the .NET web development platform has a templating tool called 'Razor,' from which much of the Blade syntax was derived. Razor..blade.. razorblade. That's it. Nothing funny there really, sorry. :(

Actually, forget what I just told you, let's reinvent that story. Just between us. The templating language was named after Taylor's alter ego 'Blade' back during his days of Vampire hunting. That's the real story.

Right, let's make a template.

Building Templates

I know, I know. I have already taught you how to create views, right? They are useful for separating the visual aspect of your application from its logic. However, that certainly doesn't mean that they can't be improved upon.

The problem with standard PHP templates is that we have to insert those ugly PHP tags within them to use the data that our logic portions have provided. They look a little out of place within our nice HTML templates. They spoil them! It makes me so angry. I could. I could. No, let me be for a moment.

> Erm...

It's ok; my rage has settled now. Let's create a Blade template to get that nasty PHP mess out of the way. To get started, we will need to make a new file. Blade templates live in the same location as our standard view files. The only difference is that they use the `.blade.php` extension rather than just `.php`.

Let's create a simple template.

Example 01: A simple template.

```
<!-- resources/views/example.blade.php -->

<h1>Dear Lord Otwell</h1>
<p>I hereby challenge you to a duel for the honour of Laravel.</p>

<?php echo $squirrel; ?>
```

Here we have our blade template. It looks fairly similar to what we have seen already, right? That's because Blade first parses the file as PHP. You see our `$squirrel`? Every view file must have a squirrel. OK, that's not true, but it does show that PHP can be parsed just as before.

We can demonstrate this using the same syntax as we would for a normal view. You might have assumed that it would require passing `example.blade` to the `view()` method, but that would be incorrect.

Example 02: Render a blade view.

```
<?php

// app/Http/routes.php

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

See? Laravel knows what a Blade template is, and how to look for one. For this reason, the `view()` statement hasn't changed at all. How convenient!

Blade does have some tricks of its own, however. Let's take a look at the first one.

Processing PHP

Many of our templates will involve echoing out data provided by the logic portion of your application. Typically, it would look something like this:

Example 03: Output data in a template.

```
<!-- resources/views/example.blade.php -->

<p><?php echo $taylorTheVampireSlayer; ?></p>
```

It's not exactly verbose, but it could be improved upon. Let's see how we can echo values using Blade templates.

Example 04: Echo values using blade.

```
<!-- resources/views/example.blade.php -->

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

Everything surrounded by `{{` double curly brackets `}}` is transformed into an echo by Blade when the template is processed. This is a much cleaner syntax, and a whole lot easier to type.

Since the Blade template directives are translated directly to PHP, we can use any PHP code within these brackets, including methods.

Example 05: Echo PHP code.

```
<!-- resources/views/example.blade.php -->

<p>{{ date('d/m/y') }}</p>
```

You don't even need to provide the closing semi-colon. Laravel does that for you.

Sometimes, you will want to protect yourself by escaping a value that you output. You might be familiar with using methods such as `strip_tags()` and `htmlentities()` for this purpose. Why? Well, consider the output of this template.

Example 06: Attempt to embed JavaScript.

```
<!-- resources/views/example.blade.php -->

<p>{{ '<script>alert("CHUNKY BACON!");</script>' }}</p>
```

What a nasty piece of Code! It will attempt to cause some Javascript to be injected into the page and a browser popup to be displayed containing the text 'CHUNKY BACON!'. Bad _why! Nasty Ruby developers are always trying to break our websites.

Fortunately, Laravel will automatically escape those HTML tags and convert them to HTML entities so that the script can't execute. For example, the output will look like this:

Example 07: Output.

```
<!-- resources/views/example.blade.php -->

<p>&lt;script&gt;alert(&quot;CHUNKY BACON!&quot;);&lt;/script&gt;</p>
```

What if we'd like out HTML to be parsed in the template? Don't worry! We can use some alternate tags to disable the auto-escaping.

Let's see this in action.

Example 08: Do not escape HTML.

```
<!-- resources/views/example.blade.php -->

<p>{!! '<script>alert("CHUNKY BACON!");</script>' !!}</p>
```

We've replaced our double curly braces with a single curly brace and two exclamation marks. Let's view the source of the page to see how it looks.

Example 09: Output.

```
<!-- resources/views/example.blade.php -->

<p><script>alert("CHUNKY BACON!");</script></p>
```

As you can see, the HTML tags and some other characters have been inserted without modification. Be careful with this method of embedding data; users could abuse this method to inject their own javascript!

Moving on!

Control Structures

PHP has some control structures. If statements, while, for and foreach loops. If you haven't heard of these before, then this book isn't the one for you! Maybe you should check out my PHP Pandas book first?

You will most likely be familiar with using the alternative syntax for control structures using colons `:` within your PHP templates. For example, your `if` statements will look like this:

Example 10: PHP alternate control structures.

```
<!-- resources/views/example.blade.php -->

<? if ($something) : ?>
    <p>Something is true!</p>
<? else : ?>
    <p>Something is false!</p>
<? endif; ?>
```

Once again, these do the trick, but they aren't very fun to type. They will slow you down, but fortunately for you, Blade will come to your rescue!

Here's how the above snippet will look within a Blade template.

Example 11: A blade control structure.

```
<!-- resources/views/example.blade.php -->

@if ($something)
    <p>Something is true!</p>
@else
    <p>Something is false!</p>
@endif
```

That's much cleaner, right? Let's take a look at what we have stripped out. For a start, the PHP opening `<?` and closing `?>` tags are gone. Those are probably the most complicated to type.

We can also strip out the colons `:` and semi `;` colons. We don't need those wasting space in our templates!

Lastly, we have made an addition to the usual syntax. We have prefixed our control statement lines with an at `@` symbol. In fact, all blade control structures and helper methods are prefixed with this symbol so that the template compiler knows how to handle them.

Let's add an `elseif` to the mix for a further example.

Example 12: Control structure with elseif.

```
<!-- resources/views/example.blade.php -->

@if ($something == 'Red Panda')
    <p>Something is red, white, and brown!</p>
@elseif ($something == 'Giant Panda')
    <p>Something is black and white!</p>
@else
    <p>Something could be a squirrel.</p>
@endif
```

We've added another statement into the mix, following the same rules of removing PHP tags, colons, and semi-colons, and adding the `@` sign. Everything works perfectly.

Here's a challenge. Try to picture a `foreach` PHP loop represented with blade syntax. Close your eyes, picture it. Focus, focus!

Did it look like this?

Example 13: Imaginary foreach

```
<!-- resources/views/example.blade.php -->

      c~~p ,---------.
 ,---'oo  )           \
( O O                  )/
 `=^='                 /
       \    ,     .   /
       \\  |-----'|  /
       ||__|    |_|__|
```

No? Good, because that is a hippo. However, if it looked a little like the following snippet, then you will get a cookie.

Example 14: The foreach loop.

```
<!-- resources/views/example.blade.php -->

@foreach ($manyThings as $thing)
    <p>{{ $thing }}</p>
@endforeach
```

Enjoy your cookie! As you can see, we have used the Blade `{{` echo `}}` syntax to output the loop value. A for loop looks just as you might imagine. Here is an example for reference.

Example 15: The for loop.

```
<!-- resources/views/example.blade.php -->

@for ($i = 0; $i < 999; $i++)
    <p>Even {{ $i }} red pandas, aren't enough!</p>
@endfor
```

It's very simple, and exactly as you might have expected. The while loop follows the same rules, but I'm going to show a quick example to allow this to be a useful reference chapter.

Example 16: The while loop.

```
<!-- resources/views/example.blade.php -->

@while (isPretty($keiraKnightley))
    <p>This loop probably won't ever end.</p>
@endwhile
```

Right, so you are the conditional master now. Nothing can phase you, right buddy? Not even PHP's 'unless' condition.

> Err Dayle, Ruby has that. I don't know if PHP ha..

OK. You've caught me out. PHP doesn't have an 'unless' condition. However, Blade has provided a helper to allow it. Let's take a look at an example.

Example 17: The unless condition.

```
<!-- resources/views/example.blade.php -->

@unless (worldIsEnding())
    <p>Keep smiling.</p>
@endunless
```

Unless is the exact opposite of an if statement. An if statement checks if a condition equates to a true value, and then executes some logic. However, the unless statement will execute its logic only if the condition equates to false. You can think of it as a control structure for pessimists.

Template Inclusion

Blade includes a few other helper methods to make your templates easier to construct and manage. It won't write your views for you, however, maybe we could add that to the task list for Laravel 6?

- php artisan project:complete
- Include strpos+1() function.
- Have views write themselves.

There we go. Until those features are in place, we will have to write our templates. That doesn't mean we have to put everything in one file, though.

With PHP, you can `include()` a file into the current file, executing its contents. You could do that with views to break them apart into separate files for organizational purposes. Laravel helps us achieve this goal by providing the `@include()` Blade helper method to import one view into another, parsing its contents as a Blade template if required. Let's take a look at an example of this in action.

Here's a `header.blade.php` file containing the header for our page, and possibly even other pages. We'll call this a partial view since it's not meant to be used alone.

Example 18: A header partial view.

```
<!-- resources/views/header.blade.php -->

<h1>When does the Narwhal bacon?</h1>
```

Here's the associated footer partial.

Example 19: A footer partial view.

```
<!-- resources/views/footer.blade.php -->

<small>Information provided based on research as of 3rd March '16.</small>
```

Next, we have our primary template. The one that is being displayed by our routed Closure or Controller action.

Example 20: Include partials from primary template.

```
<!-- resources/views/example.blade.php -->

<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Narwhals</title>
</head>
<body>
    @include('header')
    <p>Why, the Narhwal surely bacons at midnight, my good sir!</p>
    @include('footer')
</body>
</html>
```

As you can see, the helper methods within the `example.blade.php` template are pulling in the contents of our header and footer templates using the `@include()` helper. Include takes the name of the view as a parameter, in the same short format as the `view()` method that we used earlier.

Let's take a look at the resulting document.

Example 21: Output.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Narwhals</title>
</head>
<body>
    <h1>When does the Narwhal bacon?</h1>
```

```
    <p>Why, the Narhwal surely bacons at midnight, my good sir!</p>
    <small>Information provided based on research as of 3rd May '13.</small>
</body>
</html>
```

Our included templates have been... well, included into the page. This will make our header and footer templates reusable and DRY. We can include them into other pages to save repeating content, and to make that content editable in a single location. There is a better way of doing this, though, so keep reading!

Template Inheritance

Laravel's Blade provides a way to build templates that can benefit from inheritance. Many people find this confusing, yet it's a really neat feature. I'm going to try and simplify it as best as I can and hopefully you will soon find the art of creating templates to be a pleasurable experience.

First of all, let's think about templates. Some parts of a web page don't change across each page. These are the tags that need to be present for any web page we view. We can call it our boilerplate code, or even 'base' template if you like. Here's an example:

Example 22: A base template.

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

We're going to use this layout for all of our pages. Why don't we tell Laravel? Let's say that it's a Blade layout. To do that, we just need to define some areas that content can be inserted into. In Laravel, we call these areas 'sections.' This is how we define them:

Example 23: Define sections.

```
<!-- resources/views/layouts/base.blade.php -->

<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
    @section('head')
        <link rel="stylesheet" href="style.css" />
    @show
</head>
<body>
    @yield('body')
</body>
</html>
```

Here we have created a template with two sections. Let's take a look at the one within the body first. That's the easy one. It looks like this:

Example 24: Yield for content.

```
@yield('body')
```

This statement will tell blade to create a section that we can fill the content in later. We'll give it the nickname 'body' so we can refer to it later.

The other section looks like this:

Example 25: Define a section and show content.

```
@section('head')
    <link rel="stylesheet" href="style.css" />
@show
```

This one is very similar to the 'yield' section, except that you can provide some default content. In the above example, the content between the `@section` and `@show` tags will be shown unless a child template chooses to override it.

What do I mean by a child template? Well as always, let's jump right in with an example.

Example 26: Extend a base template and inject content.

```
<!-- resources/views/home.blade.php -->

@extends('layouts.base')

@section('body')
    <h1>Hurray!</h1>
    <p>We have a template!</p>
@stop
```

Right, let's walk through this. First, we have the 'extends' blade function:

Example 27: The extends function.

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

This tells Blade, which layout we will be using to render our content. The name that we pass to the function should look like those that you pass to `view()`, so in this situation, we are referring to the 'base.blade.php' file in the 'layouts' directory within 'resources/views'. Remember that a period (`.`) character represents a directory separator when dealing with views.

In Laravel 3 this function was called '@layout()', but it was renamed in Laravel 4 to bring it more inline with other templating engines such as Symfony's twig. Beware, Laravel 3 devs!

Now that we know which layout we are using, it's time to fill in the gaps. We can use the `@section` blade function to inject content into sections within the parent template. It looks like this:

Example 28: Inject content into body.

```
@section('body')
    <h1>Hurray!</h1>
    <p>We have a template!</p>
@stop
```

We pass the '@section' function the nickname that we gave our section within the parent template. Remember? We called it 'body.' Everything that is contained between '@section' and '@stop' will be injected into the parent template, where the '@yield('body')' is.

Let's create a route to see this in action. To render the template, we need only add a `view()` response to render the child template. Like this:

Example 29: Render child template.

```php
<?php

// app/Http/routes.php

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

Now if we visit `/` and view the source, we will see that the page looks like this:

Example 30: Output.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
    <link rel="stylesheet" href="style.css" />
</head>
<body>
    <h1>Hurray!</h1>
    <p>We have a template!</p>
</body>
</html>
```

Okay, so the formatting might be a little different, but the content should be the same. Our section has been injected into our parent template. Since we didn't override the contents of the 'head' section, the default value has been inserted.

As you see, we could have as many child templates as we want to inherit from this parent template. This saves us the effort of having to repeat the boilerplate code.

Let's change the child template a little to provide some content for the 'head' section. Like this:

Example 31: Override default content of section.

```
<!-- resources/views/home.blade.php -->

@extends('layouts.base')

@section('head')
    <link rel="stylesheet" href="another.css" />
@stop

@section('body')
    <h1>Hurray!</h1>
    <p>We have a template!</p>
@stop
```

As you might expect, the head section has been injected with our additional CSS file, and the source for our page now looks like this:

Example 32: Output.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
    <link rel="stylesheet" href="another.css" />
</head>
<body>
    <h1>Hurray!</h1>
    <p>We have a template!</p>
</body>
</html>
```

Do you remember how the head section had some default content between the '@section' and '@show'? Well, we might wish to append to this content, rather than replace it. To do this, we can use the `@parent` helper. Let's modify our child template to use it, like this:

Example 33: Inject content of parent section.

```
<!-- resources/views/home.blade.php -->

@extends('layouts.base')

@section('head')
    @parent
    <link rel="stylesheet" href="another.css" />
@stop

@section('body')
    <h1>Hurray!</h1>
    <p>We have a template!</p>
@stop
```

The '@parent' helper tells Blade to replace the parent marker with the default content found within the parent's section. That sentence might sound a little confusing, but it's quite simple. Let's take a look at how the source has changed for some clarity.

Example 34: Output.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
    <link rel="stylesheet" href="style.css" />
    <link rel="stylesheet" href="another.css" />
</head>
<body>
    <h1>Hurray!</h1>
    <p>We have a template!</p>
</body>
</html>
```

See? Our '@parent' marker was replaced with the default content from the parent's section. You can use this method to append new menu entries or extra asset files.

You can have as many chains of inheritance within blade templates as you like, the following example is perfectly fine.

Example 35: First view.

```
<!-- resources/views/first.blade.php -->

<p>First</p>
@yield('message')
@yield('final')
```

Example 36: Second view.

```
<!-- resources/views/second.blade.php -->
@extends('first')

@section('message')
    <p>Second</p>
    @yield('message')
@stop
```

Example 37: Third view.

```
<!-- resources/views/third.blade.php -->
@extends('second')

@section('message')
    @parent
    <p>Third</p>
    @yield('message')
@stop
```

Example 38: Fourth view.

```
<!-- resources/views/fourth.blade.php -->
@extends('third')

@section('message')
    @parent
    <p>Fourth</p>
@stop

@section('final')
    <p>Fifth</p>
@stop
```

Woah crazy right! Try to follow the inheritance chain to see how the output is constructed. It might be best to work from the child templates upwards to each parent. If we were to render the 'fourth' view, this would be the outputted source.

Example 39: Output.

```
<p>First</p>
<p>Second</p>
<p>Third</p>
<p>Fourth</p>
<p>Fifth</p>
```

To put it simply:

Fourth extends Third which extends Second which extends First, which is the base template.

You may also have noticed that the 'final' section of the base template had content provided by the fourth template file. This means you can provide content for a section from any 'layer.' As you can see, Blade is *very* flexible.

Comments

As you've likely discovered, HTML has its method of including comments. They look like this.

Example 40: A HTML comment.

```
<!-- This is a lovely HTML comment. -->
```

You are right, comment. You are lovely, but unfortunately, you also get outputted along with the rest of the page source. We don't want people reading the information that is meant for our developers.

Unlike HTML comments, PHP comments are stripped out when the page is pre-processed. This means that they won't show up when you try to view source. We could include PHP comments in our view files like this:

Example 41: A PHP comment.

```
<?php // This is a secret PHP comment. ?>
```

Sure, now our content is hidden. It's a bit ugly, though, right? No room for ugliness in our utopian Blade templates. Let's use a Blade comment instead; they compile directly to PHP comments.

Example 42: A Blade comment.

```
{{-- This is a pretty, and secret Blade comment. --}}
```

Use blade comments when you want to put notes in your views that only the developers will see.

Javascript Support

What if we want to use one of those pesky Javascript frameworks that also use curly braces for output? We don't want them to be replaced by blade, do we? Don't panic! If you prefix your opening curly braces with an `@` symbol then Blade will escape them.

Here's an example:

Example 43: Escape blade echo.

```
<script>@{{ javascriptValue }}</script>
```

12. Request Data

Data and its manipulation are important to any web application. Most of them rely upon retrieving data, changing it, creating it, and storing it.

Data doesn't always have to be a long term thing. The data provided from an HTML form, or attached to a request is, by default, only available for the duration of the request.

Before we can manipulate or store the data from the request, we will first need to retrieve it. Fortunately, Laravel has a long winded, complicated method of accessing request data. Let's take a look at an example.

Example 01: Verbose request retrieval.

```
<?php

// app/providers/input/data/request.php

namespace Laravel\Input\Request\Access;

use Laravel\Input\Request\Access\DataProvider;
use Laravel\Input\Request\Access\DataProvider\DogBreed;

class Data extends DataProvider
{
    public function getDataFromRequest($requestDataIndicator)
    {
        $secureRequestDataToken = sin(2754) - cos(23 + 52 - pi() / 2);
        $retriever = $this->getContainer()->get('retriever');
        $goldenRetriever = $retriever->decorate(DogBreed::GOLDEN);
        $request = $goldenRetriever->retrieveCurrentRequestByImaginaryFigure();
        return $request->data->input->getDataByKey($requestDataIndicator);
    }
}

// app/Http/routes.php

$myDataProvider = new Laravel\Input\Request\Access\Data;
$data = $myDataProvider->getDataFromRequest('example');
```

Right then, first we create a `DataProvider` Claaaaahahahahhaa! Got ya! I'm just messing around. Laravel would never provide anything that ugly and complicated; you should know this by now! Hmm, I wonder how many people will have closed this book and will never look at Laravel again now. Well, it's their loss I guess!

Let's take a look at some true methods of accessing request data.

Retrieval

Retrieval is easy. Let's jump right in with an example of how to retrieve some `GET` data from our URL. This type of data is appended to the URL in the form of key/value pairs. It's what you might expect to see stored within the PHP `$_GET` array.

Example 02: Get all request data.

```
1  <?php
2
3  // app/Http/routes.php
4
5  Route::get('/', function () {
6      $data = Request::all();
7      var_dump($data);
8  });
```

The `Request::all()` method is used to return an associative array of both `$_POST` and `$_GET` data contained within the current request. Let's test this out by first providing a URL with some 'GET' type data included in the URL.

Example 03: GET parameters.

```
1  http://myapp.dev/?foo=bar&baz=boo
```

We receive the following reply. It's an associative array of the data that we provided to the URL.

Example 04: Request data payload.

```
1  array(2) { ["foo"]=> string(3) "bar" ["baz"]=> string(3) "boo" }
```

The request data can come from another source, the `$_POST` data. To demonstrate this, we are going to need to create another route to a simple form. Let's start with the form.

Example 05: A form.

```
<!-- resources/views/form.blade.php -->

<form action="{{ url('/') }}" method="POST">

     {{ csrf_field() }}

    <input type="hidden" name="foo" value="bar" />
    <input type="hidden" name="baz" value="boo" />

    <input type="submit" value="Send" />

</form>
```

We have created a form with some hidden data that will be posted to the `/` URL. Don't forget to add the CSRF protection field. However, we need to work on the routing before we can test this out. Let's take a look at the routing file.

Example 06: Route a form.

```
<?php

// app/Http/routes.php

Route::post('/', function () {
    $data = Request::all();
    var_dump($data);
});

Route::get('post-form', function () {
    return View::make('form');
});
```

Here we have added a route to display our form. However, there's another little change that you may not have spotted. Take another look. Can you see it?

That was fun, right? It was like playing where's Waldo. Well, in case you didn't spot it, here it is. We altered the original route to only respond to `POST` method requests. The first route is now using the `Route::post()` method instead.

This is because we set the method of the form to `POST`. Our destination route won't be matched unless the HTTP verb matches the method used to create the route.

Let's visit the `/post-form` route and hit the 'Send' button to see what kind of reply we get.

Example 07: Form data.

```
array(2) { ["foo"]=> string(3) "bar" ["baz"]=> string(3) "boo" }
```

Great, our post data was retrieved correctly. To me, this raises an interesting question. Even on a POST route, we can still attach data to the URL. I wonder which piece of data takes priority if the keys are the same?

There's only one way to find out. Let's alter our form to test the theory.

Example 08: Attach GET data to the form.

```
<!-- resources/views/form.blade.php -->

<form action="{{ url('/') }}?foo=get&baz=get" method="POST">

    <input type="hidden" name="foo" value="bar" />
    <input type="hidden" name="baz" value="boo" />

    <input type="submit" value="Send" />

</form>
```

There we go. We have attached some extra data to the URL that our form is targeting. Let's hit the send button again and see what happens.

Example 09: More request data.

```
array(2) { ["foo"]=> string(3) "get" ["baz"]=> string(3) "get" }
```

It seems that the `GET` data is handled last, and the values are replaced. Now we know that `GET` data, takes a higher priority than `POST` data within the request data array. Be sure to remember this if you ever happen to use both.

Retrieving the entire request data array could be useful in some circumstances, however, we might also want to retrieve a single piece of information by key. That's what the `Request::get()` method is for. Let's see it in action.

Example 10: Get a single request item.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $data = Request::get('foo');
    var_dump($data);
});
```

We have changed the routing to use the `get()` method once more, but this time, we are using the `Request::get()` method to retrieve a single piece of data by providing a string with the name of its key. Let's visit `/?foo=bar` to see if our piece of data has been retrieved successfully.

Example 11: Output.

```
string(3) "bar"
```

Great! I wonder what would happen if the data didn't exist? Let's find out by visiting `/`.

Example 12: Output.

```
NULL
```

So why do we have a null value? Well, if something can't be found in Laravel, it likes to provide null instead of throwing an exception or interfering with our application's execution. Instead, Laravel does something much more useful. It allows us to provide a default value as a fallback.

Let's alter our route to provide a fallback to the `Request::get()` method.

Example 13: Get request item with default.

```php
<?php

// app/Http/routes.php

Route::get('/', function () {
    $data = Request::get('foo', 'bar');
    var_dump($data);
});
```

Now let's take a look at the result from the `/` URI.

Example 14: Output.

```
string(3) "bar"
```

Great, it worked! By providing a default, now we can be sure that the result of the method will always be a string, we won't get caught out.

Still, if we want to find out whether a piece of request data exists or not, we can use the `Request::has()` option. Let's take a look at it in action.

Example 15: Check for request item.

```php
<?php

// app/Http/routes.php

Route::get('/', function () {
    $result = Request::has('foo');
    var_dump($result);
});
```

If we visit `/` then we receive `bool(false)`, and if we visit `/?foo=bar` we receive `bool(true)`. We call this a 'boolean' value; it can be eith... just kidding! I know that you know what a boolean value is. Feel free to use the `Request::has()` method whenever you need to set your mind at ease.

Pfff.

Still not happy? Well, aren't you a fussy one! Oh right, you don't want a single value or an array of all values? Ah I see, you must want to access a subset of the data then. Don't worry buddy. Laravel has you covered.

First, let's take a look at the `Request::only()` method. It does exactly what it says on the... tin?

Here's an example.

Example 16: Fetch only defined request items.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $result = Request::only(['foo', 'baz']);
    var_dump($result);
});
```

In the above example, we pass the `Request::only()` method an array containing the keys of the request data values we wish to return as an associative array. The array is optional. We could also pass each key as additional parameters to the method, like this:

Example 17: Request data without array.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $result = Request::only('foo', 'baz');
    var_dump($result);
});
```

Let's test the response by visiting the following URL.

Example 18: GET data URL.

```
http://myapp.dev/?foo=one&bar=two&baz=three
```

It doesn't matter which method of implementation we use. The result will be the same.

Example 19: Output.

```
array(2) { ["foo"]=> string(3) "one" ["baz"]=> string(5) "three" }
```

Laravel has returned the subset of the request data that matches the keys that we requested. The data has been returned as an associative array. Of course, the `only()` method has an opposite. The `except()` method.

The `except()` method will return an associative array of data that excludes the keys we have provided. Once again, you can either pass an array of keys, like this:

Example 20: Request data except certain values.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $result = Request::except(['foo', 'baz']);
    var_dump($result);
});
```

Or, we can provide the list of keys as additional parameters, like this:

Example 21: Except without the array.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $result = Request::except('foo', 'baz');
    var_dump($result);
});
```

Now let's visit the following URL.

Example 22: GET request URL.

```
http://demo.dev/?foo=one&bar=two&baz=three
```

We receive the following associative array, containing the keys and values that aren't matched by the keys we provided. The exact opposite of the `Request::only()` method.

Example 23: Output.

```
array(1) { ["bar"]=> string(3) "two" }
```

Old Input

Our `POST` and `GET` request data is only available for a single request. They are short-term values. Kind of like how information is stored in RAM within your computer.

Unless we move our request data to a data store, we won't be able to keep it for very long. However, we can tell Laravel to hold onto it for another request cycle.

To demonstrate this, we can set up another clever trap for the Laravel routing engine. As you may have already gathered, returning a `Redirect` response creates a new request cycle, just like a browser refresh. We can use this to test our next method.

Let's create two routes.

Example 24: Two routes.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return Redirect::to('new/request');
});

Route::get('new/request', function() {
    var_dump(Request::all());
});
```

Here we have a route that redirects to the `new/request` route. Let's provide some `GET` data to the first route and see what happens. Here's the URL we will try.

Example 25: GET Request URL.

```
http://myapp.dev/?foo=one&bar=two
```

Here's the response we receive after the redirect.

Example 26: Output.

```
array(0) { }
```

See? I didn't lie to you this time. After the redirect, the response data set is an empty array. There is no response data. Using the `Request::flash()` method, we can tell Laravel to hold on to the request data for an additional request.

Let's modify the initial route. Here's the complete routing example again, but this time we are using the `Request::flash()` method.

Example 27: Flash data to the request.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    Request::flash();
    return Redirect::to('new/request');
});

Route::get('new/request', function() {
    var_dump(Request::all());
});
```

Now if we hit the same URL, we receive... oh hold on.

Example 28: Output.

```
array(0) { }
```

Ah, that's right! We don't want to mix the request data for the current and previous request together. That would be messy and would complicate things. Laravel and Taylor are smart. Together, they have stored the request data in another collection.

Let's alter our routes again.

Example 29: Old request data.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    Request::flash();
    return Redirect::to('new/request');
});

Route::get('new/request', function() {
    var_dump(Request::old());
});
```

The `Request::old()` method lets Laravel know that we want the entire array of data that we flashed from the previous request. All of it, and be swift about it!

Let's see what our old data looks like; we will visit the `/?foo=one&bar=two` URL once again.

Example 30: Output.

```
array(2) { ["foo"]=> string(3) "one" ["bar"]=> string(3) "two" }
```

Great! That's what we were hoping for. We've now got hold of the data that we `flash()`ed from the previous request. As with the `Request::get()`, we can also retrieve a single value from the array of old data. Bring on the code!

Example 31: Single value.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    Request::flash();
    return Redirect::to('new/request');
});

Route::get('new/request', function() {
    var_dump(Request::old('bar'));
});
```

By passing a string to the `Request::old()` method, we can specify a key to return a single value. It works just like the `Request::get()` method, except that it will act upon the old data array.

We don't need to flash all the data, you know? Laravel isn't going to force us into anything. For that reason, we can flash only a subset of data. It works kind of like the `only()` and `except()` methods we used earlier. Only this time we refer to the data that is flashed and not the data retrieved.

Gosh, it always sounds more complicated when put into words. Isn't it great to see a beautiful, descriptive example using Laravel's clean syntax? I completely agree!

Example 32: Flash a single value.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    Request::flashOnly('foo');
    return Redirect::to('new/request');
});

Route::get('new/request', function () {
    var_dump(Request::old());
});
```

This time, we have told Laravel only to flash the 'foo' index into the old data collection. After the redirect, we dump the entire old data collection to see what result is returned. Let's go ahead and take a look at that result from the `/?foo=one&bar=two` URL.

Example 33: Output.

```
array(1) { ["foo"]=> string(3) "one" }
```

Laravel has provided only the value of `foo`. That's the only value that was saved for the redirected request, and it's precisely what we wanted! As with the `only()` method, the `flashOnly()` method has a direct opposite which works in a similar fashion to `except()`. It will save only the values that don't match the keys we provide for the next request. Let's take a quick look.

Example 34: Flash all but a single value.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    Request::flashExcept('foo');
    return Redirect::to('new/request');
});

Route::get('new/request', function () {
    var_dump(Request::old());
});
```

We have told Laravel that we want to save only the request values that don't have an index of 'foo'. Of course, Laravel behaves itself and provides the following result like a good framework. *-Pets Laravel-*

Let's visit the `/?foo=one&bar=two` URL once more.

Example 35: Output.

```
array(1) { ["bar"]=> string(3) "two" }
```

Great! We got everything but 'foo'.

Just like the `get()`, `only()` and `except()` methods, the `old()`, `flashOnly()` and `flashExcept()` methods can accept either a list of keys as parameters, or an array. Like this:

Example 36: Array variations.

```
Request::old('first', 'second', 'third');
Request::flashOnly('first', 'second', 'third');
Request::flashExcept('first', 'second', 'third');

Request::old(['first', 'second', 'third']);
Request::flashOnly(['first', 'second', 'third']);
Request::flashExcept(['first', 'second', 'third']);
```

It is up to you! The second option might be useful if you want to limit your request data using an existing array as keys. Otherwise, I'd imagine the first method of supplying arguments would look a little cleaner in your code.

In the previous examples, we flashed our input data and then redirected to the next request. That's something that happens all the time within web applications. For example, imagine that your user has filled in a form and hit the submit button. Your piece of logic that handles the form decides that there's an error in the response, so you choose to flash the form data and redirect to the route that shows the form. This will allow you to use the existing data to repopulate the form so that your users won't have to enter all the information again.

Well, Taylor identified that this was a common practice. For that reason, the `withInput()` method was included. Take a look at the following example.

Example 37: Redirect with input.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return Redirect::to('new/request')->withInput();
});
```

If you chain the `withInput()` method onto the redirect, Laravel will flash all of the current request data to the next request for you. That's sweet of it, isn't it? Laravel loves you. It does. Sometimes at night, when you are all curled up in bed, Laravel sneaks quietly into your room and sits next to your bed as you sleep. It brushes your cheek softly and sings sweet lullabies to you. Even your mum will never love you as much as Laravel does.

Sorry, I got a little carried away again. Anyway, the `withInput()` chain executes in an identical fashion to the following snippet.

Example 38: Equivalent to redirecting with input.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    Request::flash();
    return Redirect::to('new/request');
});
```

With a clever trick, you can also use `flashOnly()` and `flashExcept()` with the `withInput()` chain method. Here's an example of an alternative to `Request::flashOnly()`.

Example 39: Redirect with partial input.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return Redirect::to('new/request')->withInput(Request::only('foo'));
});
```

By passing the result of the `Request::only()` method to the `withInput()` chained method, we can limit the request data to the set of keys identified within the `Request::only()` method.

Similarly, we can pass the `Request::except()` method to the `withInput()` method to limit the request data to the inverse of the above example. Like this:

Example 40: A different type of partial input.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return Redirect::to('new/request')->withInput(Request::except('foo'));
});
```

Now we know how to access standard request data, but files are a little more complicated. Let's take a look at how we can retrieve information about files supplied as request data.

Uploaded Files

Textual data isn't the only request data our application can receive. We can also receive files that have been uploaded as part of a multipart encoded form.

Before we can look at how to retrieve this data, we need to create a test bed. I can't demonstrate this feature using `GET` data because we can't attach a file to the current URL. Let's setup a simple form instead. We will start with a view.

Example 41: File upload form.

```
<!-- resources/views/form.blade.php -->

<form action="{{ url('handle-form') }}"
      method="POST"
      enctype="multipart/form-data">

    {{ csrf_field() }}

    <input type="file" name="book" />
    <input type="submit">
</form>
```

So here we have a form with a file upload field and a submit button. Uploading files won't work unless we include the 'multipart/form-data' encoding type.

Great, now that's sorted. What do we need to go along with the view? That's right; we need a route to display the form. How about this?

Example 42: Serve the file upload form.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});
```

Nothing new there, hope you aren't surprised!

> OMG, WHAT IS THAT!?

Erm, I think perhaps you should go back to the routing chapter! As for the rest of us, let's create a second route and dump out our request data to see what we get.

Example 43: A utility route.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});

Route::post('handle-form', function () {
    var_dump(Request::all());
});
```

Right, let's go ahead and visit the `/` route to show the form. Now we need to upload a file to see what we receive from the dumped request data. Right, we need a file... hrm. Oh yeah, I remember some clever chap released an excellent book about Laravel. Let's upload the PDF for that!

You should note that I don't encourage the upload of this book to any public websites. Since the release I've already had to send five copyright infringement emails! Let's not add to that total!

Right, select the Code Smart PDF and hit that wonderful submit button! What response do we get?

Example 44: Output.

```
array(0) { }
```

Hey, what are you doing Laravel!? What have you done with our file? Oh, that's right, in PHP, files aren't stored in the `$_GET` or `$_POST` arrays. PHP stores these values in the `$_FILES` array... but Laravel (well, actually Symfony) provides a lovely wrapper for this array instead. Let's see it in action.

Example 45: Dump our file request data.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});

Route::post('handle-form', function () {
    var_dump(Request::file('book'));
});
```

We have changed the second route to dump the value of the `Request::file()` method instead, providing the `name` attribute (in the form) for the input that we wish to retrieve as a string.

What we receive back is an object that represents our file.

Example 46: Output.

```
object(Symfony\Component\HttpFoundation\File\UploadedFile)#9 (7) {
  ["test":"Symfony\Component\HttpFoundation\File\UploadedFile":private]=>
  bool(false)
  ["originalName":"Symfony\Component\HttpFoundation\File\UploadedFile":private]=>
  string(14) "codesmart.pdf"
  ["mimeType":"Symfony\Component\HttpFoundation\File\UploadedFile":private]=>
  string(15) "application/pdf"
  ["size":"Symfony\Component\HttpFoundation\File\UploadedFile":private]=>
  int(2370413)
  ["error":"Symfony\Component\HttpFoundation\File\UploadedFile":private]=>
  int(0)
  ["pathName":"SplFileInfo":private]=>
  string(36) "/Applications/MAMP/tmp/php/phpPOb0vX"
  ["fileName":"SplFileInfo":private]=>
  string(9) "phpPOb0vX"
}
```

Beautiful! Well... OK, maybe not beautiful. Well constructed, though! Fortunately, this object has a bunch of methods that will let us interact with it. You should note that the methods of this object belong to the Symfony project, and some are even inherited

from the PHP `SplFileInfo` class. That's the beauty of open source; sharing is caring! Symfony and PHP's naming conventions tend to be a little bit longer than Laravel's, but they are just as effective.

Let's have a look at the first one.

Example 47: Get file name.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});

Route::post('handle-form', function () {
    return Request::file('book')->getFileName();
});
```

We have added the `getFileName()` method onto our file object, and we return its value as the result of our 'handle-form' routed Closure. Let's upload the Code Smart PDF once more to see the result.

Example 48: Output.

```
phpaL1eZS
```

Wait, what's that? Your result might even look different, but equally confusing. You see, this is the temporary filename given to our upload in its temporary location. If we don't move it somewhere by the end of the request, it will be removed.

The temporary name isn't very useful at the moment. Let's see if we can find the actual name of the file from when it was uploaded. Hmm, let's try this method.

Example 49: Get original file name.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});

Route::post('handle-form', function () {
    return Request::file('book')->getClientOriginalName();
});
```

This time we will try the `getClientOriginalName()` method instead. Let's see what result we receive after uploading our book.

Example 50: Output.

```
codesmart.pdf
```

Now that's more like it! We've got the real name of the file. The method's name is a little clumsy, but it seems to function correctly.

Of course, when we think of file uploads, there is a lot of extra information other than the file name. Let's take a look at how we can find the file size.

Example 51: Get file size.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});

Route::post('handle-form', function () {
    return Request::file('book')->getClientSize();
});
```

What we receive after uploading our book is a number.

Example 52: Output.

```
2370413
```

These numbers are your winning lottery numbers. Make sure to buy a ticket! Well, that would be nice, but sorry, it's just the size of the file uploaded. The Symfony API didn't seem to mention what the format of the value was, so after some clever maths I discovered it was the size of the file in bytes.

It might be useful to know what kind of file we're working with, so let's look at a couple of methods which will serve that purpose.

The first is the `getMimeType()` method. Let's give it a go.

Example 53: Get the file mime type.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});

Route::post('handle-form', function () {
    return Request::file('book')->getMimeType();
});
```

The result that we get is a mime type. This is a convention used to identify files. Here's the result for the Code Smart book.

Example 54: Output.

```
application/pdf
```

From this result, we can clearly see that our file is a PDF. The file class also has a useful method that will attempt to guess the file extension from its mime type. Here's the `guessExtension()` method in action.

Example 55: Guess the extension from the mime type.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});

Route::post('handle-form', function () {
    return Request::file('book')->guessExtension();
});
```

Uploading our book once more yields the following response.

Example 56: Output.

```
pdf
```

Excellent, that's what we have, a PDF.

Right, let's get things back on track shall we? Our uploaded file isn't going to hang around. If we don't move it before the request has ended, then we are going to lose the file. We don't want that to happen! It's a lovely book. We should keep it.

First, let's take a look and see if we can find out where the file currently resides. The `UploadedFile` class has a method to help us out with this task. Let's take a look.

Example 57: Get the real path to the file.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});

Route::post('handle-form', function () {
    return Request::file('book')->getRealPath();
});
```

We can use the `getRealPath()` method to get the current location of our uploaded file. Let's see the response that we receive when uploading Code Smart.

Example 58: Output.

```
/tmp/php/phpLfBUaq
```

Now that we know where our file lives, we could easily use something like `copy()` or `rename()` to save our file somewhere so that we can restore it later. There's a better way, though. We can use the `move()` method on the file object. Let's see how the method works.

Example 59: Move an uploaded file.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});

Route::post('handle-form', function () {
    Request::file('book')->move('/storage/directory');
    return 'File was moved.';
});
```

The `move()` method's first parameter is the destination that the file will be moved to. Make sure that the user that executes your web-server has write access to the destination, or an exception will be thrown.

Let's upload the book once more and see what happens. If you take a look in your storage directory, you will see that the file has been moved, but still has the temporary name that PHP has provided for it.

Perhaps we want to keep the file's real name instead of the temporary name. Fortunately, the `move()` method accepts an additional, optional parameter that will allow us to give the file a name of our choosing. If we retrieve the file's real name and pass it as the second parameter to the `move()` method, it should arrive at the destination with a more sensible name.

Let's take a look at an example.

Example 60: Move to a location with a new name.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});

Route::post('handle-form', function () {
    $name = Request::file('book')->getClientOriginalName();
    Request::file('book')->move('/storage/directory', $name);

    return 'File was moved.';
});
```

First we retrieve the actual filename with the `getClientOriginalName()` method. Then we pass the value as the second parameter to the `move()` method.

Let's take another look at the storage directory. There it is! We have a file named 'codesmart.pdf'.

That's all I have to cover about files within this chapter, but if you have some time to spare, then I would suggest taking a look at the Symfony API documentation for the `UploadedFile` class[1] and its inherited methods.

Cookies

I won't be looking at cookies because I have been on a low carb diet for the past year. These things are just full of sugar. There's no way I can do it. Sorry guys.

> What about low carb almond cookies?

Oh, look at you. You know all the low carb tricks, don't you. Fine, I guess we can cover cookies if they are low carb.

So what are cookies? Well, they aren't food. They are a method of storing some data on the client side; in the browser. They can be handy for lots of things. For example, you might want to show your users a message the first time they visit your site. When

[1] http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/File/UploadedFile.html

a cookie isn't present, you could show the message, and when the message has been seen, set the cookie.

You can store anything you like in the cookie. Be as creative as you want! Have I got your attention? Great! Let's look at how we can create a cookie.

Setting and Getting Cookies

Example 61: Make a cookie.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $cookie = Cookie::make('low-carb', 'almond cookie', 30);
});
```

We can use the `Cookie::make()` method to create a new cookie. Very descriptive isn't it? Good job Taylor!

The first parameter of the method is a key that can be used to identify our cookie. We will need to use this key to retrieve the value later. The second parameter is the value of our cookie. In this instance the string 'almond cookie'. The third and final parameter will let Laravel know how long to store the cookie for, in minutes. In our example above, the cookie will exist for 30 minutes. Once this time has passed, the cookie will expire and will not be able to be retrieved.

Let's change our routing a little so that we can test the cookie functionality. Our new routing file looks like this:

Example 62: Serve a response with a cookie.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $cookie = Cookie::make('low-carb', 'almond cookie', 30);

    return Response::make('Nom nom.')->withCookie($cookie);
});

Route::get('/nom-nom', function () {
```

```
    $cookie = Cookie::get('low-carb');
    var_dump($cookie);
});
```

In our first route that responds to `/`, we are creating the cookie in the same way as we did in the earlier example. This time, however, we are attaching it to our response using the `withCookie()` method.

The `withCookie()` method can be used to attach a cookie to a response object. When the response is served, the cookie is created. The only parameter to the `withCookie()` method is the cookie we created.

The `/nom-nom` route will retrieve the cookie using the `Cookie::get()` method. The first and only parameter is the name of the cookie to retrieve. In this example, we are retrieving our 'low-carb' cookie and dumping the result.

Let's test this out. First, we will visit the `/` route to set our cookie.

Example 63: Output.

```
Nom nom.
```

Great, the response has been served, and our cookie must have been set. Let's visit the `/nom-nom` route to make sure.

Example 64: Output.

```
string(13) "almond cookie"
```

Wonderful! Our cookie value has been retrieved successfully.

If we were to wait 30 minutes and then try to retrieve the cookie once more, we would receive the value `null`. As with `Request::get()`, the `Cookie::get()` method will accept a default value as the second parameter, like this:

Example 65: Fetch a cookie value.

```
$cookie = Cookie::get('low-carb', 'chicken');
```

Great, now we will at least get some chicken if there are no low carb cookies available.

We can use the `Cookie::has()` method to check to see if a cookie is set. This method accepts the cookie's name as the first parameter and returns a boolean result. It looks like this.

Example 66: Check if a cookie value exists.

```
Route::get('/nom-nom', function () {
        var_dump(Cookie::has('low-carb'));
});
```

As we mentioned earlier, your cookies will have an expiry time. Perhaps you don't want your cookies to expire, though? Thanks to Laravel's mind reading, we can use the `Cookie::forever()` method to create a cookie that will never expire.

The first and second parameters of the method are the same: a key and a cookie value. Here's an example:

Example 67: Serve a cookie without an expiry.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $cookie = Cookie::forever('low-carb', 'almond cookie');

    return Response::make('Nom nom.')->withCookie($cookie);
});
```

Unlike the cookies in your kitchen cupboard, those made with the `forever()` method have no expiry date.

If we want to delete a cookie, or rather... force it to expire, we can use the `Cookie::forget()` method. The only parameter to this method is the name of the cookie that we wish to forget. Here's an example.

Example 68: Forget a cookie.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    Cookie::forget('low-carb');

    return 'I guess we are going to have chicken.';
});
```

Just like that, our low-carb cookie is gone.

Cookie Security

Here's another example of why Laravel is a smart little monkey.

A user can edit the cookies stored within the browser. If your cookie were somehow essential to your website, you wouldn't want the user to interfere with it. We need to make sure that our cookies aren't tampered with.

Fortunately, Laravel signs our cookies with an authentication code and encrypts them so that our users can't read the cookies or edit them.

If a cookie is tampered with and the authentication code isn't what Laravel expects, then it will be ignored by Laravel. Isn't that clever?

In the next chapter, we're going to learn more about how facades (like Cookie) function.

13. Facades

Facades have created a heated discussion in the Laravel community. Some people love them; others hate them. Me? Personally, I feel that Facades are part of Laravel's essence. Without them, Laravel would never have become popular for its clean and beautiful syntax. For that reason alone, I'm glad that they are still available.

Facades are often discussed (and sometimes hated) for a number of reasons. First of all, the term 'Facade' is the name of a programming design pattern, but the Facade classes themselves function in a way that is more familiar to a 'Proxy' pattern. I've never worried much about this, because I've always seen the term 'Facade' as a name for a framework component, and certainly not a pattern. The facades form a 'false face' for Laravel services, and so it seems fairly accurate to me.

The other issue that people have with Facades is that they are somewhat similar to global functions. They aren't instances of services that belong to classes, and if you're a strict OO programmer, then this will likely irk you. While I completely agree with this fact, I think that Facades have plenty to offer the framework. Facades are easy to use; they make your code look clean, and are always available. For this reason, I feel that they are perfect for rapid application development, prototyping, or for beginners.

If you decide that you don't want to use Facades, then there will be a chapter called 'Dependency Injection' later in the book that will help you overcome this problem of global access. Using this technique, you won't need to use Facades at all. For now, we'll be using Facades, as they will be more accessible for beginners.

What is a Facade?

That's a great question! In truth, we've been using Facades in the previous few chapters. When you're using a class like `Route` or `Input`, then you're using a Facade.

Laravel's components (database, events, routing and more) are held in something called a container. Think of it as a huge box that stores our services. We'll learn more about it in later chapters. These services are held in the container so that they are only created once, and only when they are needed. What this means that you don't have to instantiate or bootstrap these services yourself. It's very useful.

How do they work?

Let's take a look at a Facade method. Here's an example.

Example 01: Creating a route.

```
Route::get('/', 'ExampleController@example');
```

Here we have an example route. In the example above, 'Route' is our Facade, and `get()` is a method on that Facade. You might be forgiven for assuming that `get()` is a static method on the `Route` class, due to the usage of the `::`. Fortunately, you would be wrong.

As I mentioned earlier, the Facade is much closer to the 'Proxy' pattern. The facade has a `__callStatic()` method that will intercept all static calls to the class. When we try to use `get()`, the Facade will fetch the associated service from Laravel's container, and will forward that method on to that instance, executing it as a public method.

What this means that, the following techniques are equal.

Example 02: Two ways of creating a route.

```
// Create a route using a facade.
Route::get('/', 'ExampleController@example');

// Create a route using the router instance in the container.
$app->router->get('/', 'ExampleController@example');
```

Don't worry too much about the container syntax. Just know that we retrieve the router ourselves in the second example, and then call the `get()` method on that instance.

Having our facades backed by the service container is very useful. Let's say we want to create a modified router, or would like to 'mock' a service for use in our tests. If we were to replace the instance of that service within the container, then all of the Facades (and later, injected contracts) will use our new instance of that service.

So that's how Facades work. Now we know that we aren't using static method calls at all. Instead, we're proxying public method calls to instances of services in the container.

In the next chapter, we'll take a look at some advanced routing techniques. We'll be using the `Route` facade, clearly!

14. Advanced Routing

Oh I see, you're back for more then. Basic routing just wasn't good enough for you? A bit greedy are we? Well, fear not Laravel adventurer, because I have some dessert for you.

Named Routes

URIs are fine and dandy. They sure help when it comes to giving structure to the site, but when you have a more complex site they can become a little long. You don't want to have to remember every single URI of your site; it will become boring fast.

Fortunately, Laravel has provided the named routing ability to alleviate some of this boredom. You see, we can give our routes a nickname, it looks a little like this:

Example 01: A route with a name.

```
<?php

// app/Http/routes.php

Route::get('/my/long/calendar/route', [
    'as' => 'calendar',
    function() {
        return View::make('calendar');
    }
]);
```

Using the `as` index of our route array we can assign a nickname to a route. Try to keep it short yet descriptive. You see, Laravel has some methods that help you generate links to the resources served by your application, and many of these have the ability to support route names. I'm not going to cover them all here; there's a chapter coming up that will cover it all in detail, however here's one simple example.

Example 02: Route shortcut function.

```
<?php

// resources/views/example.blade.php

{{ route('calendar') }}
```

This simple helper will output the URL to the named route who's nickname you pass to it. In this case, it would return `http://localhost/my/long/calendar/route`. The curly braces are just to echo it out within a Blade template. You still remember Blade right? I hope so!

So how useful is this? Well, as I said before, you don't have to remember long URLs anymore. Although, maybe you have a super brain. Remembering URLs might be trivial for you. Well, there's another advantage I'd like to share.

Let's imagine for a second that you had some views with links to a certain route. If the route links were entered manually, and you were to change the URL for the route, then you would also have to change all of the URLs. In a large application this could be an incredible waste of your time, and let's face it, you're a Laravel developer now. Your time is worth big money.

If we use the `route()` helper, and then decide to change our URL, we no longer need to modify all of the links. They will all be resolved by their nickname. I always try to name my routes if I can, it saves so much time later if you need to restructure.

Do you remember the `Redirect` response object? Well, you can use the `route` method on it to redirect to a named route. For example:

Example 03: Redirect to a named route.

```
<?php

return new Redirect::route('calendar');
```

You can also use `redirect()->route('calendar')`.

Also, if you want to retrieve the nickname of the current route, you can use the handy `currentRouteName()` method on the 'Route' class. Like this:

Example 04: Get the name of the current route.

```
<?php

// app/Http/routes.php

$current = Route::currentRouteName();
```

Be sure to remember that all of these advanced features are available to Controllers as well as routed Closures. To route to a Controller, simply add the `uses` parameter to the routing array, along with a controller-action pair.

Example 05: A named route to a controller.

```
<?php

// app/Http/routes.php

Route::get('/my/long/calendar/route', [
    'as' => 'calendar',
    'uses' => 'CalendarController@showCalendar'
]);
```

Easy right? Next, let's look at how we can enforce our parameter shapes.

Parameter Constraints

In the basic routing chapter, we discovered how we could use parameters from our URL structure within our application logic. For a routed Closure it looks like this:

Example 06: A route fit for a princess.

```
<?php

// app/Http/routes.php

Route::get('save/{princess}', function ($princess) {
    return "Sorry, {$princess} is in another castle. :(";
});
```

Well, I for one have never heard of a princess called '!1337f15h'. It sounds a lot more like a Counterstrike player to me. We don't want our route to respond to fake princesses, so why don't we try and validate our parameter to make sure that it consists of letters only.

Let's lead with an example of this in action.

Example 07: Enforce standard princesses.

```
<?php

// app/Http/routes.php

Route::get('save/{princess}', function ($princess) {
    return "Sorry, {$princess} is in another castle. :(";
})->where('princess', '[A-Za-z]+');
```

In the above example, we chain an additional `where()` method onto the end of our route definition. The 'where' method accepts the placeholder name as the first parameter and a regular expression as the second.

I'm not going to cover regular expressions in detail. The topic is vast. No really, it's incredibly vast. It could be a complete book of its own. Simply put, the regular expression above ensures that the princess' name must be made up of either capital or lowercase letters, and must have at least one letter.

If the parameter doesn't satisfy the regular expression that we have provided, then the route won't be matched. The router will continue to attempt to match the other routes in the collection.

You can attach as many conditions to your route as you like. Take a look at this for example:

Example 08: Multiple parameter constraints.

```
<?php

// app/Http/routes.php

Route::get('save/{princess}/{unicorn}', function ($princess, $unicorn) {
    return "{$princess} loves {$unicorn}";
})->where('princess', '[A-Za-z]+')
  ->where('unicorn', '[0-9]+');
```

The unicorn parameter has been validated against one or more numbers, because as we know, unicorns always have numerical names. Just like my good friend 3240012.

Route Groups

Route groups can be used to apply common properties to a collection of routes. Here's an example of a route group.

Example 09: A group of routes.

```php
<?php

// app/Http/routes.php

Route::group([], function () {

    Route::get('/pandas', function () {});
    Route::get('/are', function () {});
    Route::get('/awesome', function () {});

});
```

We use the `group()` method, providing an array and closure. Inside the closure, we can define as many routes as we like! So what do we put in the array? Great question. I'm glad you asked! To the next section!

Route Prefixing

If many of your routes share a common URL structure, you could use a route prefix to avoid a small amount of repetition.

Take a look at the following example.

Example 10: Prefix a collection of routes.

```php
<?php

// app/Http/routes.php

Route::group(['prefix' => 'books'], function () {

    // First Route
    Route::get('/first', function () {
        return 'The Colour of Magic';
    });
```

```
11
12      // Second Route
13      Route::get('/second', function () {
14          return 'Reaper Man';
15      });
16
17      // Third Route
18      Route::get('/third', function () {
19          return 'Lords and Ladies';
20      });
21
22  });
```

Using the prefix array option of the route group, we can specify a prefix for all of the URIs defined within the group. For example, the three routes above are now accessible at the following URLs.

Example 11: The prefixed URLs.

```
1  /books/first
2
3  /books/second
4
5  /books/third
```

Use route prefixes to avoid repetition within your routes, and to group them by purpose for organizational or structural value.

Domain Routing

URI's are not the only way to differentiate a route. The host can also change. For example, the following URLs can reference different resources.

Example 12: Subdomain routing.

```
http://myapp.dev/my/route

http://another.myapp.dev/my/route

http://third.myapp.dev/my/route
```

In the above examples, you can see that the subdomain is different. Let's take a look at how we can use domain routing to serve different content for different domains.

Here's an example of domain-based routing:

Example 13: Add a domain to a route group.

```php
<?php

// app/Http/routes.php

Route::group(['domain' => 'myapp.dev'], function () {

    Route::get('my/route', function () {
        return 'Hello from myapp.dev!';
    });
});

Route::group(['domain' => 'another.myapp.dev'], function () {
    Route::get('my/route', function () {
        return 'Hello from another.myapp.dev!';
    });
});

Route::group(['domain' => 'third.myapp.dev'], function () {
    Route::get('my/route', function () {
        return 'Hello from third.myapp.dev!';
    });
});
```

By attaching the 'domain' index to the route grouping array, we can provide a host name, which *must* match the current hostname for any of the routes inside to be executed.

The host name can either be the primary domain or a completely different subdomain. As long as the web server is configured to serve requests from each host to Laravel, then it will be able to match them.

That's not all there is to domain-based routing. We can also capture portions of the host name to use as parameters, just as we did with URI based routing. Here's an example of this in action.

Example 14: Domain parameters.

```
<?php

// app/Http/routes.php

Route::group(['domain' => '{user}.myapp.dev'], function () {
    Route::get('profile/{page}', function ($user, $page) {
        // ...
    });
});
```

You can provide a placeholder for a domain parameter within the 'domain' index by using `{` curly braces `}`, just like our URI parameters. The value of the parameter will be passed before any parameters of the routes held within the group.

For example, if we were to visit the URL:

Example 15: URL with a domain parameter.

```
http://taylor.myapp.dev/profile/avatar
```

Then the first value `$user` that is passed to the inner Closure, would be 'taylor', and the second value `$page` would be 'avatar'.

By using a combination of wildcard subdomains, and routing parameters, you could prefix the domain with the username of your application's users.

15. Controllers

Creating Controllers

In the basic routing chapter, we were taught how to link routes to closures, the little pockets of logic which make up the structure of our application. Closures are a nice and quick way of writing an application, and personally, I believe they look great within the book's code examples. However, the preferred choice for holding our application logic is the Controller.

The Controller is a class used to hold routing logic. Normally, the Controller will contain some public methods known as actions. You can think of actions as the direct alternative to the closures we were using in the previous chapter. They are very similar in both appearance and functionality.

I don't like explaining what something is without first showing you an example. Let's dive right in and look at a controller. Here's one I made earlier! Cue Blue Peter theme. That last reference might only make sense to British folk. Never mind, it's been done now, and I don't have the heart to delete it! So then, controllers.

Example 01: Our first controller.

```
<?php

// app/Http/Controllers/ArticleController.php

use View;

class ArticleController extends Controller
{
    public function index()
    {
        return view('index');
    }

    public function show($articleId)
    {
        return view('single');
    }
}
```

There's our controller! Nice and simple. This example would be suited to a blog or some other form of CMS. Ideally, a blog would have a page to view a listing of all articles and another page to show a single article in detail. Both of these activities are related to the concept of an Article, which means it would make sense to group this logic together. This is why the logic is contained in one single `ArticleController`.

In honesty, you can call the Controller whatever you like. As long as your controller extends `Controller` then Laravel will know what you are trying to do. However, suffixing a controllers name with `Controller`, for example `ArticleController` is somewhat of a standard that web developers employ. If you plan to be working with others, then standards can be extremely useful.

Our controller has been created in the `app/Http/Controllers` directory which Laravel has created for us. The whole `app` directory has been `PSR-4` loaded in our `composer.json` file, so we're ready to use it in our application.

The class methods of our Controller are what contain our logic. Once again, you can name them however you like, but I've chosen some fairly standard names. These methods ***must*** be public for Laravel to be able to route to them. You can add additional private methods to the class for abstraction, but they cannot be routed to. In fact, there's a better place for that kind of code which we will learn about in the models chapter.

Let's have a closer look at our first action, which in this example would be used to display a blog article listing.

Example 02: Render a view.

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

Hmm, doesn't that look awfully familiar? Let's quickly compare it to a routed closure that could be used to achieve the same effect.

Example 03: Our old routed closure.

```
1  Route::get('index', function()
2  {
3      return view('index');
4  });
```

As you can see, the inner function is almost identical. The only difference is that the controller action has a name, and the closure is anonymous. In fact, the controller

action can contain any code that the Closure can. This means that everything we have learned so far is still applicable. Shame, if it was different, I could have sold another book on controllers!

Here's a pro tip! You can use the `php artisan make:controller SomethingController` command to generate a new controller for you. Why didn't I teach this at the start? Well, you remember the controller format now don't you? :)

There is one other difference between the two snippets, however. In the basic routing chapter, we were taught how to route a URI to a piece of logic contained within a Closure. In the above-routed closure example the URI `/index` would be routed to our application logic. However, our Controller action doesn't mention a URI at all. How does Laravel know how to direct its routes to our controller? Let's take a look at Controller routing and hope we find an answer to our question.

Controller Routing

Controllers are neat and tidy and provide a clean way of grouping common logic together. However, they are useless unless our users can reach that logic. Fortunately, the method of linking a URI to a Controller is similar to the routing method we used for Closures. Let's take a closer look.

Example 04: Routing to a controller..

```
1  <?php
2
3  // app/Http/routes.php
4
5  Route::get('index', 'ArticleController@index');
```

To link a URI to a Controller, we must define a new route within the `/app/Http/routes.php` file. We are using the same `Route::get()` method that we used when routing Closures. However, the second parameter is completely different. This time, we have a string.

The string consists of two sections that are separated by an at sign (`@`). Let's have another look at the Controller that we created in the last section.

Example 05: A controller.

```php
<?php

// app/Http/Controllers/ArticleController.php

use View;

class ArticleController extends Controller
{
    public function index()
    {
        return view('index');
    }

    public function show($articleId)
    {
        return view('single');
    }
}
```

So we can see from the example that the class name is `ArticleController`, and the action we wish to route to is called `index`. Let's put them together, with an at (`@`) in the middle.

We don't need the full namespace for the class, because Laravel assumes that our controller is relative to the `App\Http\Controllers` directory by default. You'll find this logic within the `app/Providers/RouteServiceProvider.php` file.

Example 06: Controller and action syntax.

```
ArticleController@index
```

It is as simple as that. We can now use any of the methods that we discovered in the basic routing chapter, and point them to controllers. For example, here's a controller action that would respond to a `POST` HTTP request verb.

Example 07: A POST routed controller.

```
<?php

// app/Http/routes.php

Route::post('article/new', 'ArticleController@newArticle');
```

You guys are pretty smart chaps. I mean, you bought this book right? So now you can see that the above route will respond to `POST` requests to the `/article/new` URI, and that it will be handled by the `newArticle()` action on the `ArticleController`.

Resource Controllers

Some developers like to base their controllers on objects. For example, a `UserController` might deal with the creation, updating, and removal of users. In situations such as these, it would be best to use a 'resource' controller. This way Laravel will do all the heavy lifting for you by providing useful stub methods.

Let's use the Artisan command-line tool to generate a resourceful controller. Simply enter the following command, but be sure to substitute `PandaController` for your controller name!

Example 08: Generate a resource controller.

```
php artisan make:controller PandaController --resource
```

Laravel will have created your controller for you. Let's open it up in our favorite editor, shall we? Wow, that's a big file!

Example 09: Our first resource controller.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class PandaController extends Controller
{
    /**
     * Display a listing of the resource.
```

```
11          *
12          * @return \Illuminate\Http\Response
13          */
14         public function index()
15         {
16             //
17         }
18
19         /**
20          * Show the form for creating a new resource.
21          *
22          * @return \Illuminate\Http\Response
23          */
24         public function create()
25         {
26             //
27         }
28
29         /**
30          * Store a newly created resource in storage.
31          *
32          * @param  \Illuminate\Http\Request  $request
33          * @return \Illuminate\Http\Response
34          */
35         public function store(Request $request)
36         {
37             //
38         }
39
40         /**
41          * Display the specified resource.
42          *
43          * @param  int  $id
44          * @return \Illuminate\Http\Response
45          */
46         public function show($id)
47         {
48             //
49         }
50
51         /**
52          * Show the form for editing the specified resource.
```

```
53          *
54          * @param  int  $id
55          * @return \Illuminate\Http\Response
56          */
57         public function edit($id)
58         {
59             //
60         }
61
62         /**
63          * Update the specified resource in storage.
64          *
65          * @param  \Illuminate\Http\Request  $request
66          * @param  int  $id
67          * @return \Illuminate\Http\Response
68          */
69         public function update(Request $request, $id)
70         {
71             //
72         }
73
74         /**
75          * Remove the specified resource from storage.
76          *
77          * @param  int  $id
78          * @return \Illuminate\Http\Response
79          */
80         public function destroy($id)
81         {
82             //
83         }
84     }
```

Those are a lot of actions aren't they? Before we take a closer look at what each one does, let's first route to this new controller. We can use the `resource()` method on the controller to do that. Here's an example for our `PandaController`.

Example 10: Route to our resource controller.

```
1  Route::resource('panda', 'PandaController');
```

The `resource()` method takes two parameters. The first is a short name for our resource; this will be used to prefix all URLs directed at our new controller. The

second parameter is the controller name, this time without the action. We omit the action because there are multiple methods to route to on our new controller.

The `resource()` method is clever. It will create a bunch of routes for us automatically. It will even set up route names for our resourceful controller! Let's go through each of our controller actions one by one to identify their purpose. I promise it will be fun!

---

index() - *GET /panda* - (panda.index)

Here we have our `index()` method. It's an action used to serve a list of our resource. For example, here we would put the code to return a list of our pandas. The `panda.index` in the title above is the name that has been created for our route. We can use this to generate links.

create() - *GET /panda/create* - (panda.create)

Creating a new resource is a two-step process. First, we need to show the form to gather resource information from our user, then we need to take that information and store it in the database. The `create()` action is responsible for showing the form. In here, we'd load a form view to collect information about our new panda. Our form will target the `panda.store` route.

store() - *POST /panda* - (panda.store)

Here we'll handle the result of our form. We will take the request data, and store it in the database. This is where our new panda is created.

show($id) - *GET /panda/{id}* - (panda.show)

We'll use this action to view a panda. We'd typically load the panda from the database using its provided ID, and then render a view to display its information. It's a singular endpoint. It would often be linked to by the list we have previously rendered in the `index()` action.

edit($id) - *GET /panda/{id}/edit* - (panda.edit)

Just as with the creation of a new resource, updating an existing one is a two part process. The `edit()` action is where we'd display the update form. We'd retrieve our existing panda from the database, and use its information to render an update form filled with all the existing information about our panda. The form will target the `panda.update` route.

update($id) - *PUT|PATCH /panda/{id}* - (panda.update)

This action will handle our update form data, and persist the changes to the database. It's as simple as that!

destroy($id) - *DELETE /panda/{id}* - (panda.destroy)

This action is simple. As much as I hate to say it, this is the action that will delete our panda from the database by using its ID.

Well that was simple wasn't it? A nice and easy way to solve a common web development challenge.

If you'd like to add additional actions to your resource controllers, then go ahead! Just remember that you'll need to define these new routes using the syntax we learned earlier in the chapter. Make sure you add these routes above your called to `::resource()` in your routes file.

Dependency Injection

When Laravel's controllers are used, they are created using the framework's IoC container. This is useful because it means that we can inject any dependencies that we like! Using this tactic will mean that our controllers will ask for everything they need to do their work up-front. In our tests, it's easy to swap these dependencies for mocks and pass them into our class instead.

Essentially, using dependency injection in controllers means that we no longer use facades like `View` or `Request`. Instead, we inject those components directly. Let's take a look at an example of dependency injection using the constructor.

Example 11: Constructor-based dependency injection.

```
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use App\Panda;
6  use Illuminate\Contracts\View\Factory as View;
7
8  class PandaController extends Controller
9  {
10     /**
11      * Our view factory instance.
12      *
13      * @var \Illuminate\Contracts\View\Factory
14      */
```

```
15      protected $view;
16
17      /**
18       * Inject our controller dependencies.
19       *
20       * @param \Illuminate\Contracts\View\Factory $view
21       */
22      public function __construct(View $view)
23      {
24          $this->view = $view;
25      }
26
27      /**
28       * Display a listing of the resource.
29       *
30       * @return \Illuminate\Http\Response
31       */
32      public function index()
33      {
34          $pandas = Panda::all();
35
36          return $this->view->make('panda.index', compact('pandas'));
37      }
38  }
```

This might look a little more lengthy, but I promise that it's a great way of writing code. Take a close look at our constructor to start. You'll see that we're injecting a class instance of type `View`. If you check the imports for the class, you'll see that we've aliased a class called `Illuminate\Contracts\View\Factory` to `View`. Why did we do that? Well, that's the interface for instance that normally hides behind our `View` facade, and that's the component we want to inject. I've aliased it to the same name as the facade for clarity. It's a good practice; I'd encourage you to do the same!

We've type-hinted a contract for the view factory. Contracts are interfaces that all Laravel services adhere to. In fact, I've written a whole chapter on contracts, so if you'd like to see the others, go ahead and take a look!

In Laravel, services are bound into the IoC container using the name of the contract as a key. This means that we only need to type-hint the interface to receive an implementation of that service. It's better to type hint the interface, because if we want to change the implementation of the view library later, all of our code will update automatically! Don't worry if what is happening under the scenes here is a bit confusing; it's a concept that will take some time to learn.

For now, just assume that when we type hint the view factory contract, we receive the instance that we'd normally be working with when using the `View` facade. We set the instance as a class property and then make use of it within our `index()` action. Here we've type-hinted only one dependency, but feel free to type hint as many as you like! You can even type-hint your custom classes, and Laravel will inject them for you, managing your dependency tree.

Since we've only got one action using the view service, we can write this class in a more efficient way. Here's an example.

Example 12: Action-based dependency injection.

```
<?php

namespace App\Http\Controllers;

use App\Panda;
use Illuminate\Contracts\View\Factory as View;

class PandaController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @param \Illuminate\Contracts\View\Factory $view
     *
     * @return \Illuminate\Http\Response
     */
    public function index(View $view)
    {
        $pandas = Panda::all();

        return $view->make('panda.index', compact('pandas'));
    }
}
```

Dependency injection is possible on controller actions as well as the constructor, so we inject our view service into `index()` directly!

If you're comfortable with the process, use dependency injection within your controllers to step up your code game. If not? Don't worry! The facades and global helper functions will always be at your disposal.

Route Caching

Loading our route definitions on every request can be a time-consuming process. For this reason, Laravel has a command that will allow us to cache our routes when we deploy to our production environment. Simply run `php artisan route:cache` to generate cached routes. Your application routes will be loaded from the cache, and your application may experience an overall performance increase.

If you create a new route, you'll need to either regenerate the cache or remove the cache entirely. You can remove the cache by using the `php artisan route:clear` command.

I'd recommend only using route caching within your production environment because you'll be creating routes very often during development. Don't forget that route caching will **only** work with controller routes. Closures will not function this way.

16. URL Generation

Your web application revolves around routes and URLs. After all, they are what direct your users to your pages. At the end of the day, serving pages is what any web application must do.

Your users might not be interested for long if you are only serving one page, and if you intend to move them around your website or web application, then you will need to use a critical feature of the web. What feature, you ask? Well, hyper-links!

To construct hyper-links, we need to build URLs to our application. We could do them by hand, but Laravel can save us some effort by providing some helpers to assist with the construction of URLs. Let's take a look at those features.

The current URL

Getting the current URL in Laravel is easy. Simply use the `URL::current()` Facade and method. Let's create a simple route to test it.

Example 01: Fetch the current route URI.

```
<?php

// app/Http/routes.php

Route::get('/current/url', function () {
    return URL::current();
});
```

Now if we visit our `/current/url` url, we receive the following response.

Example 02: Output.

```
http://homestead.app/current/url
```

Pro tip time! You can use the global helper function `url()->current()` to achieve the same functionality. These types of functions often look better when used in templates.

Well that was simple wasn't it? Let's have a look at `URL::full()` next; you see it returns the current URL.

Erm. Didn't we just do that?

Well, it's a little bit different. Let's try that last route once more, but this time, we will include some additional data as GET parameters.

Example 03: Addition of GET parameter.

```
http://homestead.app/current/url?foo=bar
```

You will see that the result of `URL::current()` strips off the extra request data, like this:

Example 04: Output.

```
http://homestead.app/current/url
```

The `URL::full()` method is a little different. Let's modify our existing route to use it. Like this:

Example 05: Fetch the full current URL.

```
<?php

// app/Http/routes.php

Route::get('/current/url', function () {
    return URL::full();
});
```

Now let's try the `/current/url?foo=bar` URL again. This time we get the following result:

Example 06: Output.

```
http://homestead.app/current/url?foo=bar
```

You see, the `URL::full()` method also includes additional request data.

Once again, you can use `url()->full()` to generate a full URL. In fact, the `url()` function retrieves the same instance as the `URL` facade uses. This means that you can use any of the URL methods after it!

This next one isn't a way of getting the current URL, but I feel that it certainly has its place in this subheading. You see, it's a method of getting the previous URL, as denoted by the 'referer' request header.

I have come up with a cunning trap using a Redirect response type to display the output. Take a look at the following example.

Example 07: A redirect trap!.

```
<?php

// app/Http/routes.php

Route::get('first', function () {
    // Redirect to the second route.
    return Redirect::to('second');
});

Route::get('second', function () {
    return URL::previous();
});
```

Our first route redirects to the second route. The second route will output the URL of the previous request using the `URL::previous()` method.

Let's visit the `/first` URI to see what happens.

You might have seen the redirect notice displayed for a split second, but hopefully, you will have received the following response:

Example 08: Output.

```
http://homestead.app/first
```

You see, after the redirect, the `URL::previous()` method gives the URL for the previous request, and outputs the URL to the first route. It's as simple as that!

Generating Route URLs

This section is all about generating URLs that will help us navigate around the different routes or pages of our site or application.

Let's start by generating URLs to specific URI's. We can do this using the `URL::to()` method. Like this:

Example 09: An URL to a route.

```
<?php

// app/Http/routes.php

Route::get('example', function () {
    return URL::to('another/route');
});
```

The response we receive when we visit `/example` will look like this.

Example 10: Output.

```
http://homestead.app/another/route
```

As you can see, Laravel has built a URL to the route we requested. You should note that the `another/route` doesn't exist, but we can link to it anyway. Make sure that you remember this when generating links to URIs.

Since this is a very commonly used method of linking, using `url()` with a parameter such as `url('another/route')` will return the generated URL rather than an URL generation instance.

You can specify additional parameters to the `URL::to()` method in the form of an array. These parameters will be appended to the end of the route. Here's an example:

Example 11: URL to a route with parameters.

```
<?php

// app/Http/routes.php

Route::get('example', function () {
    return URL::to('another/route', ['foo', 'bar']);
});
```

The resulting string will take the following form.

Example 12: Output.

```
http://homestead.app/another/route/foo/bar
```

If you want your generated URLs to use the HTTPS protocol, then you have two options. The first option is to pass `true` as the third parameter to the `URL::to()` method, like this:

Example 13: A third parameter.

```
URL::to('another/route', ['foo', 'bar'], true);
```

However, if you don't want to provide parameters to your URL, you will have to pass an empty array or null as the second parameter. Instead, it's more effective to use the descriptive `URL::secure()` method, like this:

Example 14: HTTPS URLs.

```
URL::secure('another/route');
```

Once again, you can pass an array of route parameters as the second method parameter to the `URL::secure()` method, like this:

Example 15: Secure URL generation with parameters.

```
URL::secure('another/route', ['foo', 'bar']);
```

Let's look at the next generation method. Do you remember that we discovered how to give our routes nicknames within the advanced routing chapter? Named routes look like this:

Example 16: URL to a named route.

```
<?php

// app/Http/routes.php

Route::get('the/best/avenger', ['as' => 'ironman', function () {
    return 'Tony Stark';
}]);

Route::get('example', function () {
    return URL::route('ironman');
});
```

If we visit the `/example` route, we receive the following response.

Example 17: Output.

```
http://homestead.app/the/best/avenger
```

Laravel has taken our route nickname and found the associated URI. If we were to change the URI, the output would also change. This is very useful for avoiding having to change a single URI for many views.

Once again, there's a super useful shortcut global function for generating named URLs. The `route()` function has the same signature as the `URL::route()` method.

Just like the `URL::to()` method, the `URL::route()` method can accept an array of parameters as the second method parameter. Not only that, but it will insert them in the correct order within the URI. Let's take a look at this in action.

Example 18: URL to a named route with parameters.

```
<?php

// app/Http/routes.php

Route::get('the/{first}/avenger/{second}', [
    'as' => 'ironman',
    function($first, $second) {
        return "Tony Stark, the {$first} avenger {$second}.";
    }
]);

Route::get('example', function () {
    return URL::route('ironman', ['best', 'ever']);
});
```

If we visit the following URL...

Example 19: Hit this URL.

```
http://homestead.app/example
```

...Laravel will fill in the blanks in the correct order, with the parameters we have provided. The following URL is displayed a response.

Example 20: Output.

```
http://homestead.app/the/best/avenger/ever
```

There's one final routing method of this type that you need to know, and that's how to route to controller actions. In fact, this one should be pretty simple since it follows the same pattern as the `URL::route()` method. Let's take a look at an example.

Example 21: URL to a controller and action pair.

```php
<?php

// app/Http/routes.php

// Our Controller.
class Stark extends Controller
{
    public function tony()
    {
        return 'You can count on me, to pleasure myself.';
    }
}

// Route to the Stark controller.
Route::get('i/am/iron/man', 'Stark@tony');

Route::get('example', function () {
    return URL::action('Stark@tony');
});
```

In this example, we create a new controller called 'Stark' with a 'tony()' action. We also create a new route for the controller action. Next, we create an example route which returns the value of the `URL::action()` method. The first parameter of this method is the Class and action combination that we wish to retrieve the URL for. The format for this parameter is identical to that which we use for routing to controllers.

If we visit the `/example` URL, we receive the following response.

Example 22: Output.

```
http://homestead.app/i/am/iron/man
```

Laravel has identified the URL for the controller action pair that we requested, and delivered it as a response. Just as with the other methods, we can supply an array of parameters as a second parameter to the `URL::action()` method. Let's see this in action.

Example 23: URL to an action with parameters.

```
<?php

// app/Http/routes.php

// Our Controller.
class Stark extends Controller
{
    public function tony($whatIsTony)
    {
        // ...
    }
}

// Route to the Stark controller.
Route::get('tony/the/{first}/genius', 'Stark@tony');

Route::get('example', function () {
    return URL::action('Stark@tony', ['narcissist']);
});
```

Just as in the last example, we supply an array with a single item as a parameter to the `URL::action()` method, and Laravel constructs the URL to the controller with the parameter in the correct location.

The URL that we receive looks like this.

Example 24: Output.

```
http://homestead.app/tony/the/narcissist/genius
```

You can use the `action()` global function in the place of `URL::action()`. Which do you think looks better?

Well, that's it for route URL generation. I'm sorry if that got a bit repetitive, but hopefully, it will make for a good reference chapter.

Asset URLs

URLs to assets such as images, CSS files, and JavaScript need to be handled a little differently. Most of you will be using pretty URLs with Laravel. This is the act of rewriting the URL to remove the `index.php` front controller, and making our URLs more SEO friendly.

However, in some situations, you may not wish to use pretty URLs. However, if you were to try to link to an asset using the helpers mentioned in the previous subchapter, then the `index.php` portion of the URL would be included, and the asset links would break.

Even with pretty URLs, we don't want to link to our assets using relative URLs because our routing segments will be confused for a folder structure.

As always, Laravel, and Taylor are one step ahead of us. Helpers are provided to generate absolute URLs to our assets. Let's take a look at some of these helpers.

First, we have the `URL::asset()` method, let's take a look at it in action. The first parameter to the method is the relative path to the asset.

Example 25: URL to an asset.

```
1  <?php
2
3  // app/Http/routes.php
4
5  Route::get('example', function () {
6      return URL::asset('img/logo.png');
7  });
```

Now, if we visit the URL `/example` then we are greeted with the following response.

Example 26: Output.

```
1  http://homestead.app/img/logo.png
```

Laravel has created an absolute asset path for us.

Yep, that's right. You guessed it! Simply use the `asset()` function instead if you prefer these little shortcuts!

If we want to use a secure HTTPS protocol to reference our assets, then we can pass `true` as a second parameter to the `URL::asset()` method, like this:

Example 27: Secure asset URLs.

```
<?php

// app/Http/routes.php

Route::get('example', function() {
    return URL::asset('img/logo.png', true);
});
```

We now receive the following response from the `/example` URL.

Example 28: Output.

```
https://homestead.app/img/logo.png
```

Great! Laravel also provides a much more descriptive method of generating secure asset URLs. Simply use the `URL::secureAsset()` method and pass the relative path to your asset.

Example 29: Secure asset method.

```
<?php

// app/Http/routes.php

Route::get('example', function () {
    return URL::secureAsset('img/logo.png');
});
```

The response from this route is the same as the previous method.

Example 30: Output.

```
https://homestead.app/img/logo.png
```

If you prefer it, go ahead and use `secure_asset()` instead.

17. Databases

I have to make a confession. I'm not a big fan of databases. I got bitten by one as a child, you see. Once bitten, twice shy. Okay. Okay, I'm kidding again. It's because a database killed my brother.

Once more, just kidding. I don't have any siblings. I don't know why I don't enjoy them. I suppose I like visual things. Pretty things. Fun things. Databases are simply big grids of data. Not pretty. The anti-fun.

Fortunately, we are spoiled by lovely ORMs that will allow us to access our database rows as class instances. In a later chapter, we'll take a closer look at Laravel's ORM named Eloquent. Eloquent is lovely. It makes working with databases a pleasurable experience, even for a grim database-hater like myself.

Well then, let's put my hates aside for a moment and talk about the concept of a database. Why do we need one? Maybe we don't?

Does our application need to store data that will be available in all future requests?

I just want to show static pages.

Then we don't need a database. But what happens when we do need to store data across multiple requests and display or use it on other routes of our application? Then we're going to need a data storage method, and you will be glad that you've read these next few chapters.

Abstraction

What databases can we use with Laravel five? Let's see if any of the following take your fancy.

- MySQL Community / Standard / Enterprise Server[1]
- SQLite[2]
- PostgreSQL[3]
- SQL Server[4]

[1] http://www.mysql.com/products/
[2] http://www.sqlite.org/
[3] http://www.postgresql.org/
[4] http://www.microsoft.com/en-us/sqlserver/default.aspx

As you can see, you have a great deal of choice when selecting a database platform. For this book, I will be using the MySQL Community Server Edition[5]. It's a great free platform and one of the most popular ones used for development. You'll also find it installed on Homestead by default, along with a few others.

You don't have to worry about using another database server, though. You see, Laravel provides an abstraction layer. It decouples the framework's database components from the RAW SQL, providing different queries for different types of databases. Simply put, you don't have to worry about the SQL syntax. Let Laravel take care of it.

Another advantage of using Laravel's database abstraction layer is security. In most situations, unless I indicate otherwise, you won't have to worry about escaping the values that you send to the database from Laravel. Laravel will escape these values for you in an effort to prevent various forms of injection attacks.

Let's weigh up the flexibility of Laravel's database abstraction layer for a moment. You can switch database servers whenever you like without having to change any of the database code you have written, and you won't have to worry about simple matters of security. That to me sounds like a great chunk of work has been removed from your projects. Escaping values is boilerplate. We don't need to do that. Let's let Laravel take care of it.

Now that we know that we wish to use a database let's learn how we can setup Laravel to use one. Don't worry; it's quite a simple process! First, let's take a look at the configuration options.

Configuration

All of Laravel's database configuration is contained in the file located at `config/database.php`. That's easy to remember, right? Let's take a trip through the file, and look at some of the configuration options available.

Example 01: PDO Fetch Style.

```
1  <?php
2  /*
3  |--------------------------------------------------------------------------
4  | PDO Fetch Style
5  |--------------------------------------------------------------------------
6  |
7  | By default, database results will be returned as instances of the PHP
8  | stdClass object; however, you may desire to retrieve records in an
9  | array format for simplicity. Here you can tweak the fetch style.
```

[5] http://www.mysql.com/products/community/

```
10  |
11  */
12
13  'fetch' => PDO::FETCH_CLASS,
```

When rows are returned from a query that one of Laravel's database components executes, they will be default be in the form of a PHP `stdClass` object. This means that you can access the data in their columns in a format similar to this.

Example 02: Standard class result objects.

```
1  <?php
2
3  echo $book->name;
4  echo $book->author;
```

However, if you wish to alter the format in which rows are returned you may simply change the `fetch` option of the database configuration to something more suitable. Let's alter the option to `PDO::FETCH_ASSOC` which will instead use an associative PHP array to store our rows. Now we can access our database rows in the following manner.

Example 03: Array results.

```
1  <?php
2
3  echo $book['name'];
4  echo $book['author'];
```

For a full list of PDO fetch modes take a look at the PHP PDO constants documentation page[6]. Look for the constants that start with `FETCH_`.

Next, let's take a look at the connections array. Here's how it looks in its default form.

[6] http://www.php.net/manual/en/pdo.constants.php

Example 04: Database connection configuration.

```
<?php

/*
|--------------------------------------------------------------------------
| Database Connections
|--------------------------------------------------------------------------
|
| Here are each of the database connections setup for your application.
| Of course, examples of configuring each database platform that is
| supported by Laravel is shown below to make development simple.
|
|
| All database work in Laravel is done through the PHP PDO facilities
| so make sure you have the driver for your particular database of
| choice installed on your machine before you begin development.
|
*/

'connections' => [

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

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

    'pgsql' => [
        'driver'   => 'pgsql',
        'host'     => env('DB_HOST', 'localhost'),
```

```
        'database' => env('DB_DATABASE', 'forge'),
        'username' => env('DB_USERNAME', 'forge'),
        'password' => env('DB_PASSWORD', ''),
        'charset'  => 'utf8',
        'prefix'   => '',
        'schema'   => 'public',
    ],

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

],
```

Woah! That's a huge list of default connections. This makes it a lot easier to get started. Looking at the above array, you might think that we have a different index for each type of database. However, if you look more closely, you will notice that each nested array has a `driver` that can be used to specify the type of database. This means that we could easily have an array of different MySQL database connections, like this:

Example 05: Multiple MySQL connections.

```
<?php

'connections' => array(

    'mysql' => array(
        'driver'    => 'mysql',
        'host'      => 'localhost',
        'database'  => 'database',
        'username'  => 'root',
        'password'  => '',
        'charset'   => 'utf8',
        'collation' => 'utf8_unicode_ci',
        'prefix'    => '',
    ),

```

```
16      'mysql_2' => array(
17          'driver'    => 'mysql',
18          'host'      => 'localhost',
19          'database'  => 'database2',
20          'username'  => 'root',
21          'password'  => '',
22          'charset'   => 'utf8',
23          'collation' => 'utf8_unicode_ci',
24          'prefix'    => '',
25      ),
26
27      'mysql_3' => array(
28          'driver'    => 'mysql',
29          'host'      => 'localhost',
30          'database'  => 'database3',
31          'username'  => 'root',
32          'password'  => '',
33          'charset'   => 'utf8',
34          'collation' => 'utf8_unicode_ci',
35          'prefix'    => '',
36      ),
37
38  ),
```

By having a number of different database connections, we can switch databases at will. This way our application doesn't only have to have a single database. Very flexible. I think you will agree.

The first index of the connections array is simply a nickname given to the connection that we can supply when we need to perform an action on a specific database. You can call your databases anything you like!

Let's get to the meat and potatoes. We will take a closer look at an individual connection array. Here's an example once more.

Example 06: My connection.

```
'my_connection' => array(
    'driver'    => 'mysql',
    'host'      => 'localhost',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
),
```

The driver option can be used to specify the type of database that we intend to connect to.

Example 07: Database driver.

```
'driver'    => 'mysql',
```

Here are the possible values.

- `mysql` - MySQL
- `sqlite` - SQLite
- `pgsql` - PostgreSQL
- `sqlsrv` - SQL Server

Next, up we have the `host` index, which can be used to specify the network location of the machine that hosts the database server.

Example 08: Database host.

```
'host'      => 'localhost',
```

You can provide either an IP address (`168.122.122.5`) or a host name (`database.example.com`). In the local development environment, you will be primarily using `127.0.0.1` or `localhost` to refer to the current machine.

SQLite Databases

SQLite databases are stored at a location on disk and thus do not have a host entry. For this reason, you can simply omit this index from an SQLite database connection block.

The next index of the connection array is the `database` option.

Example 09: Database name.

```
'database'  => 'database_name',
```

It is a string value used to identify the name of the database which the connection is due to act upon. In the case of an SQLite database, it is used to specify the file that is used to store the database. For example:

Example 10: Database file path.

```
'database' => __DIR__.'/path/to/database.sqlite',
```

The `username` and `password` indexes can be used to provide access credentials for your database connection.

Example 11: Database access credentials.

```
'username'  => 'dayle',
'password'  => 'emma_w4tson_is_hot',
```

SQLite Databases

Once again, SQLite databases are a bit different here. They don't require credentials. You can omit these indexes from an SQLite connection block.

The next configuration index is `charset`, it can be used to specify the default character set for a database connection.

Example 12: Database character set.

```
'charset'   => 'utf8',
```

SQLite Databases

You guessed it! The SQLite database doesn't support this option. Just leave this index out of the connection array.

You can set the default database collation using the `collation` index.

Example 13: Database collation.

```
'collation' => 'utf8_unicode_ci',
```

SQLite Databases

Once again, SQLite chooses to be a unique snowflake. You don't need to provide the character set or collation index for it.

Finally, we have the `prefix` option, which can be used to add a common prefix to your database tables.

Example 14: Database table prefix.

```
'prefix'    => '',
```

Preparing

If you want to work through the examples in the next few chapters, then you are going to want to setup a working database connection. Go ahead and download a database platform and install it.

Next, you'll need to create a connection array, and fill in all the required parameters. Where you see `env('SOMETHING_HERE')` you can set `SOMETHING_HERE` to the value that you want within your `.env` file in the root of the project. It's always best to place your configuration variables in this file, instead of directly in the config files.

You're almost there. We simply need to tell Laravel, which database connection to use by default. Go ahead and take another look at the `config/database.php` file.

Example 15: Default database connection.

```
/*
|--------------------------------------------------------------------------
| Default Database Connection Name
|--------------------------------------------------------------------------
|
| Here you may specify which of the database connections below you wish
| to use as your default connection for all database work. Of course
| you may use many connections at once using the Database library.
|
*/

'default' => env('DB_CONNECTION', 'mysql'),
```

We need to set the `DB_CONNECTION` parameter in `.env` to the default connection that we want to use.

Well, I know you are excited by databases. You strange, strange person you! Let's not waste any more time. Flip the page and let's examine the schema builder.

18. Schema Builder

Right. You have decided that you want to store things in the database. The database isn't exactly a simple key-value store, though. Within the database, our data can have structure. It can consist of different types, and have relationships. Sexy, wonderful relationships.

To store our structured data, we must first define the structure. This isn't a book about SQL, so I hope by now that you will understand the concept of a database table and its columns. In this chapter, we are going to take a look at the `Schema` component that we can use to define the structure of our tables. We aren't going to be storing any data in this chapter, so make sure that you are in the mindset of structure and not content.

In the next chapter, you will learn about an ideal location to start building your database structure, but I like to begin by describing each feature in isolation. For now, we will be writing our schema-building code within routed closures. Well, let's not waste any more time. Let's take a quick look at the query builder.

Creating Tables

To create a table, we must make use of the `create()` method of the `Schema` facade. Here's an example.

Example 01: The Schema component.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    Schema::create('users', function($table) {
        // Let's not get carried away.
    });
});
```

The `Schema::create()` method accepts two parameters. The first is the name of the table that we wish to create. In this case, we are creating a table named 'users'. If

the table we are creating will be used to store data representing a type of object, we should name the table in lowercase as the plural of the object.

Database columns and tables are commonly named using snake-casing. This is where spaces are replaced with underscores (_) and all characters are lowercase.

The second parameter to the method is a Closure with a single parameter. In the above example, I have called the parameter `$table`, but you can call it whatever you want! The `$table` parameter can be used to build the table structure.

Let's add an auto-incrementing primary key to our table. This way our table rows can be identified by a unique index.

Example 02: Auto-incremental column.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    Schema::create('users', function($table) {
        $table->increments('id');
    });
});
```

The `increments()` method is available on our `$table` instance to create a new auto incremental column. An auto incremental column will automatically be populated with an integer value that increments as each row is added. It will start at one. This column will also be the primary key for the table. The first parameter to the increments method is the name of the column that will be created. That was simple, right?

Let's go ahead and add some more columns to this table.

Example 03: Additional columns.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    Schema::create('users', function($table) {
        $table->increments('id');
        $table->string('username', 32);
        $table->string('email', 320);
        $table->string('password', 60);
```

```
11          $table->timestamps();
12      });
13  });
```

Great! We have a blueprint to create the structure of our 'users' table. Don't worry about each of the individual columns right now. We will cover them in detail in the next section. First, let's build this table by visiting the / URI to fire our routed Closure. We aren't expecting any output.

Let's take a look at the database structure that has been created for us. I don't know which database you have chosen to use, but I'm going to be using MySQL for this book, so I will take a look at the database using the mySQL command line interface. Feel free to use whatever software you feel most comfortable with.

Example 04: Describe the users table.

```
1  mysql> use myapp;
2  Database changed
3
4  mysql> describe users;
5  +------------+------------------+-----+----------------+
6  | Field      | Type             | Key | Extra          |
7  +------------+------------------+-----+----------------+
8  | id         | int(10) unsigned | PRI | auto_increment |
9  | username   | varchar(32)      |     |                |
10 | email      | varchar(320)     |     |                |
11 | password   | varchar(60)      |     |                |
12 | created_at | timestamp        |     |                |
13 | updated_at | timestamp        |     |                |
14 +------------+------------------+-----+----------------+
15 6 rows in set (0.00 sec)
```

I've simplified the describe table a little to fit in the book's formatting restrictions, but I hope that you get the picture. Our user table structure has been built using the blueprint that we created with our `$table` object.

You must be wondering what methods and columns we have available on the `$table` object? Well then, let's take a look!

Column Types

We are going to examine the methods that are available on the `$table` blueprint object. I'm going to leave the routed closure out of these examples to simplify things. You will have to use your imagination! Let's get started.

---

increments

The increments method will add an auto incremental integer primary key to the table. This is a very useful method for building the structure for Eloquent ORM models, which we will learn about in a later chapter.

Example 05: Increments.

```
<?php

Schema::create('example', function($table) {
    $table->increments('id');
});
```

The first and only parameter for the `increments()` method is the name of the column to create. Here's the resulting table structure:

Example 06: Describe output.

```
+-------+------------------+-----+----------------+
| Field | Type             | Key | Extra          |
+-------+------------------+-----+----------------+
| id    | int(10) unsigned | PRI | auto_increment |
+-------+------------------+-----+----------------+
```

---

bigIncrements

Oh. So the increments method wasn't big enough for you? Well, the `bigIncrements()` method will create a big integer, rather than a regular one.

Example 07: Big Increments.

```
<?php

Schema::create('example', function($table) {
    $table->bigIncrements('id');
});
```

Just like the `increments()` method, the `bigIncrements()` method will accept a single string parameter as the column name.

Example 08: Describe output.

```
+-------+---------------------+-----+----------------+
| Field | Type                | Key | Extra          |
+-------+---------------------+-----+----------------+
| id    | bigint(20) unsigned | PRI | auto_increment |
+-------+---------------------+-----+----------------+
```

---

string

The `string()` method can be used to create `varchar` columns, which are useful for storing short string values.

Example 09: String.

```php
<?php

Schema::create('example', function($table) {
    $table->string('nickname', 128);
});
```

The first parameter to `string()` method is the name of the column to create. However, there is an optional second parameter to define the length of the string in characters. The default value is 255.

Example 10: Describe output.

```
+----------+--------------+
| Field    | Type         |
+----------+--------------+
| nickname | varchar(255) |
+----------+--------------+
```

---

text

The `text()` method can be used to store large amounts of text that will not fit into a `varchar` column type. For example, this column type could be used to contain the body text of a blog post.

Example 11: Text.

```
<?php

Schema::create('example', function($table) {
    $table->text('body');
});
```

The `text()` method accepts a single parameter. The name of the column that will be created.

Example 12: Describe output.

```
+-------+------+
| Field | Type |
+-------+------+
| body  | text |
+-------+------+
```

---

integer

The integer column type can be used to store integer values. Are you surprised? Well, I can't think of a way to make it any more interesting! I suppose I could mention how integer values are useful when referencing the auto incremented id of another table. This is called a foreign key. We can use this method to create relationships between tables.

Example 13: Integer.

```
<?php

Schema::create('example', function($table) {
    $table->integer('shoe_size');
});
```

The first parameter to the `integer()` method is the name of the column. The second parameter is a boolean value that can be used to define whether or not the column should be auto incremental. The third parameter is used to define whether or not the integer is unsigned. A signed integer can be positive or negative. However, if you define an integer as unsigned, then it can only be positive. Signed integers can contain a range of integers from −2,147,483,648 to 2,147,483,647, where an unsigned integer can hold value from 0 to 4,294,967,295.

Example 14: Describe output.

```
+-----------+---------+
| Field     | Type    |
+-----------+---------+
| shoe_size | int(11) |
+-----------+---------+
```

---

bigInteger

Big integer values work exactly like normal integers; only they have a much larger range. A signed integer has a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Unsigned integers have a range of 0 to 18,446,744,073,709,551,615. An integer of this size is normally used to store my waist size in inches.

Example 15: Big integer.

```php
<?php

Schema::create('example', function($table) {
    $table->bigInteger('waist_size');
});
```

The method signature for all of the integer variants is the same as the `integer()` method, so I won't waste any time by repeating it! If you have forgotten already, then maybe you should take a look at the 'integer' section again?

Example 16: Describe output.

```
+------------+------------+
| Field      | Type       |
+------------+------------+
| waist_size | bigint(20) |
+------------+------------+
```

---

mediumInteger

This column is another type of integer. Let's see if we can get through these column types a little faster shall we? I'm just going to specify the column value ranges from now on. The signed range is -8388608 to 8388607. The unsigned range is 0 to 16777215.

Example 17: Medium integer.

```
<?php

Schema::create('example', function($table) {
    $table->mediumInteger('size');
});
```

The method signature is identical to that of the `integer()` method.

Example 18: Describe output.

```
+-------+--------------+
| Field | Type         |
+-------+--------------+
| size  | mediumint(9) |
+-------+--------------+
```

---

tinyInteger

This is another integer type column. The signed range is -128 to 127. The unsigned range is 0 to 255.

Example 19: Tiny integer.

```
<?php

Schema::create('example', function($table) {
    $table->tinyInteger('size');
});
```

The method signature is identical to that of the `integer()` method.

Example 20: Describe output.

```
+-------+------------+
| Field | Type       |
+-------+------------+
| size  | tinyint(1) |
+-------+------------+
```

---

smallInteger

This is another integer type column. The signed range is -32768 to 32767. The unsigned range is 0 to 65535.

Example 21: Small integer.

```php
<?php

Schema::create('example', function($table) {
    $table->smallInteger('size');
});
```

The method signature is identical to that of the `integer()` method.

Example 22: Describe output.

```
+-------+-------------+
| Field | Type        |
+-------+-------------+
| size  | smallint(6) |
+-------+-------------+
```

---

float

Float column types are used to store floating point numbers. Here's how they can be defined.

Example 23: Float.

```
<?php

Schema::create('example', function($table) {
    $table->float('size');
});
```

The first value parameter is the name used to identify the column. The optional second and third integer parameters can be used to specify the length of the value, and the number of decimal places to use to represent the value. The defaults for these parameters are eight and two respectively.

Example 24: Describe output.

```
+-------+------------+
| Field | Type       |
+-------+------------+
| size  | float(8,2) |
+-------+------------+
```

---

decimal

The `decimal()` method is used to store, wait for it, decimal values! It looks very similar to the `float()` method.

Example 25: Decimal.

```
<?php

Schema::create('example', function($table) {
    $table->decimal('size');
});
```

This method accepts a column name as the first parameter, and two optional parameters to represent the length and number of decimal places that should be used to define the column. The defaults for the optional parameters are again, 8 and 2.

Example 26: Describe output.

```
+-------+--------------+
| Field | Type         |
+-------+--------------+
| size  | decimal(8,2) |
+-------+--------------+
```

---

boolean

Not all values consist of large ranges of digits and characters. Some only have two states, `true` or `false`, `1` or `0`. Boolean column types can be used to represent these values.

Example 27: Boolean.

```
<?php

Schema::create('example', function($table) {
    $table->boolean('hot');
});
```

The only parameter for the boolean method is the name given to the column it creates.

Example 28: Describe output.

```
+-------+------------+
| Field | Type       |
+-------+------------+
| hot   | tinyint(1) |
+-------+------------+
```

The `tinyint` in the above example is not a typo. Tiny integers are used to represent boolean values as `1` or `0`. I'd also like to mention at this point that I almost burnt the kitchen down while being distracted writing this section. I thought it might be interesting to know about the perilous life of a technical writer! No? Fine. Let's carry on with the column descriptions.

---

enum

The enumerated type will store strings that are contained within a list of allowed values. Here's an example.

Example 29: Enum.

```
<?php

Schema::create('example', function($table) {
    $table->enum('who', ['Walt', 'Jesse', 'Saul']);
});
```

The first parameter is the name of the column that will be created. The second parameter is an array of values that are permitted for this enumerated type.

Example 30: Describe output.

```
+-------+-----------------------------+------+
| Field | Type                        | Null |
+-------+-----------------------------+------+
| who   | enum('Walt','Jesse','Saul') | NO   |
+-------+-----------------------------+------+
```

---

date

As the name suggests, the `date()` method can be used to create columns that store dates.

Example 31: Date.

```
<?php

Schema::create('example', function($table) {
    $table->date('when');
});
```

The first and only parameter is used to specify the name of the column that will be created.

Example 32: Describe output.

```
+-------+------+
| Field | Type |
+-------+------+
| when  | date |
+-------+------+
```

---

dateTime

The `dateTime()` method will not only store a date but also the time. No kidding, it really will. I know, I know, a lot of these methods are similar. Trust me; this will make a great reference chapter!

Example 33: Datetime.

```php
<?php

Schema::create('example', function($table) {
    $table->dateTime('when');
});
```

Once again, the name of the column to be created is the only parameter.

Example 34: Describe output.

```
+-------+----------+
| Field | Type     |
+-------+----------+
| when  | datetime |
+-------+----------+
```

---

time

Don't want the date included with your times? Fine! Just use the `time()` method instead.

Example 35: Time.

```
<?php

Schema::create('example', function($table) {
    $table->time('when');
});
```

Once again, the first and only parameter to the `time()` method is the name of the column being created.

Example 36: Describe output.

```
+-------+------+
| Field | Type |
+-------+------+
| when  | time |
+-------+------+
```

---

timestamp

The `timestamp()` method can be used to store a date and time in the `TIMESTAMP` format. Surprised? No? Oh. Well, let's take a look at how it works.

Example 37: Timestamp.

```
<?php

Schema::create('example', function($table) {
    $table->timestamp('when');
});
```

The first and only value is the name of the database column that will be created.

Example 38: Describe output.

```
+-------+-----------+---------------------+
| Field | Type      | Default             |
+-------+-----------+---------------------+
| when  | timestamp | 0000-00-00 00:00:00 |
+-------+-----------+---------------------+
```

---

binary

The `binary()` method can be used to create columns that will store binary data. These types of columns can be useful for storing binary files such as images.

Example 39: Binary.

```
<?php

Schema::create('example', function($table) {
    $table->binary('image');
});
```

The only parameter to the binary method is the name of the column that is being created.

Example 40: Describe output.

```
+-------+------+
| Field | Type |
+-------+------+
| image | blob |
+-------+------+
```

Special Column Types

Laravel includes several special column types that have varied uses. Let's take a look at them. First up, we have the `timestamps()` method.

The `timestamps()` method can be used to add two 'TIMESTAMP' columns to the table. The `created_at` and `updated_at` columns can be used to indicate when a row was created and updated. In a later chapter, we will learn how Laravel's own Eloquent ORM can be told to automatically update these columns when an ORM instance is created or updated. Let's have a look at how the `timestamps()` method is used.

Example 41: Timestamps.

```
<?php

Schema::create('example', function($table) {
    $table->timestamps();
});
```

The `timestamps()` method doesn't accept any parameters. Here's the table structure that is created.

Example 42: Describe output.

```
+------------+-----------+---------------------+
| Field      | Type      | Default             |
+------------+-----------+---------------------+
| created_at | timestamp | 0000-00-00 00:00:00 |
| updated_at | timestamp | 0000-00-00 00:00:00 |
+------------+-----------+---------------------+
```

Next, we have the `softDeletes()` method. Occasionally, you will want to mark table row as deleted without actually deleting the data contained within. This is useful if you may wish to restore the data in the future. With the `softDeletes()` method you can place an indicator column on the row to show that the row has been deleted. The column that is created will be called `deleted_at` and will be of type 'TIMESTAMP.' Once again, Laravel's Eloquent ORM will be able to update this column without deleting the row when you use the delete method on an ORM instance. Here's how we can add the `deleted_at` column to our table.

Example 43: Soft deletes.

```
<?php

Schema::create('example', function($table) {
    $table->softDeletes();
});
```

The `softDeletes()` method does not accept any parameters. Here's the resulting table.

Example 44: Describe output.

```
+------------+-----------+------+
| Field      | Type      | Null |
+------------+-----------+------+
| deleted_at | timestamp | YES  |
+------------+-----------+------+
```

Column Modifiers

Column modifiers can be used to add extra constraints or properties to the columns that we create with the `create()` method. For example, we used the `increments()` method to create a table index column that was both auto incremental and a primary key. That's a handy shortcut, but let's take a look at how we can turn another column into a primary key using column modifiers.

First, we will make a new column and declare that it must contain unique values.

Example 45: Unique.

```php
<?php

Schema::create('example', function($table) {
    $table->string('username')->unique();
});
```

By chaining the `unique()` method on to our column creation method, we have told the database that duplicate values will not be allowed for this column. Our primary key should be used to identify individual rows, so we don't want to have duplicate values, do we!

Let's make the 'username' column the table's primary key.

Example 46: Primary.

```php
<?php

Schema::create('example', function($table) {
    $table->string('username')->unique();
    $table->primary('username');
});
```

We can mark any column as a primary key using the `primary()` method. The only parameter to this method is a string representing the name of the column to mark as the key. Let's describe the table we have just created.

Example 47: Describe output.

```
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| username | varchar(255) | NO   | PRI | NULL    |       |
+----------+--------------+------+-----+---------+-------+
```

Great! We have a new primary key.

Here's a neat trick. Both the `primary()` key, and the `unique()` methods can act on their own, or fluently chained to an existing value. This means that the above example could also be written like this:

Example 48: Fluent methods.

```php
<?php

Schema::create('example', function($table) {
    $table->string('username')->unique()->primary();
});
```

The above example shows how the column modifiers can be chained to an existing column definition. Alternatively, the column modifiers can be used in isolation by providing a column name as a parameter.

Example 49: Methods in isolation.

```php
<?php

Schema::create('example', function($table) {
    $table->string('username');
    $table->unique('username');
    $table->primary('username');
});
```

If you aren't satisfied with a single primary key for your table, then you can use multiple composite keys by providing an array of column names to the `primary()` method that we used in the previous example. Let's take a look.

Example 50: Composite primary keys.

```
<?php

Schema::create('example', function($table) {
    $table->integer('id');
    $table->string('username');
    $table->string('email');
    $table->primary(['id', 'username', 'email']);
});
```

Now our three new columns will act as a composite key, whereby any combination of the values contained in the columns will be a unique reference to an individual role. Let's have a look at the output from 'describe'.

Example 51: Describe output.

```
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| id       | int(11)      | NO   | PRI | NULL    |       |
| username | varchar(255) | NO   | PRI | NULL    |       |
| email    | varchar(255) | NO   | PRI | NULL    |       |
+----------+--------------+------+-----+---------+-------+
```

We can speed up our queries by marking columns that are used to lookup information as indexes. We can use the `index()` method to mark a column as an index. It can be used fluently. Like this:

Example 52: Index.

```
<?php

Schema::create('example', function($table) {
    $table->integer('age')->index();
});
```

Or in isolation. Like this:

Example 53: Indices in isolation.

```
<?php

Schema::create('example', function($table) {
    $table->integer('age');
    $table->index('age');
});
```

Either way, the result will be the same. The column will be marked as an index.

Example 54: Describe output.

```
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| age   | int(11) | NO   | MUL | NULL    |       |
+-------+---------+------+-----+---------+-------+
```

We can also pass an array of column names to the `index()` method to mark multiple columns as indexes. Here's an example.

Example 55: Multiple indices.

```
<?php

Schema::create('example', function($table) {
    $table->integer('age');
    $table->integer('weight');
    $table->index(['age', 'weight']);
});
```

Here's the resulting table structure.

Example 56: Describe output.

```
  +--------+---------+------+-----+---------+-------+
| Field  | Type    | Null | Key | Default | Extra |
+--------+---------+------+-----+---------+-------+
| age    | int(11) | NO   | MUL | NULL    |       |
| weight | int(11) | NO   |     | NULL    |       |
+--------+---------+------+-----+---------+-------+
```

Sometimes, we want to set a constraint on a column to state whether or not it can contain a null value. We can set a column to nullable using the `nullable()` method. It can be used as part of a method chain, like this:

Example 57: Nullable.

```
<?php

Schema::create('example', function($table) {
    $table->string('name')->nullable();
});
```

Here's the resulting table structure.

Example 58: Describe output.

```
+-------+-------------+------+
| Field | Type        | Null |
+-------+-------------+------+
| name  | varchar(255) | YES  |
+-------+-------------+------+
```

As you can see, the column can now contain a null value. If we **don't** want the column to allow a null value, we can pass boolean `false` as the first parameter to the `nullable()` chained method, like this:

Example 59: Not nullable.

```
<?php

Schema::create('example', function ($table) {
    $table->string('name')->nullable(false);
});
```

Let's take another look at the resulting table structure.

Example 60: Describe output.

```
+-------+--------------+------+
| Field | Type         | Null |
+-------+--------------+------+
| name  | varchar(255) | NO   |
+-------+--------------+------+
```

As you can see, the 'name' column can no longer contain a null value.

If we wish for our columns to contain a default value when a new row is created, we can provide the default value by chaining the `default()` method onto the new column definition. Here's an example.

Example 61: Defaults.

```
<?php

Schema::create('example', function($table) {
    $table->string('name')->default('John Doe');
});
```

The first and only parameter to the `default()` method is the intended default value for the column. Let's take a look at the resulting table structure.

Example 62: Describe output.

```
+-------+--------------+------+-----+----------+
| Field | Type         | Null | Key | Default  |
+-------+--------------+------+-----+----------+
| name  | varchar(255) | NO   |     | John Doe |
+-------+--------------+------+-----+----------+
```

If we don't provide a value for the 'name' column when creating a new row, then it will default to 'John Doe'.

We have one final column modifier to look at. This one isn't needed, but it's a nice little shortcut. Do you remember creating integer columns in the previous section? We used a boolean parameter to specify whether or not an integer was signed and could contain a negative value. Well, we can use the `unsigned()` chained method on an integer column to specify that it may not contain negative numbers. Here's an example.

Example 63: Unsigned.

```
<?php

Schema::create('example', function($table) {
    $table->integer('age')->unsigned();
});
```

Here's the resulting table structure after using the `unsigned()` chained method.

Example 64: Describe output.

```
+-------+------------------+
| Field | Type             |
+-------+------------------+
| age   | int(10) unsigned |
+-------+------------------+
```

Whether you choose to use the boolean switch, or the `unsigned()` method, the choice is entirely yours.

Updating Tables

Once a table has been created, there's no way to change it.

> Are you sure, because the heading say...

I'm sure; there's absolutely no way.

> Hmm, but the heading says updating tables?

You just won't let it go will you? Fine. I was going to take a nap, but you have convinced me. You need to know about updating tables. Let's get started.

First of all, we can change the name of a table that we have already created quite easily using the `Schema::rename()` method. Let's take a look at an example.

Example 65: Rename a table.

```
<?php

// Create the users table.
Schema::create('users', function($table) {
    $table->increments('id');
});

// Rename the users table to idiots.
Schema::rename('users', 'idiots');
```

The first parameter of the `rename()` method is the name of the table that we wish to change. The second parameter to the method is the new name for the table.

If we want to alter the columns of an existing table, then we need to use the `Schema::table()` method. Let's take a closer look.

Example 66: Modify existing table.

```
<?php

Schema::table('example', function($table) {
    // Modify the $table...
});
```

The `table()` method is almost identical to the `create()` method we used earlier to create a table. The only difference is that it acts upon an existing table that we specify within the first parameter to the method. Once again, the second parameter contains a Closure with a parameter of a table builder instance.

We can use any of the column creation methods that we discovered in the previous section to add new columns to the existing table. Here's an example.

Example 67: Modify existing table.

```
<?php

Schema::create('example', function($table) {
    $table->increments('id');
});

Schema::table('example', function($table) {
    $table->string('name');
});
```

In the above example, we use the `Schema::create()` method to build the 'example' table with a primary key. Then we use the `Schema::table()` method to add a string column to the existing table.

Here's the result from `describe example;`:

Example 68: Describe output.

```
+-------+------------------+-----+----------------+
| Field | Type             | Key | Extra          |
+-------+------------------+-----+----------------+
| id    | int(10) unsigned | PRI | auto_increment |
| name  | varchar(255)     |     |                |
+-------+------------------+-----+----------------+
```

You can use any of the column creation methods that we learned about in the previous section to add additional columns to a table. I won't cover every creation method again because their signatures haven't changed. If you need a quick refresher course, then have another look at the 'Column Types' section.

If we decide that we no longer wish to have a column on our table, then we can use the `dropColumn()` method to remove it. Let's take a look at this in action.

Example 69: Dropping columns.

```
<?php

Schema::create('example', function($table) {
    $table->increments('id');
    $table->string('name');
});

Schema::table('example', function($table) {
    $table->dropColumn('name');
});
```

In the above example, we create the 'example' table with two columns. Then we use the `dropColumn()` method to remove the 'name' column from the table. The `dropColumn()` method will accept a string parameter, which is the name of the column that we wish to remove.

Here is what our 'example' table will look like after the above code has been executed.

Example 70: Describe output.

```
+-------+------------------+------+-----+----------------+
| Field | Type             | Null | Key | Extra          |
+-------+------------------+------+-----+----------------+
| id    | int(10) unsigned | NO   | PRI | auto_increment |
+-------+------------------+------+-----+----------------+
```

As you can see, the 'name' column was removed successfully.

If we wish to remove more than one column at once, we can either provide an array of column names as the first parameter to the `dropColumn()` method.

Example 71: Dropping multiple columns.

```
<?php

Schema::table('example', function($table) {
    $table->dropColumn(['name', 'age']);
});
```

Or we can simply provide multiple string parameters for column names.

Example 72: Dropping multiple columns.

```
<?php

Schema::table('example', function($table) {
    $table->dropColumn('name', 'age');
});
```

Feel free to use whichever method suits your style of coding.

We don't have to drop our columns, though. If we want to, we can simply rename them. Let's have a look at an example.

Example 73: Rename columns.

```
<?php

Schema::create('example', function($table) {
    $table->string('name');
});

Schema::table('example', function($table) {
    $table->renameColumn('name', 'nickname');
});
```

The `renameColumn()` method is used to change the name of a column. The first parameter to the method is the name of the column that we wish to rename, and the second parameter is the new name for the column. Here's the resulting table structure for the above example.

Example 74: Describe output.

```
+----------+--------------+
| Field    | Type         |
+----------+--------------+
| nickname | varchar(255) |
+----------+--------------+
```

Do you remember the primary keys that we constructed in the previous section? What happens if we no longer wish for those columns to be primary keys? Not a problem, we just remove the key. Here's an example.

Example 75: Remove primary keys.

```
<?php

Schema::create('example', function($table) {
    $table->string('name')->primary();
});

Schema::table('example', function($table) {
    $table->dropPrimary('name');
});
```

Using the `dropPrimary()` method, we supply the name of a column as a parameter. This column will have its primary key attribute removed. Here's how the table looks after the code has executed.

Example 76: Describe output.

```
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| name  | varchar(255) | NO   |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
```

As you can see, the name column is no longer a primary key. To remove some composite keys from a table, we can instead supply an array of column names as the first parameter of the `dropPrimary()` method. Here's an example.

Example 77: Remove multiple primary keys.

```
<?php

Schema::create('example', function($table) {
    $table->string('name');
    $table->string('email');
    $table->primary(['name', 'email']);
});

Schema::table('example', function($table) {
    $table->dropPrimary(['name', 'email']);
});
```

We can remove the `unique` attribute for a column by using the `dropUnique()` method. This method accepts a single parameter which consists of the table name, column name, and 'unique' separated by underscores. Here's an example of removing the `unique` attribute from a column.

Example 78: Remove unique.

```
<?php

Schema::create('example', function($table) {
    $table->string('name')->unique();
});

Schema::table('example', function($table) {
    $table->dropUnique('example_name_unique');
});
```

Once again, we can pass an array of column names in the same format to the `dropUnique()` method if we wish. Here's an example.

Example 79: Remove multiple uniques.

```
<?php

Schema::create('example', function($table) {
    $table->string('name')->unique();
    $table->string('email')->unique();
});

Schema::table('example', function($table) {
    $table->dropUnique(['example_name_unique', 'example_email_unique']);
});
```

Finally, we can drop an `index` attribute from a table column by using, wait for it, okay you guessed it. We can use the `dropIndex()` method. Simply provide the column name in the same format as we used with the `dropUnique()` method. That's the table name, column name, and 'index'. For example:

Example 80: Remove an index.

```
<?php

Schema::create('example', function($table) {
    $table->string('name')->index();
});

Schema::table('example', function($table) {
    $table->dropIndex('example_name_index');
});
```

Dropping Tables

To drop a table, simply cut off its legs.

Just kidding. We can drop a table using the `Schema::drop()` method, let's take a look at this method in action.

Example 81: Drop a table.

```
<?php

Schema::create('example', function($table) {
    $table->string('name');
});

Schema::drop('example');
```

To drop a table, we simply pass the name of the table as the first parameter of the `Schema::drop()` method. Let's try to describe the table to see if it exists.

Example 82: Describe output.

```
mysql> describe example;
ERROR 1146 (42S02): Table 'myapp.example' doesn't exist
```

Well, I guess it worked! It looks like the table is gone.

If we try to drop a table that doesn't exist, then we will get an error. We can avoid this by instead using the `dropIfExists()` method. As the name suggests, it will only drop a table that exists. Here's an example.

Example 83: Drop a table if it already exists.

```
<?php

Schema::create('example', function($table) {
    $table->string('name');
});

Schema::dropIfExists('example');
```

Just like the `drop()` method, the `dropIfExists()` method accepts a single parameter, the name of the table to drop.

Schema Tricks

Tricks? Maybe not. However, this section is used for methods that simply don't fit into the previous sections. Let's waste no time by taking a look at the first method.

We can use the `Schema::connection()` method to perform our schema changes on an alternative database or connection. Let's take a look at an example.

Example 84: Different connections.

```
<?php

Schema::connection('mysql')->create('example', function($table) {
    $table->increments('id');
});

Schema::connection('mysql')->table('example', function($table) {
    $table->string('name');
});
```

The `connection()` method can be placed before any of the `Schema` class methods to form a chain. The first parameter for the method is the name of the database connection which subsequent methods will act upon.

The `connection()` method can be very useful if you need to write an application that uses multiple databases.

Next up, we have a couple of methods that can be used to check the existence of columns and tables. Let's go ahead and jump right in with an example.

Example 85: Check if table exists.

```
<?php

if (Schema::hasTable('author')) {
    Schema::create('books', function($table) {
        $table->increments('id');
    });
}
```

We can use the `hasTable()` method to check for the existence of a table. The first parameter to the method is the name of the table that we wish to check. In the above example, we create the 'books' table only if the 'authors' table exists.

As you might have already guessed, we have a similar method to check for the existence of a column. Let's take a look at another example.

Example 86: Check if column exists.

```
<?php

if (Schema::hasColumn('example', 'id')) {
    Schema::table('example', function($table) {
        $table->string('name');
    });
}
```

We can use the `Schema::hasColumn()` method to check if a table has a column. The first parameter to the method is the table, and the second parameter is the name of the column that we want to look for. In the above example, a 'name' column will be added to the 'example' table.

If you happen to be a database genius, you might want to change the storage engine used by the table. Here's an example.

Example 87: Change database engine.

```php
<?php

Schema::create('example', function ($table) {
    $table->engine = 'InnoDB';
    $table->increments('id');
});
```

Simply change the value of the `engine` attribute on the table blueprint to the name of the storage engine that you wish to use. Here are some of the available storage engines for the MySQL database:

- MyISAM
- InnoDB
- IBMDM2I
- MERGE
- MEMORY
- EXAMPLE
- FEDERATED
- ARCHIVE
- CSV
- BLACKHOLE

For more information about these storage engines, please consult the MySQL documentation[1] for the topic.

On MySQL databases, you can reorder the columns of a table by using the `after()` method. Here's an example.

Example 88: Ordering columns.

```php
<?php

Schema::create('example', function ($table) {
    $table->string('name')->after('id');
    $table->increments('id');
});
```

[1] https://dev.mysql.com/doc/refman/5.7/en/storage-engines.html

Simply chain the `after()` method onto the column that you wish to reposition. The only parameter to the method is the name of the column that you wish for the new column to follow. Feel free to use this method, although I would recommend simply building your tables in the intended order. This will look much clearer. This method is best used for modifying existing tables.

That's all that I have about building database schemas. Why don't we learn about a more suitable place to build our schemas. Let's move on to the migrations chapter.

19. Migrations

We have a rather impressive system at Dayle Manor. A system that will allow all of the days tasks to be completed without any fuss by my army of red panda butlers. Let me share it with you. Here's a list of jobs for my butlers. You could give them a hand you like?

- **9:00 AM** - Wash and dress Dayle.
- **10:00 AM** - Cook and grill some rare and exotic meats for breakfast.
- **12:00 PM** - (Lunch) The pandas will climb a tree and sleep for a while.
- **02:00 PM** - Polish the Apple hardware collection.
- **04:00 PM** - Prepare the writing throne for the next Code Smart chapter.
- **09:00 PM** - Drag sleeping Dayle from the writing throne, and tuck him into bed.

So that's my list for the red pandas. They have quite a busy day, and I don't know what I would do without them. The problem is that the list has a very specific order. We don't want the pandas to tuck me into bed before I have visited the writing throne. Otherwise, you won't get a new chapter. Also, there wouldn't be a lot of point in doing these tasks twice. The pandas need to ensure that they are done once, sequentially.

The pandas are so smart that it was them that came up to the solution to the problem all on their own. I gave them the original list on a notepad with a pencil, and well, they get rather excited when you give them gifts. There was a lot of playful rolling. They decided that they would write their own list. Double the fun, right?

The pandas decided to write a secondary list. Whenever they completed a task, which of course were completed in time order from the first list, they would write the time and name of the task on the second list. This way the same task would never be repeated.

Not a bad idea, I have to admit. Fortunately, some clever chaps invented a similar idea for databases. Let's take a look at migrations.

Basic Concept

When building your database, you could create its structure by hand. Type up some nifty SQL to describe your columns. But what happens when you accidentally drop the database? What if you are working as a team? You don't want to have to pass your SQL dumps around the team all the time to keep the database synchronized.

That's where migrations come in handy. Migrations are some PHP scripts that are used to change the structure or content of your database. Migrations are time stamped so that they are always executed in the correct order.

Laravel keeps a record of which migrations have already been executed within another table on your default database connection. This way it will only ever run any additional migrations that have been added.

Using migrations, you and your team will always have the same database structure, in a consistent and stable state. You know what? Actions speak louder than words. Let's create a new migration and start the learning process.

Creating Migrations

To create a migration we need to use the Artisan command-line interface. Go ahead. Open a terminal window and navigate to the project folder using whichever shell you call home.

We learned about schema building in the previous chapter, and I told you there was a better place to use the schema. I was, of course, talking about migrations. Let's recreate the schema that we used to create the users table. We will start by using Artisan to build a `create_users_table` migration.

Example 01: Create a new migration.

```
$ php artisan make:migration create_users_table
Created Migration: 2016_01_30_124846_create_users_table
```

We call the Artisan `make:migration` command and provide a name for our new migration. Laravel has now generated a new migration template within the `database/migrations` directory. The template will be located in a file named after the parameter that you supplied to the `make:migration` command with an attached timestamp. In this instance, our template is located within the following file.

Example 02: File location.

```
database/migrations/2016_01_30_124846_create_users_table.php
```

Let's open up the file in our text editor and see what we have.

Example 03: The migration file.

```php
<?php

use Illuminate\Database\Migrations\Migration;

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        //
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        //
    }
}
```

Here we have our migration class. It's important that you always use the Artisan command to generate migrations. You don't want to risk breaking the timestamps, and thus the history of your database structure. Be a good reader and use the command.

Within the migration class we have two public methods, `up()` and `down()`. Let's imagine a line between these two methods, or write one within a comment if you didn't learn about imagination from our friend Barney.

Either side of the line, a direct opposite must happen. Whatever you do in the `up()` method, you must undo within the `down()` method. Migrations are bi-directional. We can run a migration to update the structure or content of our database, but we can also undo that migration to revert it to its original state.

First, let's fill in the `up()` method.

Example 04: The up method.

```php
<?php

/**
 * Run the migrations.
 *
 * @return void
 */
public function up()
{
    Schema::create('users', function($table) {
        $table->increments('id');
        $table->string('name', 128);
        $table->string('email');
        $table->string('password', 60);
        $table->timestamps();
    });
}
```

Hopefully, there's nothing confusing within this schema construction snippet. If you don't understand any of it, then take another look at the 'Schema Builder' chapter.

We know that what goes up, must come down. For that reason, let's tackle the `down()` method and create the inverse of the structure change within the `up()` method.

Here we go.

Example 05: The down method.

```php
<?php

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

Alright, alright. I suppose it's not the direct opposite. I'm guessing you wanted to drop all of the columns individually, and then the table. They would both end in the same result. The `users` table would be dropped. So why not do it in one line?

Before we continue to the next section, let's take a look at a few tricks that relate to creating migrations. Using the `--create` and `--table` switches on the `make:migration` command we can automatically create a stub for the creation or updating of a new table.

We simply run the following command.

Example 06: Make migration with table stub.

```
php artisan make:migration create_users_table --create="users"
```

Then we receive the following migration stub.

Example 07: Migration file with stub.

```
<?php

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

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

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

Great! That shortcut has saved us a heap of time. You will notice that as well as adding the `Schema::create()` and `Schema::drop()` methods for our new table, Laravel has also added the `increments()` and `timestamps()` methods. This makes it easy to create Eloquent ORM compatible models very quickly. Don't worry too much about Eloquent for now. We will discover all about it soon enough.

One final trick for the creation of migrations is how to store them in a different location to the default `database/migrations` directory. We can use the `--path` switch to define a new location for our migration class.

Example 08: Make migration in a different path.

```
$ php artisan make:migration create_users_table --path=app/migs
Created Migration: 2016_01_30_155341_create_users_table
```

Our migration will now be created within the `app/migs` directory relative to the root of our project. However, when running your migrations, Artisan won't look in this new location by default. For this reason, be sure to let it know where to find your migrations. We will discover more about this within the next section.

Running Migrations

We went to all this effort to create our new migration; it would be a shame not to run it, wouldn't it? Let's prepare the database to use migrations. Do you remember that I told you that Laravel uses a database table to record the status of its migrations? Well, first we need to create that table.

You can call the migrations table whatever you like. The configuration for the table name is located within `config/database.php`.

Example 09: Migrations configuration.

```
<?php

/*
|--------------------------------------------------------------------------
| Migration Repository Table
|--------------------------------------------------------------------------
|
| This table keeps track of all the migrations that have already run for
| your application. Using this information, we can determine which of
| the migrations on disk haven't actually been run in the database.
|
```

```
12  */
13
14  'migrations' => 'migrations',
```

Simply change the `migrations` index to the name of the table which you wish to use to track your migration status. A sensible default has been provided.

We can install our migrations table by running another Artisan command. Let's run the `install` command now.

Example 10: Install migrations.

```
1  $ php artisan migrate:install
2  Migration table created successfully.
```

Let's examine our database, and look for the `migrations` table to see what has been created.

Example 11: Describe the migrations table.

```
1  mysql> describe migrations;
2  +-----------+--------------+------+-----+---------+-------+
3  | Field     | Type         | Null | Key | Default | Extra |
4  +-----------+--------------+------+-----+---------+-------+
5  | migration | varchar(255) | NO   |     | NULL    |       |
6  | batch     | int(11)      | NO   |     | NULL    |       |
7  +-----------+--------------+------+-----+---------+-------+
8  2 rows in set (0.01 sec)
```

A new table with two fields has been created. Don't trouble yourself with the implementation of the migrations table. Rest assured that it has been created and that the migrations system has been installed.

Oh dear, I have lied to you again. I don't know how this keeps happening? Perhaps I should visit a psychiatrist or something. Well, anyway, I told you that we needed to install the migrations table, and I lied.

You see, Laravel will automatically create the table for us if it doesn't exist when your migrations are executed. It will install the migrations system for you. At least you know about the `migrate:install` command now though right? It's almost as if I planned this whole deception.

Right, let's get started and run our migration for the first time. We can use the `migrate` command to do this.

Example 12: Run the migrations.

```
$ php artisan migrate
    Migrated: 2016_01_30_124846_create_users_table
```

The output from the command is a list of migrations that have been executed. Let's take a look at our database to see if our 'users' table has been created.

Example 13: Describe the users table.

```
mysql> describe users;
+------------+------------------+
| Field      | Type             |
+------------+------------------+
| id         | int(10) unsigned |
| name       | varchar(128)     |
| email      | varchar(255)     |
| password   | varchar(60)      |
| created_at | timestamp        |
| updated_at | timestamp        |
+------------+------------------+
6 rows in set (0.01 sec)
```

I have shortened the table a little to be more consistent with the book's formatting, but you can see that our 'users' table has been created correctly. Awesome!

Let's add a 'title' column to our 'users' table. You might be tempted to open up the migration that we have already made and update the schema to include the new column. Please don't do that.

You see, if one of your teammates had been working on the project and had already run our first migration, then he wouldn't receive our change. Our databases would be in different states.

Instead, let's create a new migration to alter our database. Here we go.

Example 14: Create another migration.

```
$ php artisan make:migration add_title_to_users
Created Migration: 2016_01_30_151627_add_title_to_users_table
```

You will notice that I have given the new migration a descriptive name. You should follow this pattern. Let's alter the schema of our 'users' table within the `up()` method to add the title column.

Example 15: Add another column.

```php
<?php

use Illuminate\Database\Migrations\Migration;

class AddTitleToUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('users', function($table) {
            $table->string('title');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        //
    }
}
```

Great! That should add the column that we need to our 'users' table. Now, say it with me, please.

What goes up must come down.

You're right! We need to provide the `down()` method for this migration class. Let's alter the table to remove the 'title' column.

Example 16: Removing a column.

```php
<?php

use Illuminate\Database\Migrations\Migration;

class AddTitleToUsers extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('users', function($table) {
            $table->string('title');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('users', function($table) {
            $table->dropColumn('title');
        });
    }
}
```

Perfect. Laravel is now able to execute our migration, and also, revert all changes if needed. Let's execute our migrations again.

Example 17: Run the new migration.

```
$ php artisan migrate
    Migrated: 2016_01_30_151627_add_title_to_users_table
```

Laravel knows that our previous migration has already been executed, and so it only executes our latest migration class. Let's examine the table once more.

Example 18: Describe the users table.

```
mysql> describe users;
+------------+------------------+
| Field      | Type             |
+------------+------------------+
| id         | int(10) unsigned |
| name       | varchar(128)     |
| email      | varchar(255)     |
| password   | varchar(60)      |
| created_at | timestamp        |
| updated_at | timestamp        |
| title      | varchar(255)     |
+------------+------------------+
7 rows in set (0.00 sec)
```

As you can see, our new column has been added to the users table. If our migration was committed and shared with the rest of the team, they could simply run `migrate` to bring their databases in line with the new structure.

If we do need to alter one of our existing migration files, we can use the `migrate:refresh` Artisan command to revert all migrations, and then run them once more. Let's try this now with our 'users' table.

Example 19: Refresh the migrations.

```
$ php artisan migrate:refresh
Rolled back: 2016_01_30_151627_add_title_to_users_table
Rolled back: 2016_01_30_124846_create_users_table
Nothing to rollback.
Migrated: 2016_01_30_124846_create_users_table
Migrated: 2016_01_30_151627_add_title_to_users_table
```

Our migrations have been rolled back using the `down()` methods, and then executed once again in the correct order using their respective `up()` methods. Our database is once again in its perfect state.

Remember how we used the `--path` switch in the previous chapter to write our migrations to a new location on the filesystem? Well, I promised you that I would show you how to execute them. I may lie once in a while, but I never go back on a promise. Let's take a look at how we can execute our nonstandard migrations.

Example 20: Migrate a different path.

```
$ php artisan migrate --path=app/migs
Migrated: 2016_01_30_155341_create_users_table
```

See, it's easy? We just use the `--path` switch again to specify the location where our migrations are stored relative to the application root.

I told you that migrations are bi-directional, so that means that we must be able to roll them back? Let's move on.

Rolling Back

Rolling. Rolling. Rolling on the riverrrrr...

Sorry about that, I was a little distracted. Let's see. Ah, yes! Rolling back migrations. We know that we can use `migrate` to execute our migrations, but how do we roll them back?

Let's assume that we used the `migrate` command to restructure our database based upon one of our team-mate's migrations. Unfortunately, our friend's schema changes have broken some of our code, leaving our application broken.

We need to rollback the changes that our team-mate has made. To do this, we can use the `rollback` command. Let's give it a try.

Example 21: Rollback a migration.

```
$ php artisan migrate:rollback
Rolled back: 2016_01_30_151627_add_title_to_users_table
```

When we use the `rollback` command, Laravel rolls back only the migrations that ran the last time we used `migrate`. It's as if the last time that we ran `migrate` never happened.

If we want to roll back **all** migrations, then we can use the `reset` command.

Example 22: Rollback all migrations.

```
$ php artisan migrate:reset
Rolled back: 2016_01_30_151627_add_title_to_users_table
Rolled back: 2016_01_30_124846_create_users_table
Nothing to rollback.
```

You should note that the `reset` command will not remove the migrations table.

Migration Tricks

Oh, you want more do you? I see. Well don't worry, I'm not going to hold anything back. Let's learn a few extra features of the migrations system.

Do you remember the array of connections that we discovered in the database configuration file at `config/database.php`? We can perform our migrations on another connection by specifying the `--database` switch to any of the migration commands.

Example 23: Migrate a different database connection.

```
$ php artisan migrate --database=mysql
Migrated: 2016_01_30_124846_create_users_table
Migrated: 2016_01_30_151627_add_title_to_users_table
```

Our migrations will now be performed on the connection that was nicknamed `mysql` within the configuration file.

Hrm... you still don't sound impressed? Well, alright. I have another trick for you. Sometimes I think that I spoil you, but I have to admit it. You are a great listener.

We can execute our migrations without altering the database, and we can see the intended SQL queries that are the result of our migrations. This way we can check to see what the next migration will do, without risking any damage to our database. This is useful for debugging.

To see the intended SQL result of a migration command, just add the `--pretend` switch. Here's an example.

Example 24: Dry-run migrations.

```
$ php artisan migrate --pretend
CreateUsersTable: create table `users` (`id` int unsigned not null
auto_increment primary key, `name` varchar(128) not null,
`email` varchar(255) not null, `password` varchar(60) not null,
`created_at` timestamp default 0 not null, `updated_at` timestamp
default 0 not null) default character set utf8 collate utf8_unicode_ci
AddTitleToUsersTable: alter table `users` add `title` varchar(255) not null
```

Here we can see the queries that would have been executed against our database if the `--pretend` switch wasn't provided. Neat trick, right?

Yep, you got me...

I told you so!

In the next chapter, we will be taking a look at the Eloquent ORM. Eloquent is a wonderful way of representing your database rows as PHP objects to that they fit in nicely with object-oriented programming.

20. Eloquent ORM

We've learned how to configure our database and how we can use the schema builder to structure tables within our database, but now it's time to get down to the nitty-gritty and learn how we can store information in the database.

Some of you who will have already encountered the database components of Laravel might be wondering why I'm choosing to start with the ORM? Why don't I begin with SQL statements, and then query building?

Let's take a step back and think about why we are here. You are a developer. A PHP developer in fact! Since you are reading this book, I'm hoping that you are a PHP 5+ developer, and will have embraced object-oriented development.

If we are describing the entities in our application as objects, then it makes sense to store them as objects. To retrieve them as objects, and more.

Let's imagine that we are writing an online bookstore.

Object-oriented application design has taught us that we need to identify the objects within our application. A bookstore isn't going to be very successful without any books, right? So there's a fair chance that we will want a book object to represent the individual books used by our application. Typically we will refer to these application objects as 'Models' since they represent part of our applications business model. Here's an example.

Example 01: A wonderful idea.

```
1  <?php
2
3  class Book
4  {
5      /**
6       * The name of our book.
7       *
8       * @var string
9       */
10     public $name;
11
12     /**
13      * A description for our book.
14      *
```

```
15      * @var string
16      */
17     public $description;
18 }
19
20 $book = new Book;
21 $book->name = 'The Colour of Magic';
22 $book->description = 'Rincewind and Twoflower in trouble!';
```

Wonderful!

We have created a book to represent Terry Pratchett's 'The Colour of Magic', one of my personal favourites! Now let's store this book in our database. We will assume that we used the schema builder and have already created a 'books' table with all required columns.

First, we will need to construct an SQL query. I know that you would probably build a prepared query for security reasons, but I want to keep the example simple. That should do the trick.

Example 02: An SQL insert.

```
1  <?php
2
3  $query = "
4      INSERT INTO
5          books
6      VALUES (
7          '{$book->name}',
8          '{$book->description}'
9      );
10 ";
```

We construct an SQL query to insert our object into the database. This query can then be executed using whichever database adapter you use.

I think that it's a real shame that we have to build an SQL query just to store that data in the database. Why bother creating the object in the first place if we are going to transform it into a string for storage? We'd just have to build the object again when retrieving it from the database too. It's a waste of time if you ask me.

I think that we should be able to 'throw' our objects directly at the database without having to build those ugly SQL queries. Hmm, perhaps something like this?

Example 03: An even more wonderful idea.

```
<?php

$book = new Book;
$book->name = 'The Colour of Magic';
$book->description = 'Rincewind and Twoflower in trouble!';
$book->save();
```

The `save()` method would handle the SQL side of things for us; persisting the object to the database. That would be great! Someone should build on this idea of mine.

It's already been done, buddy.

What? Shame... I thought I had found the idea that might bring me fame and fortune. Well, I guess that it's a good thing.

Ah, yes. I remember now. This functionality is provided by object relational mappers, or 'ORM's. ORMs can be used to map our application objects to database tables and individual instances of these objects as rows. You can think of the class properties of these objects as the individual columns for the table.

The ORM will take care of object retrieval and persistence for us. We won't have to write a single line of SQL. This is great news because I can't stand SQL! It's ugly and boring. Objects are much more fun, right?

Many ORMs also offer the ability to manage the relationships between multiple object types. For example, books and authors. Authors and publishers, etc.

Laravel ships with an ORM component called 'Eloquent.' Eloquent is very much true to its name. Its syntax is quite beautiful, and it makes interacting with the database layer of your application stack a pleasing experience, rather than a chore.

When I think about a storage layer, the word CRUD comes to mind. No, I'm not referring to my dislike for SQL this time, but rather the actions that can be performed upon the storage layer.

- **C** - Create a new row.
- **R** - Read existing rows.
- **U** - Update existing rows.
- **D** - Delete existing rows.

Let's learn more about Eloquent by tackling these steps in order. We will start with the creation of Eloquent model instances.

Creating new models.

Before we create our first Eloquent model, we need a quirky example data topic. Hrm... I've just built a new gaming PC, so let's go with video games. We can create some objects to represent video games, but first, we need to create a table schema.

We will create a new migration to build the schema for our 'games' table.

Example 04: Database migration for the games table.

```
<?php

// database/migrations/2016_01_10_213946_create_games_table.php

use Illuminate\Database\Migrations\Migration;

class CreateGamesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('games', function($table) {
            $table->increments('id');
            $table->string('name', 128);
            $table->text('description');
        });
    }

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

Hopefully, this sample code requires no introduction. If you have found anything confusing within the example, then take another look at the schema builder chapter.

You will notice that we named our table `games`. This is because we intend to call our Eloquent model `Game`. Eloquent is smart, by default, it will look for the plural form of the model name as the table to use to store instances of our objects. This behaviour can be overridden, but let's keep things simple for now.

Eloquent models have basic requirements. A model should have an auto incremental column named `id`. This is a unique primary key that can be used to identify a single row within the table. You can add this column to the table structure easily by using the `increments()` method.

Let's run the migration to update the database.

Example 05: Run migrations.

```
$ php artisan migrate
Migrated: 2016_01_10_213946_create_games_table
```

Now we can get started. Let's create a new Eloquent model to represent our games.

Example 06: The Game model.

```
<?php

// app/Game.php

namespace App;

use Eloquent;

class Game extends Eloquent
{

}
```

Here we have a complete Eloquent model that can be used to represent our games. Surprised? Yes, I suppose it is a little sparse, but that's a good thing. Many other ORMs will demand that you build an XML map of the database schema, or create annotations for each of the database columns of the table representing the object. We don't need to do this because Eloquent makes some sensible assumptions.

Let's create a new game.

Example 07: Saving a game instance.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $game = new \App\Game;
    $game->name = 'Assassins Creed';
    $game->description = 'Assassins VS templars.';
    $game->save();
});
```

Hey! That looks familiar! Isn't that how we wanted to persist our objects in the first place? It's clean and straightforward. We create a new instance of our 'Game' model and set its public attributes, which map to table columns. To the values that we require. When we are done, we just call the `save()` method on the object to persist the new row to the database.

Let's visit the `/` URI. We're expecting to receive no response since the query will execute and return nothing from our routed logic. However, we receive something entirely different.

We receive an error screen. What's this error?

Example 08: An unknown column.

```
SQLSTATE[42S22]: Column not found: 1054 Unknown column 'updated_at' in 'field li\
st' (SQL: insert into `games` (`name`, `description`, `updated_at`, `created_at`\
) values (?, ?, ?, ?)) (Bindings: array ( 0 => 'Assassins Creed', 1 => 'Assassin\
s VS templars.', 2 => '2016-01-14 16:30:55', 3 => '2016-01-14 16:30:55', ))
```

When Eloquent creates our new model, it attempts to populate the `updated_at` and `created_at` columns of our table with the current time. This is because it expects us to have added the `->timestamps()` method when building our table schema. It's a sensible default since it never hurts to have a record of creation and update times. However, if you are using an existing database table, or simply don't wish to have the timestamp columns present within your database table, you may want to disable this functionality.

To disable automatic timestamp updates with Eloquent models, just add a new public attribute to your model.

Example 09: Allow timestamps on the Game model.

```
<?php

// app/Game.php

namespace App;

use Eloquent;

class Game extends Eloquent
{
    public $timestamps = false;
}
```

The public attribute `$timestamps` is inherited from the Eloquent base class. It is a boolean value that can be used to enable or disable the automatic timestamp functionality. In the above example, we have set it to `false`, which will let Eloquent know that we wish to disable timestamp updating.

Let's visit the `/` URI once more. This time, the page shows a black result. Don't panic; this is because we have not returned a response from our routed logic. We received no error message, so the SQL query must have been executed. Let's examine the `games` table to see the result.

Example 10: Table content.

```
mysql> use myapp;
Database changed
mysql> select * from games;
+----+-----------------+------------------------+
| id | name            | description            |
+----+-----------------+------------------------+
|  1 | Assassins Creed | Assassins VS templars. |
+----+-----------------+------------------------+
1 row in set (0.00 sec)
```

We can see that our new row has been inserted correctly. Great! We have inserted a new record without writing a single line of SQL. Now that's my kind of victory.

You will notice that we didn't have to specify an `id` value for our object. The `id` column is automatically incremented so that the database layer will handle the numbering of the rows for us. It is not a good idea to modify the `id` column of an Eloquent model. Try to avoid it unless you know what you're doing.

We used the `$timestamps` attribute to disable automatic timestamps. Instead, let's take a look at what happens when we enable them. First, we need to alter our database schema. It's a bad idea to manually modify our database schema or to update existing migrations. This is because our database state might become 'out-of-sync' with our team-mates. Instead, let's create a new migration so that our team-mates can also execute to receive our changes.

Example 11: Create a new migration.

```
$ php artisan make:migration add_timestamps_to_games
Created Migration: 2016_01_14_165416_add_timestamps_to_games_table
```

Our migration has been created. You will notice that we gave the migration a descriptive name to represent our intentions with the migration. This could be useful information should your team-mates later execute the migration. Let's use the schema builder to add timestamps to our game table.

Example 12: Migration to add timestamps.

```php
<?php

// app/migrations/2016_01_14_165416_add_timestamps_to_games_table.php

use Illuminate\Database\Migrations\Migration;

class AddTimestampsToGamesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('games', function($table) {
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
```

```
     */
    public function down()
    {
        //
    }
}
```

We have used `Schema::table()` to alter our 'games' table, and the `timestamps()` method to automatically add the timestamps column. Now, ladies and gentlemen, say it with me.

What goes up must come down!

You do learn fast! Great work. Let's remove the timestamps columns from the table within the `down()` method.

Example 13: Must come down.

```
<?php

// database/migrations/2016_01_14_165416_add_timestamps_to_games_table.php

use Illuminate\Database\Migrations\Migration;

class AddTimestampsToGamesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('games', function($table) {
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
```

```
25      */
26     public function down()
27     {
28         Schema::table('games', function($table) {
29             $table->dropColumn('updated_at', 'created_at');
30         });
31     }
32 }
```

I have used the `dropColumn()` schema builder method to remove the `updated_at` and `created_at` columns from the table within the `down()` method. I thought there might be a lovely `dropTimestamps()` method for this, but apparently not. Not a problem! It's an open source project, so I'll just send a pull request later when I get some free time. Hint hint.

Let's execute our new migration to add the new columns to our 'games' table.

Example 14: Run the migration.

```
1 $ php artisan migrate
2 Migrated: 2016_01_14_165416_add_timestamps_to_games_table
```

Now we have a choice; we could either set the `$timestamps` attribute within our model to true, which would enable the automatic timestamp updating feature.

Example 15: Enable timestamps.

```
1  <?php
2
3  // app/Game.php
4
5  namespace App;
6
7  use Eloquent;
8
9  class Game extends Eloquent
10 {
11     public $timestamps = true;
12 }
```

Or we could remove it because the default value of the `$timestamps` attribute within the parent `Eloquent` model is `true`. Its value will be inherited within our model.

Example 16: Timestamps by default.

```
<?php

// app/Game.php

namespace App;

use Eloquent;

class Game extends Eloquent
{

}
```

Great. Let's execute our / URI once again to insert a new row. We will examine the 'games' table within the database to see the result.

Example 17: Table content.

```
mysql> use myapp;
Database changed
mysql> select * from games;
+----+-----------------+-----------------------+---------------------+---------\
------------+
| id | name            | description           | created_at          | updated_\
at          |
+----+-----------------+-----------------------+---------------------+---------\
------------+
|  1 | Assassins Creed | Assassins VS templars. | 2016-01-14 17:14:13 | 2016-01-\
14 17:14:13 |
+----+-----------------+-----------------------+---------------------+---------\
------------+
1 row in set (0.00 sec)
```

As you can see, the `created_at` and `updated_at` columns have been populated with the current timestamp for us. This is a great timesaver! You might want to remind your applications users about their one year anniversary using your application, and could compare the current date with the `created_at` column for this purpose.

Before we move to the next section, I'd like to share a little trick with you. If you don't want your database table name to be the plural form of your model, then you will need to let Eloquent know about it. Just add the `$table` public attribute to your model and set its value to the string name of your table. Eloquent will then use the provided table name for all future queries relating to this model.

Example 18: Define the table name.

```
<?php

// app/Game.php

namespace App;

use Eloquent;

class Game extends Eloquent
{
    public $table = 'gamezilla_roar';
}
```

If you chose to namespace your models, you would need to use the `$table` attribute to provide simple table names. This is because a model with the namespace and class combination of `MyApp\Models\Game` will result in an expected table name of `my_app_models_games` to avoid collisions with packages within other namespaces which also use the database. You will also notice that Eloquent is brilliant, and will expand a camel cased namespace or model name into its snake cased variant.

Most of the time, I provide the `$table` property just to be on the safe side. It's a more declarative habit, and probably not a bad one to have!

We have learned how to create new rows within our database tables by using Eloquent to treat them as PHP objects. Next, let's take a look at how we can retrieve these existing rows.

Reading Existing Models

Eloquent offers some methods of querying for instances of models. We will examine them all in a future chapter, but for now, we will use the `find()` method to retrieve a single model instance from the database by its `id` column. Here's an example.

Example 19: Find an existing model.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $game = \App\Game::find(1);
    return $game->name;
});
```

We have used the static `find()` method of our model to retrieve an instance of `Game` representing the database row with an `id` value of `1`. We can then access the public attributes of the model instance to retrieve the column values. Let's visit the `/` URL to see the result.

Example 20: Output.

```
Assassins Creed
```

Great! Our existing value has been retrieved. The static `find()` method is inherited from the Eloquent parent class and does not need to be created within your model. As I said earlier, there are many other retrieval methods which will be covered in a later chapter about querying Eloquent models. For now, let's look at how we can update existing table rows.

Updating Existing Models

If you have recently created a new model, then the chances are that you have assigned it to a variable. In the previous section we created a new instance of our `Game` model, assigned the instance to the `$game` variable, updated its columns and used the `save()` method to persist it to the database.

Example 21: Create a new model.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $game = new \App\Game;
    $game->name = 'Assassins Creed';
    $game->description = 'Assassins VS templars.';
    $game->save();
});
```

Just because we have `save()`d our model instance, it doesn't have to mean we can't modify it. We can alter its values directly and call the `save()` method once more to update the existing row. You see, the first time `save()` is used on a new object. It will create a new row and assign an auto incremental `id` column value. Future calls to the `save()` method will persist only the changes to columns for the existing row in our database.

Take a look at the following example.

Example 22: Update a model several times.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $game = new \App\Game;
    $game->name = 'Assassins Creed';
    $game->description = 'Show them what for, Altair.';
    $game->save();

    $game->name = 'Assassins Creed 2';
    $game->description = 'Requiescat in pace, Ezio.';
    $game->save();

    $game->name = 'Assassins Creed 3';
    $game->description = 'Break some faces, Connor.';
    $game->save();
});
```

You might imagine that the above example would have created three entries within the `games` table, but you would be wrong. You will notice that we are only creating a single new instance of the `Game` class. All future calls to `save()` serve to modify this existing database row. The last saved state of the object will be present within the database.

I'm going to truncate my `games` table within the database to demonstrate this example. Go ahead and do it too if you're following along at home.

Example 23: Truncate the games table.

```
mysql> truncate games;
Query OK, 0 rows affected (0.00 sec)
```

Now let's visit the `/` URI once more to execute our routed logic. Here's the resulting contents of the `games` table.

Example 24: Table content.

```
mysql> select * from games;
+----+------------------+--------------------------+---------------------+----\
-----------------+
| id | name             | description              | created_at          | upd\
ated_at          |
+----+------------------+--------------------------+---------------------+----\
-----------------+
|  1 | Assassins Creed 3 | Break some faces, Connor. | 2016-01-14 17:38:50 | 201\
6-01-14 17:38:50 |
+----+------------------+--------------------------+---------------------+----\
-----------------+
1 row in set (0.00 sec)
```

As you can see, 'Assassins Creed 3' was updated by our last `save()` method.

The above method is very useful when you already have a reference to an existing instance of our model, but what if you don't? What if you created the model a long time ago? We can use the `find()` method to retrieve an instance representing an existing database row, and alter it accordingly.

Here's an example.

Example 25: Find a previously saved model by ID.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $game = \App\Game::find(1);
    $game->name = 'Assassins Creed 4';
    $game->description = 'Shiver me timbers, Edward.';
    $game->save();
});
```

Upon inspecting the games table, we discovered that our previous row was inserted with an `id` value of `1`. Using the static `find()` method on our model and the known id, we can retrieve an instance of `Game` representing the existing table row. Once the new instance has been returned, we can modify its column values and use `save()` in the same way as we did earlier.

Here's the resulting table row.

Example 26: Table content.

```
mysql> select * from games;
+----+------------------+---------------------------+---------------------+---\
------------------+
| id | name             | description               | created_at          | up\
dated_at          |
+----+------------------+---------------------------+---------------------+---\
------------------+
|  1 | Assassins Creed 4 | Shiver me timbers, Edward. | 2016-01-14 17:38:50 | 20\
16-01-14 17:49:28 |
+----+------------------+---------------------------+---------------------+---\
------------------+
1 row in set (0.00 sec)
```

As you can see, our existing row has been updated accordingly. Once again, we did not write a single line of SQL. Only beautiful, eloquent PHP. You will also notice that the `updated_at` column has been populated with update time automatically. Very useful!

Deleting Existing Models

Deleting Eloquent models is a simple process. First, we need to get our hands on the instance of the model that we wish to delete. For example, we could use the `find()` method that we discovered in a previous subchapter.

Example 27: Find a model.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $game = \App\Game::find(1);
});
```

Once we have our grubby little mitts on an Eloquent model instance, we can use the `delete()` method to remove the row represented by our model from the database.

Example 28: Find an delete a model.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $game = \App\Game::find(1);
    $game->delete();
});
```

We can also delete a single instance, or multiple instances of our models from the database using their `id` column values and the `destroy()` static method.

Example 29: Delete a model by ID.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    \App\Game::destroy(1);
});
```

To destroy multiple records, you can either pass some `id` values as parameters to the `destroy()` method.

Example 30: Delete multiple models by ID.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    \App\Game::destroy(1, 2, 3);
});
```

Or an array of `id`'s. Like this:

Example 31: Delete multiple models using an array.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    \App\Game::destroy([1, 2, 3]);
});
```

It's entirely up to you!

SQL offers some different and complex ways to query for a particular subset of records. Don't worry! Eloquent can also perform this simple task. In the next chapter, we will learn about the variety of query methods available to the Eloquent ORM. Go ahead and flip the page!

21. Eloquent Queries

In the previous chapter, we discovered how to express our database rows as columns and our tables as classes. This removes the need for writing statements using the Structured Query Language (SQL) and will result in code that is much more readable. We're writing PHP, right? Why bother complicating things by adding another language?

Well, there are some good bits about the SQL. For example, the Q part. Querying. With SQL, we can use some complex comparisons and set arithmetic to retrieve only the results that we require. Replicating *all* of this functionality with Eloquent would be a tremendous task, but fortunately, Eloquent has alternate methods for the most useful of queries. For all the bits that are missing, we can use `raw` queries to provide SQL statements that will return Eloquent ORM result instances. We'll take a closer look at this a bit later. Let's first prepare our database for this chapter.

Preparation

Soon, we will learn how to fill our database with sample data using a technique known as 'seeding.' For now, we will create some dummy records within our database using Eloquent. We won't use any new functionality here. Instead, we will use the skills that we have learned in recent chapters.

First, we are going to need to create a migration to build the schema for our sample table. We are going to use music albums as our demo data. Let's create a migration to build an `albums` table.

Example 01: Create a migration.

```
$ php artisan make:migration create_albums_table
Created Migration: 2016_01_21_103250_create_albums_table
```

Let's fill in the method stubs with the code required to build the schema for our new `albums` table.

Example 02: Our new migration.

```
<?php

use Illuminate\Database\Migrations\Migration;

// database/migrations/2016_01_21_103250_create_albums_table.php

class CreateAlbumsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('albums', function($table) {
            $table->increments('id');
            $table->string('title', 256);
            $table->string('artist', 256);
            $table->string('genre', 128);
            $table->integer('year');
        });
    }

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

In the `up()` method of our migration we use the `Schema` facade to create a new table called `albums`. The table will contain `varchar` columns for the album title, the artist that performed the music, and the genre of the music. We also have an auto-incremental `id` field as required by the Eloquent ORM. Finally, we have an integer field to store the release year for the album.

In the `down()` method we drop the table, restoring the database to its original form.

Let's run our migration to structure the database.

Example 03: Run the migration.

```
$ php artisan migrate
Migration table created successfully.
Migrated: 2016_01_21_103250_create_albums_table
```

Our database now has a structure to hold our sample album data. We now need to create an Eloquent model definition so that we can interact with our tables rows using PHP objects.

Example 04: The Album model.

```
<?php

// app/Album.php

namespace App;

use Eloquent;

class Album extends Eloquent
{
    public $timestamps = false;
}
```

Lovely, simple, clean. We have disabled timestamps on our `Album` model definition to simplify the examples within this section. Normally I like to add timestamps to all of my models. While there may be a slight performance overhead in doing so, and a little extra storage required, I find that the timestamps are most useful in providing an audit trail for an individual application model.

We now have all we need to fill our database table with the dummy album data. As I mentioned earlier, ideally, we would be using database seeding for this task, but for now, we will only create a routed Closure. As a result, we will encounter a little repetition. We will let it slide this time. We only intend to visit this route once.

Example 05: A seeding route.

```
<?php

// app/Http/routes.php

Route::get('/seed', function() {
    $album = new \App\Album;
    $album->title           = 'Some Mad Hope';
    $album->artist          = 'Matt Nathanson';
    $album->genre           = 'Acoustic Rock';
    $album->year            = 2007;
    $album->save();

    $album = new Album;
    $album->title           = 'Please';
    $album->artist          = 'Matt Nathanson';
    $album->genre           = 'Acoustic Rock';
    $album->year            = 1993;
    $album->save();

    $album = new Album;
    $album->title           = 'Leaving Through The Window';
    $album->artist          = 'Something Corporate';
    $album->genre           = 'Piano Rock';
    $album->year            = 2002;
    $album->save();

    $album = new Album;
    $album->title           = 'North';
    $album->artist          = 'Something Corporate';
    $album->genre           = 'Piano Rock';
    $album->year            = 2002;
    $album->save();

    $album = new Album;
    $album->title           = '...Anywhere But Here';
    $album->artist          = 'The Ataris';
    $album->genre           = 'Punk Rock';
    $album->year            = 1997;
    $album->save();

    $album = new Album;
```

```
42      $album->title           = '...Is A Real Boy';
43      $album->artist          = 'Say Anything';
44      $album->genre           = 'Indie Rock';
45      $album->year            = 2006;
46      $album->save();
47  });
```

These are some personal favourites of mine. I'm hoping that those of you who aren't fans of punk rock haven't been too offended by music taste, and will continue with the chapter.

As you can see, for each of our dummy rows we create a new instance of an `Album` model, populate all fields, and save our populated model to the database.

Go ahead and visit the `/seed` URI to fill the `albums` table with our sample data. You should receive a blank page because we did not return a response from the route.

Now that our sample data has been written to the database we can delete the `/seed` route. We don't need it anymore! Our preparation is complete. Let's start learning about Eloquent queries.

Eloquent To String

Objects in PHP can optionally include a `__toString()` method. You may have come across this in the past. It was added in PHP 5.2 along with some of the other double _ underscore prefixed magic methods. This method can be used to control how the object should be represented as a string.

Thanks to this method, our Eloquent models can also be expressed as a string. You see, the Eloquent base class that we extend with our models contains a `__toString()` method. This method will return a JSON string that will represent the values of our Eloquent model.

This might sound a little confusing, and it's been a while since we have seen an example. Let's first look at the normal way of exposing the values contained within our Eloquent model instances.

Example 06: Find an album.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $album = \App\Album::find(1);
    return $album->title;
});
```

In the example above we use the inherited static `find()` method of our `Album` model and pass an integer value of `1`. We retrieve a model instance representing the album table row with an `id` column value of `1`. Next, we return the `title` attribute of the model instance to be displayed as the response of the route.

If we visit the `/` URI, we receive the following response.

Example 07: Output.

```
 Some Mad Hope
```

The title of the first dummy album inserted into our database as expected. Let's modify the route to return instead the model instance itself as a response from our routed Closure.

Example 08: Return a model instance.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::find(1);
});
```

Let's visit the `/` URI to examine the response.

Example 09: Output.

```
{"id":1,"title":"Some Mad Hope","artist":"Matt Nathanson","genre":"Acoustic Rock\
","year":2007}
```

That looks like JSON to me! All JSON strings created by the framework have all extra whitespace and indentation removed to save bandwidth when transferring the data. I'm going to beautify all of the JSON examples within this chapter manually. Don't be surprised if your outputs look a little more jumbled than the ones displayed in this chapter.

Let's beautify the above output.

Example 10: Output.

```
{
    "id": 1,
    "title": "Some Mad Hope",
    "artist": "Matt Nathanson",
    "genre": "Acoustic Rock",
    "year": 2007
}
```

Laravel has executed the inherited `__toString()` method of our Eloquent model instance to represent its values as a JSON string. This is useful when creating APIs that serve JSON data. It's also a great way of expressing the output of our queries for the rest of the chapter.

Some Eloquent methods will return some model instances as a result, instead of the single model instance returned by the above example. Let's take a quick look at the `all()` method which is used to retrieve all rows as Eloquent model instances.

Example 11: Iterate our albums.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $albums = \App\Album::all();
    foreach ($albums as $album) {
        echo $album->title;
    }
});
```

We use the `all()` method of our `Album` model to retrieve an array of Eloquent model instances that represent the rows of our `albums` table. Then we loop through the array and output the title for each of the `Album` instances.

Here's the result that we receive from the `/` URI.

Example 12: Output.

```
Some Mad HopePleaseLeaving Through The WindowNorth...Anywhere But Here...Is A Re\
al Boy
```

Great! Those are all of the album titles. They are all stuck together because we didn't insert a HTML line break `<br />` element. Don't worry about it. At least we can retrieve them all.

I'm sorry about this, but, once again I have lied to you. If you previously used Laravel 3, the concept of a retrieval method returning an array of model instances will be familiar to you. However, Laravel 5 doesn't return an array from such methods. Instead, it returns a `Collection`.

> I don't believe you. If it doesn't return an array then how did we loop through the results?

That's simple. The `Collection` (`Illuminate\Database\Eloquent\Collection`) object implements an interface which allows the object to be iterable. It can be looped through using the same functionality as standard PHP arrays.

> Hmm, I see. I'm not going to take the word of a liar so quickly, though.

Ah I see, you require additional proof? Let's dump the `$albums` attribute to see what we are working with. This should do the trick.

Example 13: Dump all albums.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $albums = \App\Album::all();
    var_dump($albums);
});
```

When we visit the `/` URI, we receive the following response.

Example 14: Output.

```
object(Illuminate\Database\Eloquent\Collection)[134]
  protected 'items' =>
    array (size=6)
      0 =>
        object(App\Album)[127]
          public 'timestamps' => boolean false
          protected 'connection' => null
          protected 'table' => null
          protected 'primaryKey' => stri

        ... loads more information ...
```

> Woah, you weren't lying this time!

As you can see, the result of any methods that return multiple model instances is represented by an instance of `Illuminate\Database\Eloquent\Collection`. You can see from the output of `var_dump` that this object holds an internal array of our model instances called `items`.

The benefit of the collection object is that it also includes some useful methods for transforming and retrieving our model instances. In a later chapter, we will examine these methods in more detail, but for now, it's worth knowing that the `Collection` object also includes a `__toString()` method. This method functions in a similar manner to the one on our model instances but instead creates a JSON string that will represent our model instances as a multi-dimensional array.

Let's return the `Collection` object that is the result of the `all()` method as the response of our routed Closure.

Like this:

Example 15: Return all model instances.

```php
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::all();
});
```

The response that we receive after visiting the `/` URI is as follows.

Example 16: Output.

```
[
    {
        "id": 1,
        "title": "Some Mad Hope",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 2007
    },
    {
        "id": 2,
        "title": "Please",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 1993
    },
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 5,
        "title": "...Anywhere But Here",
        "artist": "The Ataris",
        "genre": "Punk Rock",
        "year": 1997
    },
    {
        "id": 6,
        "title": "...Is A Real Boy",
        "artist": "Say Anything",
        "genre": "Indie Rock",
```

```
42          "year": 2006
43      }
44  ]
```

We receive a JSON string containing an array of objects that represent the values of our individual albums.

So, why are we learning about the `__toString()` functionality now? We don't intend to build a JSON API in this chapter, are we? No, we're not quite ready for that yet.

You see, I can use the JSON output to display the results of the queries that we will be executing throughout the rest of the chapter. It will be more readable than a bunch of `foreach()` loops for our result sets. Now you know exactly why these results are being outputted as JSON. Everyone's a winner!

Now that we have our database filled with dummy data, and have identified a way of displaying query results, let's take a look at the structure of Eloquent queries.

Query Structure

Eloquent queries are used to retrieve results based on some rules or criteria. We don't always want to retrieve all of your album rows. Sometimes, we will only want to retrieve the discography for a single artist. In these circumstances, we would use a query to ask for only rows that have a `title` column value of the artist that we desire.

Eloquent queries can be broken down into three parts.

- The model.
- Query Constraints
- Fetch methods.

The model is the model instance that we wish to perform the query upon. All of the examples within this section will be forming queries based on the `Album` model.

Query constraints are rules that are used to match a subset of our table rows. This way we can return only the rows that we are interested in. The most common constraint used with SQL is the `WHERE` clause.

Finally, we have the fetch methods. These are the methods that are used to perform the query and return the result.

Let's take a look at the structure of an Eloquent query in its simplest form.

Example 17: A fetch method.

```
<?php

Model::fetch();
```

All of our queries act upon one of our Eloquent models. The constraint methods are entirely optional, and in the above example, are not present.

Next, we have a fetch method. This method doesn't exist. We're just using it to demonstrate the shape of a query. The first method of a query chain is always called statically, with two colons `::`.

Eloquent queries can consist of no constrains, a single constraint, or many constraints. It's entirely up to you. Here's how a query will look with a single constraint.

Example 18: A constraint method.

```
 <?php

Model::constraint()
    ->fetch();
```

Notice how the constraint is now the static method and our fetch method has been chained onto the end of the first method? We can add as many constraints to the query as we require.

Example 19: Multiple constraints.

```
 <?php

Model::constraint()
    ->constraint()
    ->constraint()
    ->fetch();
```

Constraints are entirely optional, but all queries must begin with a model and end with a fetch method. Here's an example using our `Album` model.

Example 20: Constraints are optional.

```
<?php

Album::all();
```

The `Album` is our model and the `all()` method is one of our fetch methods because it is used to retrieve the result of our query.

Fetch methods can be used to return either a single model instance or a `Collection` of model instances. However, as we discovered earlier, both can be expressed in JSON format as the response of a routed Closure or controller action.

We know that our query constraints are optional, so let's start by looking at the variety of fetch methods that are available.

Fetch Methods

Let's begin with some of the fetch methods that you might have encountered within previous chapters. First, we have the `find()` method.

Find

The `find()` method can be used to retrieve a single Eloquent model instance by the `id` column of its row. If the first parameter of the method is an integer, then only a single instance will be returned. Here is an example.

Example 21: Return a model.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::find(1);
});
```

We want to retrieve the database row with an `id` of `1`, so only a single model instance is returned.

Example 22: Output.

```
{
    "id": 1,
    "title": "Some Mad Hope",
    "artist": "Matt Nathanson",
    "genre": "Acoustic Rock",
    "year": 2007
}
```

If we instead provide an array of `id` values, then we receive a `Collection` of model instances.

Example 23: Find by array.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::find([1, 3]);
});
```

Here is the result. A collection containing the model instances that represent rows with an `id` column value of `1` and `3`.

Example 24: Output.

```
[
    {
        "id": 1,
        "title": "Some Mad Hope",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 2007
    },
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    }
]
```

All

The `all()` method can be used to return a collection of model instances that represent all rows contained within the table. Here is an example of the `all()` method.

Example 25: All albums.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::all();
});
```

We receive a collection containing instances of all of the Albums contained within our database.

Example 26: Output.

```
[
    {
        "id": 1,
        "title": "Some Mad Hope",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 2007
    },
    {
        "id": 2,
        "title": "Please",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 1993
    },
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
```

```
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 5,
        "title": "...Anywhere But Here",
        "artist": "The Ataris",
        "genre": "Punk Rock",
        "year": 1997
    },
    {
        "id": 6,
        "title": "...Is A Real Boy",
        "artist": "Say Anything",
        "genre": "Indie Rock",
        "year": 2006
    }
]
```

First

In circumstances where a collection of model instances will typically be returned, the `first()` fetch method can be used to retrieve the *first* model instance stored within. It's very useful if you would rather that a query return a single instance, rather than a collection of model instances. Without a constraint, `first()` will just return the first row in the database table.

Here's an example.

Example 27: First.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::first();
});
```

We receive a single model instance. The first album stored within our database table.

Example 28: Output.

```
{
    "id": 1,
    "title": "Some Mad Hope",
    "artist": "Matt Nathanson",
    "genre": "Acoustic Rock",
    "year": 2007
}
```

Update

We don't have to retrieve only model instances. We can also change them. Using the `update()` method, we can update the values of the table rows that are the result of the Eloquent query. Only pass a key-value array as the first parameter to the `update()` method to change the column values for each row. The array key represents the name of the column to change, and the value represents the new intended value for the column.

The `update()` method is unique, and cannot be used without a constraint. Therefore, I will use a simple `where()` constraint within the example. If you don't understand it, then don't worry about it. We will cover constraints in detail within the next section. Here's an example that will modify our `albums` table. (Don't worry. I will restore it for the next example.)

Example 29: Update.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
        \App\Album::where('artist', '=', 'Matt Nathanson')
            ->update(array('artist' => 'Dayle Rees'));

        return \App\Album::all();
});
```

We update the `artist` field of all rows with an artist of `Matt Nathanson`, changing its value to `Dayle Rees`. The `update()` method doesn't retrieve model instances, so instead we return a collection of all model instances using `all()`.

Example 30: Output.

```
[
    {
        "id": 1,
        "title": "Some Mad Hope",
        "artist": "Dayle Rees",
        "genre": "Acoustic Rock",
        "year": 2007
    },
    {
        "id": 2,
        "title": "Please",
        "artist": "Dayle Rees",
        "genre": "Acoustic Rock",
        "year": 1993
    },
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 5,
        "title": "...Anywhere But Here",
        "artist": "The Ataris",
        "genre": "Punk Rock",
        "year": 1997
    },
    {
        "id": 6,
        "title": "...Is A Real Boy",
        "artist": "Say Anything",
        "genre": "Indie Rock",
```

```
        "year": 2006
    }
]
```

As you can see, I'm now a rockstar. Awesome!

Delete

Much like the `update()` method, the `delete()` method will not return any instances. Instead, it will remove the rows that are the result of the query from the database table.

Example 31: Delete.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
        \App\Album::where('artist', '=', 'Matt Nathanson')
            ->delete();

        return \App\Album::all();
});
```

We query for all albums that have an artist column value of `Matt Nathanson`, and then we use the `delete()` method to delete their rows from the database.

Example 32: Output.

```
[
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
```

```
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 5,
        "title": "...Anywhere But Here",
        "artist": "The Ataris",
        "genre": "Punk Rock",
        "year": 1997
    },
    {
        "id": 6,
        "title": "...Is A Real Boy",
        "artist": "Say Anything",
        "genre": "Indie Rock",
        "year": 2006
    }
]
```

The albums by Matt Nathanson have been removed from our database. Which is a real shame, because he makes beautiful music!

Here's a quick tip. If you intend to delete all table rows for a particular model, then you might find the `truncate()` method to be more descriptive.

Here's an example.

Example 33: Truncate.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
        \App\Album::truncate();
        return \App\Album::all();
});
```

As you can see, all of our table rows have now vanished!

Example 34: Output.

```
[ ]
```

Get

Get is the most important of our fetch methods. It is used to retrieve the result of the query. For example, if we use a `where()` constraint to limit a result set to a single artist, then it wouldn't make any sense to use the `all()` trigger method. Instead, we use the `get()` method to retrieve a model instance collection.

Confused? Here's the `get()` method in combination with a `where()` constraint.

Example 35: Get.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('artist', '=', 'Something Corporate')
        ->get();
});
```

We receive a collection of model instances that have an `artist` column containing a value of `Something Corporate`.

Example 36: Output.

```
[
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    }
]
```

The `get()` method has an optional parameter. You can pass an array of column names to it, and the result objects will only contain values for those columns. Here's an example.

Example 37: Get columns.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('artist', '=', 'Something Corporate')
        ->get(['id', 'title']);
});
```

We pass an array with the values `id` and `title` to the `get()` method. Here is the result set that we retrieve.

Example 38: Output.

```
[
    {
        "id": 3,
        "title": "Leaving Through The Window"
    },
    {
        "id": 4,
        "title": "North"
    }
]
```

As you can see, only the columns that we requested are present in the results.

Pluck

The `pluck()` method can be used to retrieve a value for a single column. Here's an example.

Example 39: Pluck.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::pluck('artist');
});
```

The first and only parameter is the name of the column that we wish to retrieve the value for. If the query matches multiple results, then only the value of the first result will be returned. Here's the result that we receive from the above example.

Example 40: Output.

```
Matt Nathanson
```

Lists

While the `pluck()` method will retrieve only one value for a particular column, the `lists()` method will retrieve an array of values for the specified column across all result instances. Let's clarify this with an example.

Example 41: Lists.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::lists('artist');
});
```

Once again, the `lists()` method accepts one single parameter. The name of the column that we wish to retrieve all values for. Here is the result of our query.

Example 42: Output.

```
[
    "Matt Nathanson",
    "Matt Nathanson",
    "Something Corporate",
    "Something Corporate",
    "The Ataris",
    "Say Anything"
]
```

As you can see, we have retrieved the values contained within the `artist` column for all of our table rows.

ToSql

Right, this one isn't a fetch method, but it is useful! You can use the `toSql()` method anywhere where you would ordinarily use a fetch method, typically the end of a query chain, and it will return the SQL that represents the query as a string.

Let's look at an example.

Example 43: To SQL.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('artist', '=', 'Something Corporate')
        ->toSql();
});
```

Similar to the previous example, but this time we call `toSql()` instead of `get()`. Here's the result we receive.

Example 44: Output.

```
select * from `albums` where `artist` = ?
```

Very useful for debugging indeed!

What's the question mark for?

Laravel's query builder uses prepared statements. This means that the question marks are placeholders that will be replaced with your actual values or 'bindings'. The benefit is that your bindings will be escaped before replaced into the string, to avoid an SQL injection attempt.

Now that we have discovered the fetch methods, it's time to learn about how to add rules to our queries.

Query Constraints

The fetch methods from the previous chapter are useful for retrieving model collections and instances from our database. However, we sometimes need to fine-tune the result to only a few specific rows. That's when query constraints become useful.

Set based arithmetic allows us to capture a subset of a much larger set of values. This is what we are trying to accomplish using query constraints. However, I have also snuck some transformation methods into the chapter to change the ordering of results.

Let's get started with a method that represents the most common SQL query constraint. The `WHERE` clause.

Where

The first constraint method that we will examine is the `where()` method. If you have used SQL in the past, then you will likely have come across the `WHERE` clause for retrieving table rows by matching the value of their columns.

Let's lead with an example.

Example 45: Where.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('artist', '=', 'Matt Nathanson')
        ->get();
});
```

We can use the `where()` method to limit the results to albums which have an artist column value of `Matt Nathanson` only.

The `where()` method will accept three parameters. The first parameter is the name of the column that we wish to perform the comparison on. In this example, we wish to perform the comparison on the `artist` column. The second parameter is the operator to use for the comparison. In our example, we wish to ensure that the `artist` column **is equal** to a value, so we use the equals `=` symbol.

We could have used any of the other standard comparison operators supported by the SQL, such as `<`, `>`, `=>`, `=<`, etc. Experiment with operator types to retrieve the results that you require.

The third parameter is the value that we will compare with. In our example, we wish to ensure that the `artist` column matches `Matt Nathanson`. In this instance `Matt Nathanson` is the value.

Once again, the `where()` method is only a query constraint. We will use the `get()` method to retrieve a `Collection` of results. Let's take a look at the response from the `/` URI.

Example 46: Output.

```
[
    {
        "id": 1,
        "title": "Some Mad Hope",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 2007
    },
    {
        "id": 2,
        "title": "Please",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 1993
    }
]
```

Excellent. Both albums from the database that have an `artist` column value of `Matt Nathanson` were returned to us. This would be useful if we intended on providing sections of a music website for displaying the discographies for an individual artist.

It's worth remembering that the `get()` and `first()` methods are interchangeable. Let's alter the existing example to retrieve only the first instance that matches the provided condition.

Example 47: First instance.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('artist', '=', 'Matt Nathanson')
        ->first();
});
```

Now the query will only retrieve a single model instance representing the first row that is a match to the provided constraint. Here is the result from the `/` URI.

Example 48: Output.

```
{
    "id": 1,
    "title": "Some Mad Hope",
    "artist": "Matt Nathanson",
    "genre": "Acoustic Rock",
    "year": 2007
}
```

Let's try another operator with the `where()` method. How about `LIKE`? The `LIKE` SQL operator can be used to compare parts of a string by using a percentage `%` symbol as a wildcard.

Here's an example.

Example 49: Like.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('title', 'LIKE', '...%')
        ->get();
});
```

In the above example, we would like to retrieve all rows with a `title` field that starts with three period characters `...`. The percentage `%` sign will let the database know that we don't care what value comes after our triple period. As a side note, I know that a triple period is known as an ellipsis, I just thought this might be easier for readers who don't have English as a first language.

Let's take a look at the result from the `/` URI.

Example 50: Output.

```
[
    {
        "id": 5,
        "title": "...Anywhere But Here",
        "artist": "The Ataris",
        "genre": "Punk Rock",
        "year": 1997
    },
    {
        "id": 6,
        "title": "...Is A Real Boy",
        "artist": "Say Anything",
        "genre": "Indie Rock",
        "year": 2006
    }
]
```

We receive a result collection for the albums titled '...Anywhere But Here' and '...Is A Real Boy'. Both of which start with a triple period, and are fabulous albums.

We aren't limited to a single `where()` method call within a query. We can chain multiple `where()` methods together to retrieve rows based on some different criteria. Here's an example.

Example 51: Multiple wheres.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('title', 'LIKE', '...%')
        ->where('artist', '=', 'Say Anything')
        ->get();
});
```

In the above example, we wish to find rows that have an artist column that starts with a triple period **and** an artist column that is equal to 'Say Anything'. The **and** is important. Both constraints must match for a row to exist within the result set.

Here's the result from the above example.

Example 52: Output.

```
[
    {
        "id": 6,
        "title": "...Is A Real Boy",
        "artist": "Say Anything",
        "genre": "Indie Rock",
        "year": 2006
    }
]
```

A collection containing a single model instance, an album with a title that begins with a triple period and an artist value of 'Say Anything'. We receive a `Collection` containing Say Anything's '...Is A Real Boy' album. One of my personal favourites!

OrWhere

We don't always need both constraints to match. Sometimes a match for either condition is good enough for us. In situations such as this, we can use the `orWhere()` method. In fact, most of the constraints within this chapter will have an alternative version prefixed with `or` that will allow an other constraint to match. For this reason, I won't provide separate sections for the `or` method variations in future.

As always, here's an example.

Example 53: Or where.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('title', 'LIKE', '...%')
        ->orWhere('artist', '=', 'Something Corporate')
        ->get();
});
```

We provide an initial constraint stating that the album title *must* begin with a triple period (yes, I know it's called ellipses). Then we include an `orWhere()` method that states that our result set can also consist of results which have an `artist` column value of `Something Corporate`.

Let's take a look at the result.

Example 54: Output.

```
[
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 5,
        "title": "...Anywhere But Here",
        "artist": "The Ataris",
        "genre": "Punk Rock",
        "year": 1997
    },
    {
        "id": 6,
        "title": "...Is A Real Boy",
        "artist": "Say Anything",
        "genre": "Indie Rock",
        "year": 2006
    }
]
```

We receive a `Collection` of result instances that have an album `title` that starts with a triple period **or** `artist` columns with a value of `Something Corporate`.

You can chain together as many `where()` and `orWhere()` methods as you need to filter your table rows down to the required result set.

WhereRaw

The `whereRaw()` method can be used to provide a string of SQL to perform a `WHERE` condition on the result set. Here's an example.

Example 55: Where raw.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::whereRaw('artist = ? AND title LIKE ?', [
        'Say Anything', '...%'
    ])
    ->get();
});
```

The `whereRaw()` method accepts a string of SQL as its first paramater. All question `?` marks within the string are replaced by array elements from the second parameter to the method in sequential order. If you have ever bound properties to a prepared statement with SQL, then this syntax will be familiar to you. The values provided will be escaped to avoid SQL injection attacks.

Once the query builder has performed the neccesary transformation, the resulting SQL will look like this.

Example 56: Parsed query.

```
artist = 'Say Anything' AND title LIKE '...%'
```

The result of our query is as follows.

Example 57: Output.

```
[
    {
        "id": 6,
        "title": "...Is A Real Boy",
        "artist": "Say Anything",
        "genre": "Indie Rock",
        "year": 2006
    }
]
```

You can use the `whereRaw()` method in circumstances where you require complex SQL in addition to your `where()` type constraints. Just like the `where()` method, the `whereRaw()` method can be chained multiple times and with other constraint methods to limit your result set. Once again, the `orWhereRaw()` method is included to allow for other conditions.

WhereBetween

The `whereBetween()` method is used to check that the value of a column is between two provided values. It's best described using an example. In fact, I think everything is. Strange, right? Maybe it's because Laravel code seems to speak for itself!

Example 58: Where between.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::whereBetween('year', array('2000', '2010'))
    ->get();
});
```

The first parameter to the `whereBetween()` method is the name of the column that we wish to compare. The second parameter is an array of two values, a starting value, and a limit. In the above example, we are looking for albums that have a release `year` between 2000 and 2010. Here are the results.

Example 59: Output.

```
[
    {
        "id": 1,
        "title": "Some Mad Hope",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 2007
    },
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
```

```
        "year": 2002
    },
    {
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 6,
        "title": "...Is A Real Boy",
        "artist": "Say Anything",
        "genre": "Indie Rock",
        "year": 2006
    }
]
```

The result is as expected. Some albums from the 2000s.

As with the other `where()` type methods, you can chain as many as you need to, and as always, we have an `orWhereBetween()` alternative method.

WhereNested

The `whereNested()` method is a clean way of applying multiple where constraints to a query. Simply pass a Closure as the first parameter to the method, and give the Closure a placeholder parameter named whatever you like. I like to name mine `$query`.

Example 60: Where nested.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::whereNested(function($query)
    {
        $query->where('year', '>', 2000);
        $query->where('year', '<', 2005);
    })
    ->get();
});
```

Within the Closure, we may apply as many `where()` type constraints or `orWhere()` type constraints to the `$query` object as we wish. This will then become part of your main query. It just looks a whole lot neater! Here's the result set from the above example.

Example 61: Output.

```
[
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    }
]
```

Note that there is no `orWhereNested()` alternative to this method, but here's the secret. We can also pass a Closure to `orWhere()`. Here's an example.

Example 62: Or where Closure.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::whereNested(function($query)
    {
        $query->where('year', '>', 2000);
        $query->where('year', '<', 2005);
    })
    ->orWhere(function($query)
    {
        $query->where('year', '=', 1997);
    })
    ->get();
});
```

We wish for an album to have a release year between 2000 and 2005 **or** have the release year of 1997. Here's the SQL that is generated from the above method.

Example 63: Parsed query.

```
select * from `albums` where (`year` > ? and `year` < ?) or (`year` = ?)
```

These are the results from the above query.

Example 64: Output.

```
[
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 5,
        "title": "...Anywhere But Here",
        "artist": "The Ataris",
        "genre": "Punk Rock",
        "year": 1997
    }
]
```

WhereIn

The `whereIn()` method can be used to check that a column value exists within a set of values. It's useful when you already have an array of possible values to hand. Let's take a look at how it can be used.

Example 65: Where in.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $values = ['Something Corporate', 'The Ataris'];
    return \App\Album::whereIn('artist', $values)->get();
});
```

The first parameter to the `whereIn()` method is the column that we wish to perform the comparison on. The second value is the array of values to search within.

The resulting SQL from the above query looks like this.

Example 66: Parsed query.

```
select * from `albums` where `artist` in (?, ?)
```

Here's the collection of results that we receive from the example query.

Example 67: Output.

```
[
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 5,
        "title": "...Anywhere But Here",
        "artist": "The Ataris",
```

```
        "genre": "Punk Rock",
        "year": 1997
    }
]
```

The `whereIn()` method also has the usual method alternative in the form of `orWhereIn()` and can be chained multiple times.

WhereNotIn

The `whereNotIn()` method is the direct opposite to the `whereIn()` method. This time, you provide a list of values and the column value must not exist within the set.

Let's take a look at an example.

Example 68: Where not in.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    $values = ['Something Corporate', 'The Ataris'];
    return \App\Album::whereNotIn('artist', $values)->get();
});
```

Once again, we pass the comparison column as the first parameter and our array of values as the second parameter.

Here's the resulting SQL.

Example 69: Parsed query.

```
select * from `albums` where `artist` not in (?, ?)
```

Finally, here's the result set from our sample query. All of the albums that aren't identified by the artists within our values array.

Example 70: Output.

```
[
    {
        "id": 1,
        "title": "Some Mad Hope",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 2007
    },
    {
        "id": 2,
        "title": "Please",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 1993
    },
    {
        "id": 6,
        "title": "...Is A Real Boy",
        "artist": "Say Anything",
        "genre": "Indie Rock",
        "year": 2006
    }
]
```

Once again, the `orWhereNotIn()` is also available as an alternative.

WhereNull

The `whereNull()` constraint can be used when you need to retrieve rows that have a column value of `NULL`. Let's check an example.

Example 71: Where null.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::whereNull('artist')->get();
});
```

The single parameter for the `whereNull()` method is the name of the column that you wish to contain a null value. Let's take a look at the generated SQL for this query.

Example 72: Parsed query.

```
select * from `albums` where `artist` is null
```

Now let's take a look at the result set for the query.

Example 73: Output.

```
[ ]
```

Oh, that's right, we don't have any `NULL` values in our database! I don't want to rewrite this chapter again, so you will have to use your imagination here. If we had an artist column with a `value` of `NULL`, then its row would appear in the result set.

Yes, you guessed it! The `orWhereNull()` method is also available.

WhereNotNull

The `whereNotNull()` method is the opposite of the `whereNull()` method. This time, we should be able to see some results. It will return rows that have a column value that isn't equal to `NULL`. Let's take a closer look.

Example 74: Where not null.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::whereNotNull('artist')->get();
});
```

The first and only parameter to the method is the column name. Here's the generated SQL for the query.

Example 75: Parsed query.

```
select * from `albums` where `artist` is not null
```

Here is the result set matching the example query.

Example 76: Output.

```
[
    {
        "id": 1,
        "title": "Some Mad Hope",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 2007
    },
    {
        "id": 2,
        "title": "Please",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 1993
    },
    "... 4 more ..."
]
```

All of the albums in our database. This is because none of the `artist` columns have a value of `NULL`.

Once more, the `orWhereNotNull()` method is available to perform an **or** type query.

OrderBy

The `orderBy()` method can be used to order the results returned by your query by the value of a specific column. Let's dive right in with an example.

Example 77: Order by.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('artist', '=', 'Matt Nathanson')
        ->orderBy('year')
        ->get();
});
```

The first parameter to the `orderBy()` method is the name of the column that we wish to order by. By default, the ordering will be in ascending order.

Here's the generated SQL.

Example 78: Parsed query.

```
select * from `albums` where `artist` = ? order by `year` asc
```

Here's the result set from the query.

Example 79: Output.

```
[
    {
        "id": 2,
        "title": "Please",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 1993
    },
    {
        "id": 1,
        "title": "Some Mad Hope",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 2007
```

```
    }
]
```

Great! Our albums have been returned in ascending order by release year. What if we want them to be descending? Don't worry. Laravel has got you covered!

Example 80: Descending order.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('artist', '=', 'Matt Nathanson')
        ->orderBy('year', 'desc')
        ->get();
});
```

We add a second parameter to the `orderBy()` method with a value of `desc`. This tells Laravel that we wish to retrieve our results in descending order. Here's the generated SQL.

Example 01: Parsed query.

```
select * from `albums` where `artist` = ? order by `year` desc
```

Now here's the updated result set.

Example 81: Output.

```
[
    {
        "id": 1,
        "title": "Some Mad Hope",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 2007
    },
    {
        "id": 2,
        "title": "Please",
        "artist": "Matt Nathanson",
```

```
        "genre": "Acoustic Rock",
        "year": 1993
    }
]
```

Switcheroo! Our results are now in descending order.

We can use the `orderBy()` clause with any combination of the constraints within this chapter. We can also use additional `orderBy()` methods to provide additional ordering in the order that the methods are provided.

Take

The `take()` method can be used to limit the result set. Here's an example.

Example 82: Take.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::take(2)
        ->get();
});
```

The first parameter to the `take()` method is the amount of rows that you wish to limit by. In the above example, we only wish for the query to return two result objects.

Here's the SQL that is generated by the query.

Example 83: Parsed query.

```
select * from `albums` limit 2
```

Finally, here is the result set that we receive.

Example 84: Output.

```
[
    {
        "id": 1,
        "title": "Some Mad Hope",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 2007
    },
    {
        "id": 2,
        "title": "Please",
        "artist": "Matt Nathanson",
        "genre": "Acoustic Rock",
        "year": 1993
    }
]
```

Take can be used in combination with any of the other query constraints. Mix and match!

Skip

When using the `take()` method, the `skip()` method can be used to provide an offset for the query result set. Here's an example.

Example 85: Skip.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::take(2)
        ->skip(2)
        ->get();
});
```

The `skip()` method accepts a single parameter to provide an offset. In the above example, the first two rows will be discarded from the result set. Here is the generated SQL.

Example 86: Parsed query.

```
select * from `albums` limit 2 offset 2
```

Here's the result set that we receive.

Example 87: Output.

```
[
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    }
]
```

As you can see, the first and second rows have been skipped. Instead, starting at the third row in the database.

Magic Where Queries

Now it's time for something magical! By now, you must be more than familiar with the `where()` query. The `where()` query is responsible for restricting a column to a certain value within your result set. Here's an example to remind you.

Example 88: Normal where query.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('artist', '=', 'Something Corporate')
        ->get();
});
```

We intend to retrieve a result set where the `artist` column of each row is equal to `Something Corporate`. It's a nice clean way of restricting the database rows to our desired result set. Could it get any cleaner? Well, as it happens, yes it could! We could use the magical where query syntax.

Take a close look at the following example.

Example 89: Magic where query.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::whereArtist('Something Corporate')->get();
});
```

Wait, what's that `whereArtist()` method? We didn't learn about that in our query constraints chapter. Well, this method is a little special. First, let's visit the `/` URI to see the result.

Example 90: Output.

```
[
    {
        "id": 3,
        "title": "Leaving Through The Window",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    },
    {
```

```
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    }
]
```

It appears to function in a similar manner to our `where()` method with an equals `=` operator. Right, it's time to explain what's happening. You see, the `where()` equals query is likely the most common query of all, and because of this, Taylor has provided a convenient shortcut.

You can append the name of the column that you wish to query on to the `where()` method. First, you must capitalise the first letter of the field that you wish to compare. In our example, we used the `artist` column, so the resulting method name is `whereArtist()`. If our field name is snake cased, for example, `shoe_size`, then we must uppercase the first letter of each word to `whereShoeSize()`.

The only parameter to the magical `where()` method is the expected value of the column. Let's take a look at another example.

Example 91: Another magic where query.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::whereTitle('North')->get();
});
```

Fetch all albums with a `title` column value of `North`. Here's the result of the query.

Example 92: Output.

```
[
    {
        "id": 4,
        "title": "North",
        "artist": "Something Corporate",
        "genre": "Piano Rock",
        "year": 2002
    }
]
```

Wonderful! There's our album! Be sure to remember the magical `where()` query if you ever find yourself wanting to retrieve ORM instances by column values.

Query Scopes

Query scopes can be very useful if you find yourself repeating the same queries over and over again. Let's begin with an example. Remember those two albums with names beginning with triple periods? '...Is A Real Boy' and '...Anywhere But Here'. Let's imagine that fetching albums that begin with a triple period is a common action within our application.

Let's see. We could query for the albums every time, like this.

Example 93: Where query.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::where('title', 'LIKE', '...%')->get();
});
```

That would be terribly repetitive, though, wouldn't it? We don't want to be repeating ourselves. Why don't we use a Query Scope? Let's get started. First, let's revisit our `Album` model. Right now it looks like this.

Example 94: Our model.

```
<?php

// app/Album.php

namespace App;

use Eloquent;

class Album extends Eloquent
{
    public $timestamps = false;
}
```

Let's add a new method to this model. Now our model looks like this.

Example 95: Our model with a scope.

```
<?php

// app/Album.php

namespace App;

use Eloquent;

class Album extends Eloquent
{
    public $timestamps = false;

    public function scopeTriplePeriod($query)
    {
        return $query->where('title', 'LIKE', '...%');
    }
}
```

We've added the `scopeTriplePeriod()` method to our model. It's a special method with a specific function. It will help us reuse common queries. All scope methods begin with the word scope and then an identifier. The method will accept a single parameter, a `$query` object. This object can be used to construct queries like the ones that we discovered in the previous sections. In our example, we use the `return` statement to return the value from our `where()` method. The `where()` method takes the same shape as our previous example.

Now let's switch back to our routing file. Let's alter our existing query. Here's the new routed Closure.

Example 96: Using our scope.

```
<?php

// app/Http/routes.php

Route::get('/', function() {
    return \App\Album::triplePeriod()->get();
});
```

We can change our `where()` query to instead call the `triplePeriod()` scope. Next, we simply call `get()` on the result to retrieve the results. Note that the `scope` part of the method name is not included. Be sure to leave that out of you method calls! Let's take a look at the result.

Example 97: Output.

```
[
    {
        "id": 5,
        "title": "...Anywhere But Here",
        "artist": "The Ataris",
        "genre": "Punk Rock",
        "year": 1997
    },
    {
        "id": 6,
        "title": "...Is A Real Boy",
        "artist": "Say Anything",
        "genre": "Indie Rock",
        "year": 2006
    }
]
```

Great! That's the result set we were expecting. Use as many scopes as you need to reduce your repetition.

22. Eloquent Collections

I love collections. I have far too many of them. As a kid, I used to collect egg cups and transformer toys. As an adult, I collect video games and manga comics. Nerdy collections are the best.

Laravel has its own collections too. It has a collection of loving fans, eager to help each other and develop the community. It has a collection of amazing developers contributing to it. It has a collection of stories about where the name came from, most of them false. It also has Eloquent Collections.

The Collection Class

Eloquent collections are an extension of Laravel's `Collection` class with some handy methods for dealing with query results. The `Collection` class itself, is merely a wrapper for an array of objects but has a bunch of other interesting methods to help you pluck items out of the array.

In Laravel three, an array of model instances was returned from any query method that is used to provide multiple results. However, in Laravel four and above you will instead receive a `Collection` of model instances. Don't worry; you can still iterate through a collection of results with the variety of loops that PHP offers because it inherits some properties of an array, but because the collection is a class, and not native type, there are also methods available on the object.

Let's take a look at the available methods of the `Collection` class. For our sample data we will use the `albums` table from the previous chapter, and assume that the collection is the result of a call to `Album::all()`, like this.

Example 01: Retrieve all values.

```
1  <?php
2
3  // app/Http/routes.php
4
5  Route::get('/', function () {
6      $collection = \App\Album::all();
7  });
```

Collection Methods

Let's look at some of the methods available to the `Collection` class. Some of the methods are related to inserting and retrieving elements by their key. However, in the case of Eloquent results, the keys do not match the primary key of the tables that the model instances represent, and so these methods will not be very useful to us. Instead, I will cover the methods that do have some use.

All

The `all()` method can be used to get hold of the internal array utilized by the `Collection` object. What this means is that if you want your results to be identical to those supplied by Laravel 3, then just call the `all()` method and you will have your instance array.

Let's `var_dump()` the result.

Here's our code.

Example 02: Retrieve all albums.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();
    var_dump($collection->all());
});
```

And here is the result.

Example 03: Output.

```
array (size=6)
  0 =>
    object(Album)[127]
      public 'timestamps' => boolean false
      protected 'connection' => null
      protected 'table' => null
      protected 'primaryKey' => string 'id' (length=2)
      protected 'perPage' => int 15
      public 'incrementing' => boolean true
```

```
      protected 'attributes' =>
        array (size=5)
          'id' => int 1
          'title' => string 'Some Mad Hope' (length=13)
          'artist' => string 'Matt Nathanson' (length=14)
          'genre' => string 'Acoustic Rock' (length=13)
          'year' => int 2007
      protected 'original' =>
        array (size=5)
          'id' => int 1
          'title' => string 'Some Mad Hope' (length=13)
          'artist' => string 'Matt Nathanson' (length=14)
          'genre' => string 'Acoustic Rock' (length=13)
          'year' => int 2007
      protected 'relations' =>
        array (size=0)
          empty
      protected 'hidden' =>
        array (size=0)
          empty
      protected 'visible' =>
        array (size=0)
          empty
      protected 'fillable' =>
        array (size=0)
          empty
      protected 'guarded' =>
        array (size=1)
          0 => string '*' (length=1)
      protected 'touches' =>
        array (size=0)
          empty
      protected 'with' =>
        array (size=0)
          empty
      public 'exists' => boolean true
      protected 'softDelete' => boolean false
  1 =>
    object(Album)[128]
    ... TONNES MORE INFO ...
```

As you can see, we have an array of our Eloquent model instances.

First

The `first()` method of the collection can be used to retrieve the first element in the set. This will be the first element contained within the collections internal array.

Let's give it a go.

Example 04: Retrieve the first item of the collection.

```php
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();
    var_dump($collection->first());
});
```

Now we visit the `/` URI to view the result. What do you expect it will be?

Example 05: Output.

```
object(Album)[127]
  public 'timestamps' => boolean false
  protected 'connection' => null
  protected 'table' => null
  protected 'primaryKey' => string 'id' (length=2)
  protected 'perPage' => int 15
  public 'incrementing' => boolean true
  protected 'attributes' =>
    array (size=5)
      'id' => int 1
      'title' => string 'Some Mad Hope' (length=13)
      'artist' => string 'Matt Nathanson' (length=14)
      'genre' => string 'Acoustic Rock' (length=13)
      'year' => int 2007
  protected 'original' =>
    array (size=5)
      'id' => int 1
      'title' => string 'Some Mad Hope' (length=13)
      'artist' => string 'Matt Nathanson' (length=14)
      'genre' => string 'Acoustic Rock' (length=13)
      'year' => int 2007
  protected 'relations' =>
```

```
    array (size=0)
      empty
  protected 'hidden' =>
    array (size=0)
      empty
  protected 'visible' =>
    array (size=0)
      empty
  protected 'fillable' =>
    array (size=0)
      empty
  protected 'guarded' =>
    array (size=1)
      0 => string '*' (length=1)
  protected 'touches' =>
    array (size=0)
      empty
  protected 'with' =>
    array (size=0)
      empty
  public 'exists' => boolean true
  protected 'softDelete' => boolean false
```

That's right! It's a single model instance that represents one of our albums. The first row that we inserted into the table. Note that it's only the first row because we used the `all()` method as part of the query. If we had used a different query, then the array in our result set might be of a different order and then a call to `first()` could yield a different result.

Last

This one should be obvious. The `first()` method was used to retrieve the first value contained within the collections internal array; this means that the `last()` method must retrieve the last item of the array.

Let's prove this theory.

Example 06: Retrieve the last item in the set.

```php
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();
    var_dump($collection->last());
});
```

Here's the result from the / URI.

Example 07: Output.

```
object(Album)[138]
  public 'timestamps' => boolean false
  protected 'connection' => null
  protected 'table' => null
  protected 'primaryKey' => string 'id' (length=2)
  protected 'perPage' => int 15
  public 'incrementing' => boolean true
  protected 'attributes' =>
    array (size=5)
      'id' => int 6
      'title' => string '...Is A Real Boy' (length=16)
      'artist' => string 'Say Anything' (length=12)
      'genre' => string 'Indie Rock' (length=10)
      'year' => int 2006
  protected 'original' =>
    array (size=5)
      'id' => int 6
      'title' => string '...Is A Real Boy' (length=16)
      'artist' => string 'Say Anything' (length=12)
      'genre' => string 'Indie Rock' (length=10)
      'year' => int 2006
  protected 'relations' =>
    array (size=0)
      empty
  protected 'hidden' =>
    array (size=0)
      empty
  protected 'visible' =>
```

```
    array (size=0)
      empty
  protected 'fillable' =>
    array (size=0)
      empty
  protected 'guarded' =>
    array (size=1)
      0 => string '*' (length=1)
  protected 'touches' =>
    array (size=0)
      empty
  protected 'with' =>
    array (size=0)
      empty
  public 'exists' => boolean true
  protected 'softDelete' => boolean false
```

Great, that's the last album contained within the internal array. It's also the last row of the database, but that just depends on the query we use to retrieve the collection.

Shift

The `shift()` method is similar to the `first()` method. It will retrieve the first value within the collections internal array. However, unlike the `first()` method, the `shift()` method will also remove that value from the array. Let's prove this with a little test.

Example 08: Retrieve first item and remove from set.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();
    var_dump(count($collection));
    var_dump($collection->shift());
    var_dump(count($collection));
});
```

Our test will display the number of elements within the collection using the PHP `count()` method before and after we have `shift()`ed a value. Remember that the

collection inherits many properties of an array, this allows the `count()` method to act upon it.

Let's take a look at the result of our test.

Example 09: Output.

```
int 6
object(Album)[127]
  public 'timestamps' => boolean false
  protected 'connection' => null
  protected 'table' => null
  protected 'primaryKey' => string 'id' (length=2)
  protected 'perPage' => int 15
  public 'incrementing' => boolean true
  protected 'attributes' =>
    array (size=5)
      'id' => int 1
      'title' => string 'Some Mad Hope' (length=13)
      'artist' => string 'Matt Nathanson' (length=14)
      'genre' => string 'Acoustic Rock' (length=13)
      'year' => int 2007
  protected 'original' =>
    array (size=5)
      'id' => int 1
      'title' => string 'Some Mad Hope' (length=13)
      'artist' => string 'Matt Nathanson' (length=14)
      'genre' => string 'Acoustic Rock' (length=13)
      'year' => int 2007
  protected 'relations' =>
    array (size=0)
      empty
  protected 'hidden' =>
    array (size=0)
      empty
  protected 'visible' =>
    array (size=0)
      empty
  protected 'fillable' =>
    array (size=0)
      empty
  protected 'guarded' =>
    array (size=1)
      0 => string '*' (length=1)
```

```
  protected 'touches' =>
    array (size=0)
      empty
  protected 'with' =>
    array (size=0)
      empty
  public 'exists' => boolean true
  protected 'softDelete' => boolean false
int 5
```

As you can see, we receive our `Album` model instance. It's the same as the one we obtained using the `first()` method. However, if you look at the two integer values, you will notice that the size of the array has decreased. This is because the instance has not only been returned from the method but also removed from the array.

Pop

Pop is a genre of music that has been around for decades and is used to encourage the consumption of alcohol, or the wild dreams of teenagers.

Oh yes, it's also a method on the Eloquent model instances collection. It works in a similar way to the `shift()` method in that it will return the value from the end of the internal array, and remove it. Let's trap its result with our test.

Example 10: Retrieve last item from set and remove it.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();
    var_dump(count($collection));
    var_dump($collection->pop());
    var_dump(count($collection));
});
```

Here is the result.

Example 11: Output.

```
int 6
object(Album)[138]
  public 'timestamps' => boolean false
  protected 'connection' => null
  protected 'table' => null
  protected 'primaryKey' => string 'id' (length=2)
  protected 'perPage' => int 15
  public 'incrementing' => boolean true
  protected 'attributes' =>
    array (size=5)
      'id' => int 6
      'title' => string '...Is A Real Boy' (length=16)
      'artist' => string 'Say Anything' (length=12)
      'genre' => string 'Indie Rock' (length=10)
      'year' => int 2006
  protected 'original' =>
    array (size=5)
      'id' => int 6
      'title' => string '...Is A Real Boy' (length=16)
      'artist' => string 'Say Anything' (length=12)
      'genre' => string 'Indie Rock' (length=10)
      'year' => int 2006
  protected 'relations' =>
    array (size=0)
      empty
  protected 'hidden' =>
    array (size=0)
      empty
  protected 'visible' =>
    array (size=0)
      empty
  protected 'fillable' =>
    array (size=0)
      empty
  protected 'guarded' =>
    array (size=1)
      0 => string '*' (length=1)
  protected 'touches' =>
    array (size=0)
      empty
  protected 'with' =>
```

```
    array (size=0)
      empty
  public 'exists' => boolean true
  protected 'softDelete' => boolean false
int 5
```

We receive the last element of the array, and the result from our `count()` methods suggest that the length of the array has been decreased. This is because the value we received was removed from the internal array.

Each

If you have used the 'Underscore' library for Javascript or PHP, then you will be familiar with the next few methods. Instead of creating a `foreach()` loop to iterate over our results, we can instead pass a Closure to the `each()` method.

This is best described using an example.

Example 12: Iterate each item in the set with a callback.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();
    $collection->each(function ($album) {
        var_dump($album->title);
    });
});
```

Our Closure accepts a parameter that will be a placeholder for the current object within the iteration. This closure is then passed to the `each()` method. For each iteration, we will dump the value of the model's 'title' column.

Let's examine the result.

Example 13: Output.

```
string 'Some Mad Hope' (length=13)
string 'Please' (length=6)
string 'Leaving Through The Window' (length=26)
string 'North' (length=5)
string '...Anywhere But Here' (length=20)
string '...Is A Real Boy' (length=16)
```

Brilliant! Those are our album titles as expected.

Map

The `map()` function works in a similar way to the `each()` method. However, it can be used to iterate and work with our collection elements, returning a new collection as a result.

Let's imagine that we want to prefix all of our album titles with `An ode to a fair panda:` . We can do this using the `map()` function.

Example 14: Use a callback to mutate our set.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();

    $new = $collection->map(function ($album) {
        return 'An ode to a fair panda: '.$album->title;
    });

    var_dump($new);
});
```

First, we ensure that the value of `Collection::map()` method is assigned to a variable. Then we iterate the collection in the same manner as the `each()` method, but this time we return each value that we wish to be present within the new collection.

Here's the result.

Example 15: Output.

```
object(Illuminate\Database\Eloquent\Collection)[117]
  protected 'items' =>
    array (size=6)
      0 => string 'An ode to a fair panda: Some Mad Hope' (length=37)
      1 => string 'An ode to a fair panda: Please' (length=30)
      2 => string 'An ode to a fair panda: Leaving Through The Window' (length=5\
0)
      3 => string 'An ode to a fair panda: North' (length=29)
      4 => string 'An ode to a fair panda: ...Anywhere But Here' (length=44)
      5 => string 'An ode to a fair panda: ...Is A Real Boy' (length=40)
```

Now we have a collection of strings that we built using our iterative `map()` method.

Filter

The `filter()` method can be used to reduce the number of elements contained within the resulting collection by using a Closure. If the result of the Closure is boolean `true`, then the current element of the iteration will be given to the resulting collection. If the Closures iteration returns a `false` or nothing at all, then that element will not exist within the new collection.

This might be easier to understand with an example.

Example 16: Reduce a set.

```php
 <?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();

    $new = $collection->filter(function ($album) {
        if ($album->artist == 'Something Corporate') {
            return true;
        }
    });

    var_dump($new);
});
```

We iterate the collection with the `filter()` method and our Closure. For each iteration, if the value of the `artist` column on our model instance is equal to 'Something Corporate' then we will return true. This indicates that the model instance should be present within the new collection.

Here's the result.

Example 17: Output.

```
object(Illuminate\Database\Eloquent\Collection)[117]
  protected 'items' =>
    array (size=2)
      2 =>
        object(Album)[135]
          public 'timestamps' => boolean false
          protected 'connection' => null
          protected 'table' => null
          protected 'primaryKey' => string 'id' (length=2)
          protected 'perPage' => int 15
          public 'incrementing' => boolean true
          protected 'attributes' =>
            array (size=5)
              'id' => int 3
              'title' => string 'Leaving Through The Window' (length=26)
              'artist' => string 'Something Corporate' (length=19)
              'genre' => string 'Piano Rock' (length=10)
              'year' => int 2002
          protected 'original' =>
            array (size=5)
              'id' => int 3
              'title' => string 'Leaving Through The Window' (length=26)
              'artist' => string 'Something Corporate' (length=19)
              'genre' => string 'Piano Rock' (length=10)
              'year' => int 2002
      3 =>
        object(Album)[136]
          public 'timestamps' => boolean false
          protected 'connection' => null
          protected 'table' => null
          protected 'primaryKey' => string 'id' (length=2)
          protected 'perPage' => int 15
          public 'incrementing' => boolean true
          protected 'attributes' =>
            array (size=5)
```

```
              'id' => int 4
              'title' => string 'North' (length=5)
              'artist' => string 'Something Corporate' (length=19)
              'genre' => string 'Piano Rock' (length=10)
              'year' => int 2002
          protected 'original' =>
            array (size=5)
              'id' => int 4
              'title' => string 'North' (length=5)
              'artist' => string 'Something Corporate' (length=19)
              'genre' => string 'Piano Rock' (length=10)
              'year' => int 2002
```

I shortened the results a little to save space, but you can clearly see that we have our two albums by 'Something Corporate.'

Sort

The `sort()` method can be used to sort the collection. It hands our Closure to the `uasort()` PHP method which uses integer values to represent a comparison between two values. Our closure receives two parameters, let's call them A and B. Then we apply the rules of our sorting to provide an integer result from our closure.

If A > B then we return 1. If A < B then we return -1. If A = B then we return 0.

Here's an example.

Example 18: Sort set using a callback.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();

    $collection->sort(function ($a, $b) {
        $a = $a->year;
        $b = $b->year;
        if ($a === $b) {
            return 0;
        }
        return ($a > $b) ? 1 : -1;
```

```
    });

    $collection->each(function ($album) {
        var_dump($album->year);
    });
});
```

We provide a closure with two parameters; these represent any two albums from the collection. First, we assign `$a` and `$b` to the `year` column to simplify our comparison code. If `$a` is equal to `$b` we return `0`. If `$a` is greater than `$b` we return `1`, otherwise `-1`.

This method is destructive. It alters the original collection. We iterate through the collection dumping the year values to show the now order. Here is the result.

Example 19: Output.

```
int 1993
int 1997
int 2002
int 2002
int 2006
int 2007
```

As you can see, our albums are now ordered by `year` in ascending order. Here's some homework for you. Try to modify the above example changing only a single character so that the album years are instead presented in descending order. I promise it can be done!

Reverse

The `reverse()` method can be used to `reverse()` the models contained within the internal array. Does this really require an example? Oh go on then, you are wonderfully handsome readers after all.

Example 20: Reverse collection values.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();

    $collection->each(function ($album) {
        var_dump($album->title);
    });

    $reverse = $collection->reverse();

    $reverse->each(function ($album) {
        var_dump($album->title);
    });
});
```

First, we iterate through all of the albums outputting their title. Next, we reverse the collection, assigning the now collection to the `$reverse` variable. We then iterate through the `$reverse` collection to see what has changed.

Here is the result.

Example 21: Output.

```
string 'Some Mad Hope' (length=13)
string 'Please' (length=6)
string 'Leaving Through The Window' (length=26)
string 'North' (length=5)
string '...Anywhere But Here' (length=20)
string '...Is A Real Boy' (length=16)


string '...Is A Real Boy' (length=16)
string '...Anywhere But Here' (length=20)
string 'North' (length=5)
string 'Leaving Through The Window' (length=26)
string 'Please' (length=6)
string 'Some Mad Hope' (length=13)
```

Great! Our collection has been reversed.

Merge

The `merge()` method can be used to combine two collections. The only parameter to the method is the collection that should be merged. Here's an example.

Example 22: Merge two or more collections.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $a = \App\Album::where('artist', '=', 'Something Corporate')
        ->get();
    $b = \App\Album::where('artist', '=', 'Matt Nathanson')
        ->get();

    $result = $a->merge($b);

    $result->each(function ($album) {
        echo $album->title.'<br />';
    });
});
```

In the above example, we perform two queries. Result set `$a` is a collection of albums by 'Something Corporate'. Set `$b` contains albums by 'Matt Nathanson'. We use the `merge()` method to merge the collections, and assign the result to a new collection named `$result`. Then we echo the album titles within an `each()` loop.

Here are the results.

Example 23: Output.

```
Leaving Through The Window
North
Some Mad Hope
Please
```

Slice

The `slice()` method is the equivalent of the PHP `slice()` function. It can be used to produce a subset of models using a collection offset. Confused? Take a look at this.

Example 24: Slice our collection.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();

    $sliced = $collection->slice(2, 4);

    $sliced->each(function($album)
    {
        echo $album->title.'<br />';
    });
});
```

We create a new collection using the `slice()` method. The first parameter is the position that the new subset is started. Here we are telling it to start slicing at the second element of the array. The second optional parameter is the length of the collection. We tell the `slice()` method that we want four elements, after the second element in the array.

Let's take a look at the result.

Example 25: Output.

```
Leaving Through The Window
North
...Anywhere But Here
...Is A Real Boy
```

Did you know that you can pass a negative value to `slice()`? For example:

Example 26: A negative slice.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();

    $sliced = $collection->slice(-2, 4);

    $sliced->each(function ($album) {
        echo $album->title.'<br />';
    });
});
```

By passing `-2` as the first parameter, we are telling it to start the collection two elements from the **end** of the collection. Here's the result.

Example 27: Output.

```
...Anywhere But Here
...Is A Real Boy
```

Wait, didn't we tell it to retrieve four models?

We did, however since we are positioned two elements from the end of the array, only two elements are available to retrieve. The `slice()` method does not wrap around the collection.

IsEmpty

The `isEmpty()` method can be used to check whether or not the container has elements within it. I bet you didn't see that coming! It accepts no value and returns a boolean result. Here's an example.

Example 28: Determine whether set is empty.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    // This query will return items.
    $a = \App\Album::all();

    // This query won't.
    $b = \App\Album::where('title', '=', 'foo')->get();

    var_dump($a->isEmpty());
    var_dump($b->isEmpty());
});
```

Aha, a cunning trap! We know that the first query will return results and that the second query will not. Let's dump the result of the `isEmpty()` method for both to see what we get.

Example 29: Output.

```
boolean false
boolean true
```

The first `isEmpty()` returns a boolean `false` because the array has elements within it. The second collection is empty, and the `isEmpty()` method returns a boolean `true`.

ToArray

The `toArray()` method can be used to return the internal array of the collection. Also, any elements within the array that can be transformed into an array, for example, objects, will be transformed during the process.

Example 30: Convert collection to array.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();
    var_dump( $collection->toArray() );
});
```

Here's our result.

Example 31: Output.

```
array (size=6)
  0 =>
    array (size=5)
      'id' => int 1
      'title' => string 'Some Mad Hope' (length=13)
      'artist' => string 'Matt Nathanson' (length=14)
      'genre' => string 'Acoustic Rock' (length=13)
      'year' => int 2007
  1 =>
    array (size=5)
      'id' => int 2
      'title' => string 'Please' (length=6)
      'artist' => string 'Matt Nathanson' (length=14)
      'genre' => string 'Acoustic Rock' (length=13)
      'year' => int 1993
  2 =>
    array (size=5)
      'id' => int 3
      'title' => string 'Leaving Through The Window' (length=26)
      'artist' => string 'Something Corporate' (length=19)
      'genre' => string 'Piano Rock' (length=10)
      'year' => int 2002
      ... and more ...
```

As you can see, not only has an array representing our collection been returned, but the model instances held within have also been transformed into arrays.

ToJson

The `toJson()` method will transform the collection into a JSON string that can be used to represent its contents. In the previous chapter, we discovered how to return collections directly from a routed closure or controller action to serve a JSON response. The `toString()` method allowing the collection to be transformed to JSON makes a call to the `toJson()` method internally.

Let's take a look at an example.

Example 32: Convert collection into a JSON string.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();
    var_dump( $collection->toJson() );
});
```

Here is the result from the above example.

Example 33: Output.

```
string '[{"id":1,"title":"Some Mad Hope","artist":"Matt Nathanson","genre":"Acou\
stic Rock","year":2007},{"id":2,"title":"Please","artist":"Matt Nathanson","genr\
e":"Acoustic Rock","year":1993},{"id":3,"title":"Leaving Through The Window","ar\
tist":"Something Corporate","genre":"Piano Rock","year":2002},{"id":4,"title":"N\
orth","artist":"Something Corporate","genre":"Piano Rock","year":2002},{"id":5,"\
title":"...Anywhere But Here","artist":"The Ataris","genre":"Punk Rock","year":1\
997},{"id":6,"title":"...Is A Real Boy","artist":"Say Anything","genre":"Indie R\
ock","year":2006}]' (length=570)
```

It's our entire collection represented by a JSON string.

Count

In an earlier example, I used the PHP `count()` method to count the number of model instances contained within the collection's internal array. How silly of me! I had completely forgotten about the `count()` method of the collection. In honesty, it does the same thing. Let me show you.

Example 34: Count the members of a collection.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $collection = \App\Album::all();
    var_dump( $collection->count() );
});
```

Here's the result. I bet you can't wait!

Example 35: Our new migration.

```
int 6
```

So the result is the same, which should we use? Whichever ever one suits us best. I leave the decision up to you. You are over 300 pages into the book by now and are old and wise. Still handsome, though. Well done you!

Best Practice

Some of the methods available to the collection are duplicates of those available on the query builder. For example, we can use the `first()` method when building eloquent queries to retrieve only a single model instance. We can also use the `first()` method of the collection to retrieve the first element contained within. Take a close look at the following example.

Example 36: Chaining first.

```
    \App\Album::all()->first();
    \App\Album::first();
```

These two lines both arrive at the same result. The first item within the `albums` table will be returned as a result. So which should we use?

Well, as always, the answer is "it depends". Yes, I know that nobody likes to hear that reply, but sometimes it's true. Let's take a look at two different scenarios.

In the first scenario, we wish to display the title of the first album stored in the database within one of our templates. Sure, not a problem! We can use the `first()` method on the Eloquent, query builder.

Example 37: Get the first model.

```
1  \App\Album::first();
```

In the second scenario, we wish to display a listing of all album titles, but also display the title of the first album within its little box. For example, the 'album of the year' section. Well, we could do something like this I suppose.

Example 38: Get the first album from the database.

```
1  $allAlbums = \App\Album::all();
2  $albumOfTheYear = \App\Album::first();
```

I'm sure that would do what we need, but this method will run two queries against the database. Increasing the number of queries sent to the database server is a way of rapidly decreasing application performance. It might not matter within our examples, but within your enterprise software, every split second wasted is money lost.

We could alleviate some stress upon the database server, and reduce the number of queries to one by shifting some of the query responsibilities to the collection.

Example 39: Get the first album from the collection.

```
1  $allAlbums = \App\Album::all();
2  $albumOfTheYear = $allAlbums->first();
```

We retrieve all of our albums as usual, but this time, we use the `first()` method on the collection to retrieve the album of the year. This results in only a single query.

Choosing whether to perform a query on the collection, or on the database is a matter of choice. Querying against the database will allow for faster and more complex searches that would result in high CPU / memory usage on the webserver if performed on the collection. Then again, using the collection efficiently can also benefit your application with fewer individual SQL queries.

Use your best judgment to decide when to use the collection helper methods. I trust you!

23. Eloquent Relationships

In the previous chapters, we have discovered how to represent the rows stored within our database table as objects. Class instances which represent a single row. This means that we have broken down our objects into their simplest form. Book, Fruit, BoyBand, whatever they happen to be.

Since these objects are now simple, if we wish to store data related to them we will need to create new objects and form a relationship between them. So what do we mean by the term 'relationship'? Well, I don't mean the hugs and kisses and sweaty bedsheets type of relationship. This kind of relationship is best explained with an example.

Let's take our Book model from the previous chapter. If we think about books for a moment, we soon come to the conclusion that they have to have been written by someone. I know that you would like to believe that I don't exist, but the sad truth is that I do. Somewhere out there there's a crazy British guy who is fanatical about Laravel. There are other authors out there too, not just me.

So we know that a book **belongs to** an author. There's our first relationship. The author has a 'link' to the book. In a way, the author identifies the book. It's not necessarily a property of the book, like its title for example. No, an author has its own set of properties. A name, a birthplace, a favorite pizza topping. It deserves to be its own model. An Author class.

So how do we link these two together? With relational databases, we can use 'foreign key's to identify relationships. These are usually integer columns.

Let's build two example tables.

books

Example 01: The books table.

```
+---------+-----------------------+
| id (PK) | name                  |
+---------+-----------------------+
| 1       | Code Sexy             |
| 2       | Code Dutch            |
| 3       | Code Smart            |
+---------+-----------------------+
```

Here's our Book table with three books held within. You will notice that each table has a unique integer primary key. This can be used to identify each row. Now let's take a look at the second table.

authors

Example 02: The authors table.

```
+---------+------------------------+
| id (PK) | name                   |
+---------+------------------------+
| 1       | Dayle Rees             |
| 2       | Matthew Machuga        |
| 3       | Shawn McCool           |
+---------+------------------------+
```

Here we have another table containing the names of three fantastically handsome developers. We will call this the Author table. Notice how each row once again has an integer primary key that can be used as a unique identifier?

Right, let's form a relationship between the two tables. This is a relationship between a Book and an Author; we will ignore co-authorship for now and assume that a book will only have a single author. Let's add a new column to the books table.

books

Example 03: Our new migration.

```
+---------+------------------------+-----------------+
| id (PK) | name                   | author_id (FK)  |
+---------+------------------------+-----------------+
| 1       | Code Sexy              | 2               |
| 2       | Code Dutch             | 3               |
| 3       | Code Smart             | 1               |
+---------+------------------------+-----------------+
```

We have added our first foreign key. This is the `author_id` column. Once again it's an integer value, but it isn't used to identify rows on this table. Instead, it's used to identify related rows on another table. A foreign key column is normally used to identify a primary key on an adjacent table, however in some circumstances, it can be used to identify other columns.

Do you see how the addition of a foreign key has created a relationship between the author and a book? Our `author_id` foreign key references the primary key within the Author table. Take a closer look at row three of the book table. Code Smart has an `author_id` of `1`. Now look at row `1` of the Author table and we will see `Dayle Rees`. This means that Code Smart was written by Dayle Rees. It's as simple as that.

> Why didn't we place the foreign key on the Author table?

Well, an Author could have many books, couldn't they? For example, take a look at this relationship.

authors

Example 04: Authors once again.

```
+---------+------------------------+
| id (PK) | name                   |
+---------+------------------------+
| 1       | Dayle Rees             |
| 2       | Matthew Machuga        |
| 3       | Shawn McCool           |
+---------------------------------+
```

books

Example 05: A new old book.

```
+---------+------------------------+-----------------+
| id (PK) | name                   | author_id (FK)  |
+---------+------------------------+-----------------+
| 1       | Code Sexy              | 2               |
| 2       | Code Dutch             | 3               |
| 3       | Code Smart             | 1               |
| 4       | Code Happy             | 1               |
+---------------------------------+-----------------+
```

Notice how Dayle Rees, I mean, me, notice how I... oh dear I've ruined this sentence. Notice how there are two books belonging to myself. Both 'Code Happy' and 'Code Smart' have an `author_id` value of `1`. This means that they were both written by Dayle Rees.

Now let's try to express the above example with the foreign key on the Author table instead.

authors

Example 06: Foreign keys on the authors table.

```
+---------+------------------------+---------------+-----------+
| id (PK) | name                   | book_one (FK) | book_two  |
+---------+------------------------+---------------+-----------+
| 1       | Dayle Rees             | 3             | 4         |
| 2       | Matthew Machuga        | 1             | null      |
| 3       | Shawn McCool           | 2             | null      |
+---------+------------------------+---------------+-----------+
```

books

Example 07: Remove foreign key from the books table.

```
+---------+------------------------+
| id (PK) | name                   |
+---------+------------------------+
| 1       | Code Sexy              |
| 2       | Code Dutch             |
| 3       | Code Smart            |
| 4       | Code Happy             |
+---------+------------------------+
```

As you can see, we have had to add some foreign keys to the Author table. Not only that but because some authors will not have two books, many of the columns will have `null` values contained within. What happens if we want our authors to have three or four books? We can't keep adding columns; our tables will start to look messy!

> Very well, we will keep the foreign key on the Book table.

Why don't we learn the names of these relationship types? It will be useful when we become to implement these relationships with Eloquent. Here we go.

We have the `author_id` field on the Book, which identified its single author. This means that it **belongs to** an Author. That's our first relationship type.

Book **belongs_to** Author

Relationships also have inverse variations. For example, if a Book **belongs_to** an Author, then this means that an Author **has_many** Books. We have learned the name of another relationship.

Author **has_many** Book

If instead the Author had only a single book but the `books` table contained the identifying primary key, then we would use the **has_one** relationship type instead.

Don't forget those three relationship types, but let's go ahead and move on to a fourth. We need another example. Hmm... How about a 'favorites' system? Where users can vote on books. Let's try to express this using the relationships we have already discovered.

User **has_many** Book

Book **has_many** User

The 'has many' is the relationship that created the favorite. We don't need an entire entity to express a favorite since it will have no attributes. When both the relationship and its inverse relationship are **has_many** then we will need to implement a new type of relationship. First of all, instead of saying **has_many**, we will say **belongs_to_many**. This way we won't confuse it with the other relationship. This new type of relationship forms a **many_to_many** relationship and requires an additional table.

Let's view the table structure.

users

Example 08: Our users table.

```
+---------+------------------------+
| id (PK) | name                   |
+---------+------------------------+
| 1       | Dayle Rees             |
| 2       | Matthew Machuga        |
| 3       | Shawn McCool           |
+---------------------------------+
```

books

Example 09: Our books table.

```
+---------+------------------------+
| id (PK) | name                   |
+---------+------------------------+
| 1       | Code Sexy              |
| 2       | Code Dutch             |
| 3       | Code Smart            |
| 4       | Code Happy             |
+---------------------------------+
```

book_user

Example 10: Our first pivot table.

```
+------------------------+
| id  | user_id | book_id |
+-----+---------+---------+
| 1   | 1       | 2       |
| 2   | 1       | 3       |
| 3   | 2       | 2       |
| 4   | 3       | 2       |
+-----+---------+---------+
```

Wait a minute, what's that third table?

Well spotted! That would be our join table, or pivot table, or lookup table, or intermediary table, or doing the hibbity-bibbity table. It has a lot of names. The Laravel documentation tends to refer to them as pivot tables, so I'm going to stick with that. Whenever you need a **many_to_many** relationship, you will find a need for a pivot table. It's the database table that links the two entities together by using two foreign keys to define the rows from the other tables.

Looking at the first two rows of the pivot table, we can see that the user 'Dayle Rees' has favorited both 'Code Dutch' and 'Code Smart.' We can also see that the users Matthew Machuga and Shawn McCool have both favorited Code Dutch.

There is an additional type of relationship known as a **polymorphic** relationship. Due to it's complex nature, and it's long name, we will cover it within a later chapter on advanced Eloquent tactics. We won't need it yet.

That's enough learning. It's time for different learning! Practical learning. Now that we have discovered the variety of relationships available let's learn how to implement them with Eloquent.

Implementing Relationships

Right, let's set the stage. First, we are going to need to construct some tables. Normally I'd create a new migration for each table, but to simplify the examples, I will put them all in one.

We are going to build a system using Artists, Albums, and Listeners. Listeners are simply users that like to listen to a variety of albums. Let's think about the relationships.

- An *Artist* **has_many** *Albums*.
- An *Album* **belongs_to** an *Artist*.

- A *Listener* **belongs_to_many** *Albums*.
- An *Album* **belongs_to_many** *Listeners*.

We have a simple relationship between an Artist and many Albums, and a many to many relationship between Listeners and Albums. So let's think about our table structure. If an Album **belongs_to** an Artist then the identifying foreign key for the relationship will exist on the Album table. The many to many relationship will require a pivot table.

We know that Eloquent requires a plural form of the model for its table name (unless we specify otherwise), but how do we name the pivot table? The default format for the pivot table is the singular form of the two related models separated by an _ underscore. The order of the instances should be alphabetical. This means that our pivot table is called `album_listener`.

All foreign key columns follow a similar naming convention. The singular form of the related model appended with `_id`. Our pivot table will contain both `album_id` and `listener_id`. The foreign keys will be unsigned, and we will use the `references()` and `on()` schema builder methods to reference other tables.

Let's look at the constructed migration, and then we will examine anything that's new in more detail.

Example 11: Create our example tables.

```
<?php

use Illuminate\Database\Migrations\Migration;

// database/migrations/2016_03_26_130751_create_tables.php

class CreateTables extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('artists', function ($table) {
            $table->increments('id');
            $table->string('name', 64);
            $table->timestamps();
        });
```

```
21
22          Schema::create('albums', function ($table) {
23              $table->increments('id');
24              $table->string('name', 64);
25              $table->integer('artist_id')->unsigned();
26              $table->foreign('artist_id')->references('id')->on('artists');
27              $table->timestamps();
28          });
29
30          Schema::create('listeners', function ($table) {
31              $table->increments('id');
32              $table->string('name', 64);
33              $table->timestamps();
34          });
35
36          Schema::create('album_listener', function ($table) {
37              $table->integer('album_id')->unsigned();
38              $table->foreign('album_id')->references('id')->on('albums');
39              $table->integer('listener_id')->unsigned();
40              $table->foreign('listener_id')->references('id')->on('listeners');
41          });
42      }
43
44      /**
45       * Reverse the migrations.
46       *
47       * @return void
48       */
49      public function down()
50      {
51          Schema::drop('artists');
52          Schema::drop('albums');
53          Schema::drop('listeners');
54          Schema::drop('album_listener');
55      }
56  }
```

The schema builder entries for the Album, Artist, and Listener are typical of Eloquent model schema definitions. However, we have yet to construct a pivot table. We simply create a `album_listener` table and add two integer fields to act as foreign keys. We don't need timestamps or a primary key since this table simply acts as a 'join' between the two model instances.

Time to create our Eloquent models. Let's start with the `Artist`.

Example 12: Our Artist model.

```
<?php

// app/Artist.php

namespace App;

use Eloquent;

class Artist extends Eloquent
{
    // Artist __has_many__ Album
    public function albums()
    {
        return $this->hasMany(Album::class);
    }
}
```

Here's something new! Inside our Eloquent model definition, we have a relationships method. Let's examine this a little more closely.

Example 13: Define relationships as public methods.

```
public function albums()
```

The name of the public method doesn't require a strict format. It will serve as a nickname that we can use to refer to the relationship. This method could have just as easily been called `relatedAlbums()`.

Example 14: A has many relationship.

```
return $this->hasMany(Album::class);
```

Within the relationship method, we return the result of the `$this->hasMany()` method. This is one of the many relationship methods that are inherited from the Eloquent base class. The first parameter to the relationship method is the full name of the model to be related. If we decide to namespace our `Album` model later, we will need to insert the full namespaced class as a parameter.

If our foreign key on the `Album` table is named differently to the default naming scheme of `artist_id` then we can specify the alternative name of the column as an optional second parameter to this method. For example:

Example 15: Relationship with alternate foreign key name.

```
return $this->hasMany('Album', 'the_related_artist');
```

So what exactly is being returned? Well, we don't worry about that for now! Let's finish creating our model definitions first. Next, we have the `Album` model.

Example 16: Our Album model.

```
<?php

// app/Album.php

namespace App;

use Eloquent;

class Album extends Eloquent
{
    // Album __belongs_to__ Artist
    public function artist()
    {
        return $this->belongsTo(Artist::class);
    }

    // Album __belongs_to_many__ Listeners
    public function listeners()
    {
        return $this->belongsToMany(Listener::class);
    }
}
```

The `Album` table has two relationship methods. Once again, the names of our public methods aren't important. Let's look at the content of the first method.

Example 17: The belongs to relationship.

```
return $this->belongsTo(Artist::class);
```

Since the foreign key exists in this table, we use the `$this->belongsTo()` method to state that the `Album` model is related to an `Artist`. The first parameter is once again the related model name, and once more we can provide an optional second parameter to use an alternative column name.

The second method forms one side of our many to many relationship. Let's take a closer look.

Example 18: The belongs to many relationship.

```
return $this->belongsToMany('Listener');
```

The `$this->belongsToMany()` method informs Eloquent that it should look at a pivot table for related models. The first parameter is, once again, the name of the related model including namespace if present. This time, we have a different set of optional parameters. Let's construct another example.

Example 19: Define a belongs to many relationship with optional parameters.

```
return $this->belongsToMany('Listener', 'my_pivot_table', 'first', 'second');
```

The second, optional parameter is the name of the pivot table to use for related objects. The third and fourth parameters are used to identify alternate naming schemes for the two foreign keys that are used to related our objects within the pivot table.

We have one model left. Let's take a look at the Listener.

Example 20: Our Listener model.

```
<?php

// app/Listener.php

namespace App;

use Eloquent;

class Listener extends Eloquent
{
    // Listener __belongs_to_many__ Album
    public function albums()
    {
        return $this->belongsToMany(Album::class);
    }
}
```

The listener forms the inverse side... well is it inverse? The listener forms the *other* side of the many to many relationship. Once again we can use the `$this->belongsToMany()` method to construct the relationship.

Right then! Our models have been created. Let's create some model instances.

Relating and Querying

First, let's create an Artist and an Album, and an association between the two. I'll be using my `/` routed closures as I believe that they are a nice simply way to demo code.

Example 21: Associate an artist and an album.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $artist = new \App\Artist;
    $artist->name = 'Eve 6';
    $artist->save();

    $album = new \App\Album;
    $album->name = 'Horrorscope';
    $album->artist()->associate($artist);
    $album->save();

    return View::make('hello');
});
```

Wait, what's this new line?

Example 22: Associate models.

```
$album->artist()->associate($artist);
```

Let's break the method chain down. Here's the first method.

Example 23: Access a relationship.

```
$album->artist();
```

Do you remember that method? That's the relationship method that we created on the `Album` object. So what exactly does it return? The method returns an instance of the Eloquent query builder, just like the one that we used in a previous chapter.

However, this query builder instance will have some constraints placed on it for us. The current set of results represented by the query builder will be the related `Artists` (in this case, a collection of one) to our `Album`.

It's the same as the following.

Example 24: Relationship method translation.

```
1 Artist::where('album_id', '=', __OUR__CURRENT__ALBUM__);
```

That's handy, right? We could chain on more queries if we wanted to. For example:

Example 25: Add a constraint to a relationship.

```
1 $album->artist()->where('genre', '=', 'rock')->take(2);
```

Be sure to remember that the query builder required a trigger method at the end of the join to return a result collection. Like this:

Example 26: Constraint and limit a relationship.

```
1 $album->artist()->where('genre', '=', 'rock')->take(2)->get();
```

This also means that we can retrieve a full collection of related artists by performing the following.

Example 27: Fetch all related models.

```
1 $album->artist()->get();
```

Or, a single instance using the `first()` trigger method.

Example 28: Fetch a single related model.

```
1 $album->artist()->first();
```

Oh, the flexibility! Right, let's take a look at our earlier example.

Example 29: Associate an artist to an album.

```
1 $album->artist()->associate($album);
```

> I haven't seen associate before!

Don't panic! Just because it's new doesn't mean it's evil. Only spiders are evil. Also, Kevin Bacon.

The associate method is a helper method available on relationships to create the relation between two objects. It will update the foreign key for the relationship accordingly.

This means that by passing our `$artist` instance to the `associate()` method we will have updated the `artist_id` column of our `Album` table row with the ID of the `Artist`. If we wanted to, we could also handle the foreign key directly.

Here's an example.

Example 30: Associate models.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $artist = new \App\Artist;
    $artist->name = 'Eve 6';
    $artist->save();

    $album = new \App\Album;
    $album->name = 'Horrorscope';
    $album->artist_id = $artist->id;
    $album->save();

    return View::make('hello');
});
```

Both code snippets would have the same result, and a relationship will have been created between the two objects.

Don't forget to `save()` the model that you intend to relate before passing it to the `associate()` method. The reason for this is that the model instance will require a primary key to create the relation, and the model will only receive a primary key value when it has been saved.

Relating objects that form many to many relationships is handled in a slightly different way. Let's take a closer look by introducing the `Listener` model.

Example 31: Relate models.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $artist = new \App\Artist;
    $artist->name = 'Eve 6';
    $artist->save();

    $album = new \App\Album;
    $album->name = 'Horrorscope';
    $album->artist()->associate($artist);
```

```
13      $album->save();
14
15      $listener = new \App\Listener;
16      $listener->name = 'Naruto Uzumaki';
17      $listener->save();
18      $listener->albums()->save($album);
19
20      return View::make('hello');
21  });
```

Let's focus on this bit:

Example 32: Create a relationship with a new listener.

```
1  <?php
2
3  $listener = new \App\Listener;
4  $listener->name = 'Naruto Uzumaki';
5  $listener->save();
6  $listener->albums()->save($album);
```

After we have populated our new `Listener` model instance, we must `save()` it. This is because we need it to have a primary key before we can create an entry in our pivot table.

This time, instead of using the `associate()` method on our relationship, we instead use the `save()` method, passing the object to relate as the first parameter. The effect is the same, only, this time, the pivot table will be updated to define the relationship.

If you would like to associate a model with its primary key directly, you can use the `attach()` method which accepts the primary key as the first parameter. For example.

Example 33: Create a relationship.

```
1  $album->artist()->attach(2);
```

To remove a relationship between two objects you can use the `detach()` method on the relationship. Just pass either a primary key or an object as the first parameter.

For example:

Example 34: Remove a relationship.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
    $album = \App\Album::find(5);
    $listener = \App\Listener::find(2);
    $album->listeners()->detach($listener);

    return View::make('hello');
});
```

We can remove all associations for an object that has a many to many relationship by calling the `detach()` method with no parameter on the relationship method.

Here's an example.

Example 35: Detach a record.

```
$album->listeners()->detach();
```

Now the album will have no related listeners.

Eloquent is a beautiful and vast topic full of many amazing features, but I don't want to burden you with all of them right now. It will only cause you to forget some of the basics that we have just learned. Let's keep it simple for now and move onto something new!

24. Validation

Don't trust your users. If there's one thing that I have learned working in the IT industry, it's that if you have a point of weakness in your application, then your users will find it. Honestly, they are evil little blighters! Trust me when I say that they will exploit it. Let's not give them the opportunity. We'll use validation to ensure that we always receive good input.

What do we mean by good input? Let's have a look at an example.

Simple Validation

Let's say that we have an HTML form that is used to collect registration information for our application. In fact, it's always great to have a little review session. We'll create the form with the Blade templating engine.

Here's our view.

Example 01: A blade form.

```
<!-- resources/views/form.blade.php -->

<h1>Registration form for our club!</h1>

<form action="{{ url('registration') }}" method="POST">

    {{-- CSRF Token. ---------------------------}}
    {{ csrf_field() }}

    {{-- Username field. ------------------------}}
    <label for="username">Username</label>
    <input type="text" name="username" />

    {{-- Email field. ---------------------------}}
    <label for="email">Email</label>
    <input type="text" name="email" />

    {{-- Password field. ------------------------}}
    <label for="password">Password</label>
    <input type="password" name="password" />
```

```
21
22      {{-- Password confirmation field. -----------}}
23      <label for="password_confirmation">Password confirmation</label>
24      <input type="password" name="password_confirmation" />
25
26      {{-- Form submit button. --------------------}}
27      <input type="submit" value="Register" />
28
29  </form>
```

Woah, how beautiful is that view?

> That is one beautiful view.

Excellent, I'm glad to see that you're still mentally conditioned from reading Code Happy.

You can see that our registration form is targeting the `/registration` route and will by default use the `POST` request verb.

We have a CSRF token hidden field, a text field for a username, an email field for a user's email address, and two password fields to collect a password and a password confirmation. At the bottom of the form, we have a submit button that we can use to send the form on its merry way.

You may have noticed the blade comments I have added to the source. This is a habit of mine which I find it makes it easier to browse for the field I want if I have to return to the form. You can feel free to do the same, but if you don't want to, then don't worry about it! Code your way!

Right, now we are going to need a couple of routes to display and handle this form. Let's go ahead and add them into our `routes.php` file now.

Example 02: Form routes.

```
1  <?php
2
3  // app/Http/routes.php
4
5  /**
6   * Show the registration form.
7   */
8  Route::get('/', function () {
9      return view('form');
```

```
});

/**
 * Handle a submitted registration form.
 */
Route::post('/registration', function (Request $request) {
    $formData = $request->all();

    // Handle the form...
});
```

We have a `GET /` route that will be used to display our form and a `POST /registration` route to handle its submission.

In the form handler route, we have collected all the data from the form, but we can't use it yet. Why? Let's learn by doing.

Go ahead and load up the `/` route and you should see the form. Right. Don't fill it in! Just hit the 'Register' button. The screen will go blank as the second route is triggered and our form has posted blank information. If we were to use the blank information, then it could cause severe problems for our application or even a security vulnerability. This is bad data.

We can avoid this hassle by implementing validation to ensure that our data, is instead, good data. Before we perform the validation, we first need to provide a list of validation constraints. We can do this in the form of an array. Are you ready? Great, let's take a look.

Example 03: Validation rules array.

```
<?php

// app/Http/routes.php

/**
 * Show the registration form.
 */
Route::get('/', function () {
    return view('form');
});

/**
 * Handle a submitted registration form.
 */
```

```
Route::post('/registration', function (Request $request) {
    $formData = $request->all();

    $rules = [
        'username' => 'alpha_num'
    ];
});
```

Right, let's start small. A set of validation rules takes the form of an associative array. The array key represents the field that is being validated. The array value will consist of one or many rules that will be used to validate. We will start by looking at a single validation constraint on a single field.

Example 04: Validation rules.

```
$rules = [
    'username' => 'alpha_num'
];
```

In the above example, we wish to validate that the 'username' field conforms to the `alpha_num` validation rule. The `alpha_num` rule can be used to ensure that a value consists of only alphanumeric characters.

Let's set up the validation object to make sure our rule is working correctly. To perform validation within Laravel, we first need to create an instance of the `Validation` object. Here we go.

Example 05: Our first validator.

```
<?php

// app/Http/routes.php

/**
 * Show the registration form.
 */
Route::get('/', function () {
    return view('form');
});

/**
 * Handle a submitted registration form.
 */
```

```
Route::post('/registration', function (Request $request) {
    $formData = $request->all();

    $rules = [
        'username' => 'alpha_num'
    ];

    // Create a new validator instance.
    $validator = Validator::make($formData, $rules);
});
```

We can use the `Validator::make()` method to create a new instance of our validator. The first parameter to the `make()` method is an array of data that will be validated. In this example, we intend to validate our request data, but we could just as easily validate any other array of data. The second parameter to the method is the set of rules that will be used to validate the data.

Now that our validator instance has been created, we can use it to check whether or not the data we provided conforms to the validation constraints that we have provided. Let's set up an example.

Example 06: Perform validation.

```
<?php

// app/Http/routes.php

/**
 * Show the registration form.
 */
Route::get('/', function () {
    return view('form');
});

/**
 * Handle a submitted registration form.
 */
Route::post('/registration', function (Request $request) {
    $formData = $request->all();

    $rules = [
        'username' => 'alpha_num'
    ];
```

```
21
22      // Create a new validator instance.
23      $validator = Validator::make($formData, $rules);
24
25      if ($validator->passes()) {
26
27          // Normally we would do something with the data.
28
29          return 'Data was saved.';
30      }
31
32      return redirect('/');
33  });
```

To test the result of the validation, we can use the `passes()` method on the validator instance. This method will return a boolean response to show whether or not the validation has passed. A `true` response indicates that the data conforms to all of the validation rules. A `false` indicates that the data does not meet the validation requirements.

We have used an `if` statement in the above example to decide whether to store the data or to redirect to the entry form. Let's test this by visiting the `/` URI.

First, enter the value 'johnzoidberg' into the username field and hit the submit button. We know that the username 'johnzoidberg' consists of alphanumeric characters so that the validation will pass. We are presented with the following sentence.

Example 07: Success.

```
1  Data was saved.
```

Great! That's what we expected. Let's invert the result of the validation. Go ahead and visit the `/` URI once again. This time, enter the value '!!!'. We know that an exclamation mark is not an alphabetical or numerical character, so the validation should fail. Hit the submit button, and we should be redirected back to the registration form.

I love that `passes()` method; it's really useful. The only problem is that it's a little optimistic. The world isn't a perfect place, let's be honest with ourselves. We aren't all Robert Downey Jr. Let's allow ourselves to be a little more pessimistic shall we? Excellent! Wait. Satisfactory. Let's try the `fails()` method instead.

Example 08: Less happy validation.

```
<?php

// app/Http/routes.php

/**
 * Show the registration form.
 */
Route::get('/', function () {
    return view('form');
});

/**
 * Handle a submitted registration form.
 */
Route::post('/registration', function (Request $request) {
    $formData = $request->all();

    $rules = [
        'username' => 'alpha_num'
    ];

    // Create a new validator instance.
    $validator = Validator::make($formData, $rules);

    if ($validator->fails()) {
        return redirect('/');
    }

    // Normally we would do something with the data.
    //
    return 'Data was saved.';
});
```

The `fails()` method returns the boolean opposite to the `passes()` method. How wonderfully pessimistic! Feel free to use whichever method suits your current mood. If you like, you could also write about your feelings using PHP comments.

Some validation rules can accept parameters. Let's swap out our `alpha_num` rule for the `min` one. Here's the source.

Example 09: Validation rules with parameters.

```php
<?php

// app/Http/routes.php

/**
 * Show the registration form.
 */
Route::get('/', function () {
    return view('form');
});

/**
 * Handle a submitted registration form.
 */
Route::post('/registration', function (Request $request) {
    $formData = $request->all();

    $rules = [
        'username' => 'min:3'
    ];

    // Create a new validator instance.
    $validator = Validator::make($formData, $rules);

    if ($validator->passes()) {

        // Normally we would do something with the data.

        return 'Data was saved.';
    }

    return redirect('/');
});
```

The `min` validation rule ensures that the value is greater than or equal to the parameter provided. The parameter is provided after the colon `:`. This validation rule is a little special. It will react differently depending on the data that has been provided.

For example, on a string value, our parameter '3' will ensure that the value is at least three characters long. On a numerical value, it will make sure that the value is mathematically greater than or equal to our parameter. Finally, on uploaded files,

the `min` validation rule will ensure that the uploaded file's size in bytes is greater than or equal to the provided parameter.

Let's test out our new validation rule. First, we will visit the `/` page and enter the value 'Jo' into the username field. If you submit the form, then you will be redirected back to the registration form. This is because our value is not long enough.

> That's what she s...

Oh no, you don't. This is a serious book; we aren't going to spoil it with 'she said' jokes.

We have now used two validation constraints, but what if we want to use them together? That's not a problem. Laravel will allow us to use any number of validation rules on our fields. Let's take a look at how this can be done.

Example 10: Multiple validation rules.

```
<?php

// app/Http/routes.php

/**
 * Show the registration form.
 */
Route::get('/', function () {
    return view('form');
});

/**
 * Handle a submitted registration form.
 */
Route::post('/registration', function (Request $request) {
    $formData = $request->all();

    $rules = [
        'username' => 'alpha_num|min:3'
    ];

    // Create a new validator instance.
    $validator = Validator::make($formData, $rules);

    if ($validator->passes()) {

        // Normally we would do something with the data.
```

```
28
29          return 'Data was saved.';
30      }
31
32      return redirect('/');
33  });
```

As you can see, we can pass some validation constraints within the value portion of our rules array by separating them with pipe | characters. This chapter would have arrived much sooner. However, I use Linux at work and a Mac at home, and it just took me 30 seconds to find the pipe key.

There's an alternative way to provide multiple validation rules. If you aren't a 60's grandfather, then you might not appreciate pipes. If you like, you can use a multi-dimensional array to specify additional validation rules.

Here's an example.

Example 11: Multiple validation rules as an array.

```
1   <?php
2
3   // app/Http/routes.php
4
5   /**
6    * Show the registration form.
7    */
8   Route::get('/', function () {
9       return view('form');
10  });
11
12  /**
13   * Handle a submitted registration form.
14   */
15  Route::post('/registration', function (Request $request) {
16      $formData = $request->all();
17
18      $rules = [
19          'username' => ['alpha_num', 'min:3']
20      ];
21
22      // Create a new validator instance.
23      $validator = Validator::make($formData, $rules);
24
```

```
    if ($validator->passes()) {

        // Normally we would do something with the data.

        return 'Data was saved.';
    }

    return redirect('/');
});
```

Laravel is a flexible framework, and it gives its users options. Use whichever method of assigning multiple validation rules that you prefer. I'll be using the pipe to emulate a distinguished British gentleman.

Validation Rules

Right people, listen up. There are a bunch of validation rules, so if we are going to get through this in one go, then I'm going to need your full attention. If you are reading this on your Kindle while in bed, then put the book down and go to sleep. You're going to need to be more awake.

Here are the available validation rules in alphabetical order.

---

accepted

This rule can be used to ensure that a positive confirmation has been provided. It will pass if the value under validation is one of the following values: 'yes', 'on' or a numeric 1. Its intended purpose is for when you wish to ensure that the user has agreed to something, for example, a terms and conditions checkbox.

Example 12: Validate acceptance.

```
['field' => 'accepted']
```

---

active_url

The 'active_url' validation will check to make sure that the value is a valid URL. To do this, it uses PHP's own `checkdnsrr()` method, which not only checks the structure of the URL but also confirms that the URL is available within your DNS records.

Example 13: Validate URLs.

```
['field' => 'active_url']
```

---

after

The 'after' validation rule accepts a single parameter, a string representation of time. The rule will ensure that the field contains a date that occurs after the given parameter. Laravel will use the `strtotime()` PHP method to convert both the value and the rule's parameter to a timestamp for comparison. How do I know this? Well, I was the one who originally submitted this feature!

Example 14: Validate date exists after parameter.

```
['field' => 'after:04/24/16']
```

---

alpha

The 'alpha' validation rule ensures that the provided value consists entirely of alphabetical characters.

Example 15: Validate alphabetic characters.

```
['field' => 'alpha']
```

---

alpha_dash

The 'alpha_dash' rule will ensure that the provided value consists of alphabetical characters and also dashes - and/or underscores _. This validation rule is very useful for validating URL portions such as 'slugs'.

Example 16: Validate alphabetic characters and dashes.

```
['field' => 'alpha_dash']
```

---

alpha_num

The 'alpha_num' rule will ensure that the provided value consists of alphabetical and numeric characters. I like to use this rule to validate username fields.

Example 17: Validate alphanumeric characters.

```
['field' => 'alpha_num']
```

---

before

The 'before' rule accepts a single parameter. The value under question must occur before the parameter when both values are converted to timestamps using PHP's `strtotime()` method. It is the exact opposite of the 'after' rule.

Example 18: Validate date exists before parameter.

```
['field' => 'before:04/24/16']
```

---

between

The 'between' rule accepts two parameters. The value that is being validated must have a size that exists between these two parameters. The type of comparison depends on the type of data being compared. For example, on numerical fields, the comparison will be a mathematical one. On a string, the comparison will be based on the length of the string in characters. On a file, the comparison will be based on the size of the file in bytes. A valid value must fall between the two specified values.

Example 19: Validate number exists between values.

```
['field' => 'between:5,7']
```

---

confirm

The 'confirmed' validation rule can be used to ensure that another field exists that matches the name of the current field appended with `_confirmation`. The value being validated must match the value of this other field. One use for this rule is for password field confirmations to ensure that the user has not inserted a typographical error into either of the fields. The following example will ensure that 'field' matches the value of 'field_confirmation'.

Example 20: Validate confirmation field.

```
['field' => 'confirm']
```

---

date

The 'date' validation rule will ensure that our value is a valid date. It will be confirmed by running the value through PHP's own `strtotime()` method.

Example 21: Validate date.

```
['field' => 'date']
```

---

date_format

The 'date_format' validation rule will ensure that our value is a date string that matches the format provided as a parameter. To learn how to construct a date format string, take a look at the PHP documentation for the `date()` method.

Example 22: Validate format of date.

```
['field' => 'date_format:d/m/y']
```

---

different

The 'different' validation rule will ensure that the value being validated is different to the value contained in the field described by the rule parameter.

Example 23: Validate value is different from another value.

```
['field' => 'different:another_field']
```

---

email

The 'email' validation rule will ensure that the value being validated is a valid email address. This rule is very useful when constructing registration forms.

Example 24: Validate email address.

```
['field' => 'email']
```

---

exists

The 'exists' validation rule will ensure that the value is present within a database table identified by the rule parameter. The column that will be searched will be the same name as the field being validated. Alternatively, you can provide an optional second parameter to specify a column name.

This rule can be very useful for registration forms to check whether a username has already been taken by another user.

Example 25: Validate value exists in database.

```
['field' => 'exists:users,username']
```

Any additional pairs of parameters passed to the rule will be added to the query as additional `where` clauses. Like this:

Example 26: Validate value exists in database with extra constraints.

```
['field' => 'exists:users,username,role,admin']
```

The above example will check to see if the value exists within the username column of the users table. The role column must also contain the value 'admin'.

---

image

The 'image' validation rule will ensure that the file that has been uploaded is a valid image. For example, the extension of the file must be one of the following: `.bmp`, `.gif`, `.jpeg` or `.png`.

Example 27: Validate image.

```
['field' => 'image']
```

---

in

The 'in' validation rule will ensure that the value of the field matches one of the provided parameters.

Example 28: Validate value exists in set.

```
['field' => 'in:red,brown,white']
```

---

integer

This is an easy one! The 'integer' validation rule will ensure that the value of the field is an integer. That's it!

Example 29: Validate value as integer.

```
['field' => 'integer']
```

---

ip

The 'ip' validation rule will check to make sure that the value of the field contains a well-formatted IP address.

Example 30: Validate value as IP address.

```
['field' => 'ip']
```

---

max

The 'max' validation rule is the exact opposite of the 'min' rule. It will ensure that the size of the field being validated is less than or equal to the supplied parameter. If the field is a string, the parameter will refer to the length of the string in characters. For numerical values, the comparison will be made mathematically. For file upload fields the comparison will be made using the size of the file in bytes.

Example 31: Validate max length/size of value.

```
['field' => 'max:3']
```

---

mimes

The 'mimes' validation rule ensures that the provided string is the name of a French mime. Just kidding. This rule will check the mime type of an uploaded file to ensure that it matches one of the parameters provided.

Example 32: Validate file mime type.

```
['field' => 'mimes:pdf,doc,docx']
```

---

min

The 'min' validation rule is the direct opposite of the 'max' rule. It can be used to ensure that a field value is greater than or equal to the provided parameter. If the field is a string, the parameter will refer to the length of the string in characters. For numerical values, the comparison will be made mathematically. For file upload fields the comparison will be made using the size of the file in bytes.

Example 33: Validate value is less than parameter.

```
['field' => 'min:5']
```

---

not_in

As the name suggests, this validation rule is the exact opposite of the 'in' rule. It will ensure that the field's value does not exist within the list of supplied parameters.

Example 34: Validate value is not in set.

```
['field' => 'not_in:blue,green,pink']
```

---

numeric

The 'numeric' rule will check to make sure that the field being validated contains a numeric value.

Example 35: Validate value is numeric.

```
['field' => 'numeric']
```

---

regex

The 'regex' validation rule is the most flexible rule available within Laravel's validation component. With this rule, you can provide a custom regular expression as a parameter that the field under validation must match. This book will not cover regular expressions in detail since it is a vast topic worthy of a book of its own.

You should note that because pipe | characters can be used within regular expressions, you should use nested arrays rather than pipes to attach multiple validation rules when using the 'regex' rule.

Example 36: Validate using regular expression.

```
1 ['field' => 'regex:[a-z]']
```

---

required

The 'required' validation rule can be used to ensure that the current field exists within the validation data array.

Example 37: Validate value is present.

```
1 ['field' => 'required']
```

---

required_if

The 'required_if' validation rule ensures that the current field must be present only if a field defined by the first parameter of the rule matches the value supplied by the second parameter.

Example 38: Validate if value is conditionally present.

```
1 ['field' => 'required_if:username,zoidberg']
```

---

required_with

The 'required_with' is used to ensure that the current value is present only if one or more fields defined by the rule parameters are also present.

Example 39: Validate value if another field exists.

```
['field' => 'required_with:age,height']
```

---

required_without

The 'required_without' rule is the direct opposite of the 'required_with' rule. It can be used to ensure that the current field is present only when the fields defined by the rule parameters are not present.

Example 40: Validate value if another field does not exist.

```
['field' => 'required_without:age,height']
```

---

same

The 'same' validation rule is the direct opposite of the 'different' rule. It is used to ensure that the value of the current field is the same as another field defined by the rule parameter.

Example 41: Validate value is the same as another field.

```
['field' => 'same:age']
```

---

size

The 'size' rule can be used to ensure that the value of the field is of a given size provided by the rule parameter. If the field is a string, the parameter will refer to the length of the string in characters. For numerical values, the comparison will be made mathematically. For file upload fields the comparison will be made using the size of the file in bytes.

Example 42: Validate length or size of value.

```
['field' => 'size:8']
```

---

unique

The 'unique' rule ensures that the value of the current field is not already present within the database table defined by the rule parameter. By default, the rule will use the name of the field as the table column in which to look for the value. However, you can provide an alternate column within the second rule parameter. This rule is useful for checking whether a user's provided username is unique when handling registration forms.

Example 43: Validate value is unique in the database.

```
1  ['field' => 'unique:users,username']
```

You can provide extra optional parameters to list some IDs for rows that will be ignored by the unique rule.

Example 44: Extra rows to be ignored.

```
1  ['field' => 'unique:users,username,4,3,2,1']
```

---

url

The 'url' validation rule can be used to ensure that the field contains a valid URL. Unlike the 'active_url' validation rule, the 'url' rule only checks the format of the string and does not check DNS records.

Example 45: Validate URL format.

```
1  ['field' => 'url']
```

Well, that's all of them. That wasn't so bad, right? We aren't done yet, though. Let's take a look at error messages.

Error Messages

In the first chapter, we learned how we can perform validation and how to redirect back to a form upon failure. However, that method doesn't offer the user much in the way of constructive feedback.

Fortunately, Laravel collects some error messages describing why the validation attempt failed. Let's take a look at how we can access this information.

Example 46: Retrieve error messages.

```php
<?php

// app/Http/routes.php

/**
 * Show the registration form.
 */
Route::get('/', function () {
    return view('form');
});

/**
 * Handle a submitted registration form.
 */
Route::post('/registration', function (Request $request) {
    $formData = $request->all();

    $rules = [
        'username' => 'alpha_num'
    ];

    // Create a new validator instance.
    $validator = Validator::make($formData, $rules);

    if ($validator->passes()) {

        // Normally we would do something with the data.

        return 'Data was saved.';
    }

    $errors = $validator->messages();

    return redirect('/');
});
```

In the above example, you will see that we can access the validation error messages object using the `messages()` method of our validator instance. Since we are redirecting to our form route, how do we access the error messages within our forms?

Well, I think we could probably use the `withErrors()` method for this.

> I don't believe you, you keep lying to me!

Oh yeah? Well, check this out.

Example 47: Redirect with errors.

```php
<?php

// app/Http/routes.php

/**
 * Show the registration form.
 */
Route::get('/', function () {
    return view('form');
});

/**
 * Handle a submitted registration form.
 */
Route::post('/registration', function (Request $request) {
    $formData = $request->all();

    $rules = [
        'username' => 'alpha_num'
    ];

    // Create a new validator instance.
    $validator = Validator::make($formData, $rules);

    if ($validator->passes()) {

        // Normally we would do something with the data.

        return 'Data was saved.';
    }

    return redirect('/')->withErrors($validator);
});
```

You will notice that we pass the validator instance to the `withErrors()` chained method. This method flashes the errors from the form to the session. Before we continue any further, let's add more validation rules to our example.

Example 48: Additional validation rules.

```
<?php

// app/Http/routes.php

/**
 * Show the registration form.
 */
Route::get('/', function () {
    return view('form');
});

/**
 * Handle a submitted registration form.
 */
Route::post('/registration', function (Request $request) {
    $formData = $request->all();

    $rules = [
        'username'    => 'required|alpha_num|min:3|max:32',
        'email'       => 'required|email',
        'password'    => 'required|confirm|min:3'
    ];

    // Create a new validator instance.
    $validator = Validator::make($formData, $rules);

    if ($validator->passes()) {

        // Normally we would do something with the data.

        return 'Data was saved.';
    }

    return redirect('/')->withErrors($validator);
});
```

Now let's see how we can access our error messages from the form view.

Example 49: Use validation errors within a form.

```
<!-- resources/views/form.blade.php -->

<h1>Registration form for our club!</h1>

<form action="{{ url('registration') }}" method="POST">

    {{-- CSRF Token. ---------------------------}}
    {{ csrf_field() }}

    <ul class="errors">
        @foreach($errors->all() as $message)
            <li><p>{{ $message }}</p></li>
        @endforeach
    </ul>

    {{-- Username field. ------------------------}}
    <label for="username">Username</label>
    <input type="text" name="username" />

    {{-- Email field. ---------------------------}}
    <label for="email">Email</label>
    <input type="text" name="email" />

    {{-- Password field. ------------------------}}
    <label for="password">Password</label>
    <input type="password" name="password" />

    {{-- Password confirmation field. -----------}}
    <label for="password_confirmation">Password confirmation</label>
    <input type="password" name="password_confirmation" />

    {{-- Form submit button. --------------------}}
    <input type="submit" value="Register" />

</form>
```

When our view is loaded, the `$errors` variable is added to the view data. It's always there, and it's always an error messages container instance. You don't have to worry about checking for its existence or contents. If we have used `withErrors()` to flash our error messages to the session in a previous request, then Laravel will automatically add them to the errors object. How convenient!

We can access an array of all error messages by using the `all()` method on the `$errors` error messages instance. In the above view, we loop through the entire array of messages outputting each of them within a list element.

Right, that's enough yapping. Let's give it a try. Go ahead and visit the `/` URL. Submit the form without entering any information to see what happens.

We are redirected back to the form. This time, however, a set of error messages are displayed.

- The username field is required.
- The email field is required.
- The password field is required.

Great! Our application's users are now aware of any validation errors upon registration. However, it's more convenient for our users if the error messages are nearer to the fields that they describe. Let's alter the view a little.

Example 50: Show validation errors on a per-field basis.

```
<!-- resources/views/form.blade.php -->

<h1>Registration form for our club!</h1>

<form action="{{ url('registration') }}" method="POST">

    {{-- CSRF Token. ---------------------------}}
    {{ csrf_field() }}

    {{-- Username field. ------------------------}}
    <label for="username">Username</label>
    <input type="text" name="username" />
    @if($errors->has('username'))
        <small>{{ $errors->first('username') }}</small>
    @endif

    {{-- Email field. ---------------------------}}
    <label for="email">Email</label>
    <input type="text" name="email" />
    @if($errors->has('email'))
        <small>{{ $errors->first('email') }}</small>
    @endif

    {{-- Password field. ------------------------}}
```

```
25      <label for="password">Password</label>
26      <input type="password" name="password" />
27      @if($errors->has('password'))
28          <small>{{ $errors->first('password') }}</small>
29      @endif
30
31      {{-- Password confirmation field. -----------}}
32      <label for="password_confirmation">Password confirmation</label>
33      <input type="password" name="password_confirmation" />
34      @if($errors->has('password_confirmation'))
35          <small>{{ $errors->first('password_confirmation') }}</small>
36      @endif
37
38      {{-- Form submit button. --------------------}}
39      <input type="submit" value="Register" />
40
41  </form>
```

We can use the `first()` method on the validation errors object to retrieve a single error for a given field. Simply pass the name of the field as the first parameter to the `first()` method.

Let's resubmit the form. This time, place a single exclamation `!` mark within the username field.

- The username may only contain letters and numbers.

That's better. Well... it's a little bit better. Most forms only show a single validation error per field, so as not to overwhelm the application's user. While there's only one error here, if there were additional fields, we might see more. We should only display a single error message, so as not to overwhelm our user.

Let's take a look at how this is done with the Laravel validation errors object.

Example 51: Show single validation error per field.

```
<!-- resources/views/form.blade.php -->

<h1>Registration form for our club!</h1>

<form action="{{ url('registration') }}" method="POST">

    {{-- CSRF Token. ---------------------------}}
    {{ csrf_field() }}

    {{-- Username field. ------------------------}}
    <span class="error">{{ $errors->first('username') }}</span>
    <label for="username">Username</label>
    <input type="text" name="username" />

    {{-- Email field. ---------------------------}}
    <span class="error">{{ $errors->first('email') }}</span>
    <label for="email">Email</label>
    <input type="text" name="email" />

    {{-- Password field. ------------------------}}
    <span class="error">{{ $errors->first('password') }}</span>
    <label for="password">Password</label>
    <input type="password" name="password" />

    {{-- Password confirmation field. -----------}}
    <span class="error">{{ $errors->first('password_confirmation') }}</span>
    <label for="password_confirmation">Password confirmation</label>
    <input type="password" name="password_confirmation" />

    {{-- Form submit button. --------------------}}
    <input type="submit" value="Register" />

</form>
```

By using the `first()` method on the validation errors object and passing the field name as a parameter, we can retrieve the first error message for that field.

Once again, submit the form with only an exclamation `!` mark within the username field. This time, we receive only a single error message for the first field.

- The username may only contain letters and numbers.

By default, the validation messages instance's methods return an empty array or `null` if no messages exist. What this means is that you can use it without having to check for the existence of messages. However, if you wish to check whether or not an error message exists for a field, you can use the `has()` method.

Example 52: Check for the existence of a validation error.

```
@if($errors->has('email'))
    <p>Yey, an error!</p>
@endif
```

In the previous examples for the `all()` and `first()` methods you will have noticed that we wrapped our error messages within HTML elements. However, if one of our methods return null, the HTML would still be displayed.

We can avoid having empty HTML elements appearing in our view source code by passing the containing HTML in string format as the second parameter to the `all()` and `first()` methods. For example, here's the `all()` method with the wrapping list elements as a second parameter.

Example 53: Extra formatting for validation errors.

```
<!-- resources/views/form.blade.php -->

<h1>Registration form for our club!</h1>

<form action="{{ url('registration') }}" method="POST">

    {{-- CSRF Token. ---------------------------}}
    {{ csrf_field() }}

    <ul class="errors">
    @foreach($errors->all('<li>:message</li>') as $message)
        {{ $message }}
    @endforeach
    </ul>

    {{-- Username field. ------------------------}}
    <label for="username">Username</label>
    <input type="text" name="username" />

    {{-- Email field. ---------------------------}}
    <label for="email">Email</label>
    <input type="text" name="email" />
```

```
23
24      {{-- Password field. ------------------------}}
25      <label for="password">Password</label>
26      <input type="password" name="password" />
27
28      {{-- Password confirmation field. -----------}}
29      <label for="password_confirmation">Password confirmation</label>
30      <input type="password" name="password_confirmation" />
31
32      {{-- Form submit button. --------------------}}
33      <input type="submit" value="Register" />
34
35  </form>
```

The `:message` portion of the second parameter to the `all()` method will be replaced by the actual error message when the array is constructed.

The `first()` method has a similar optional parameter.

Example 54: Extra formatting for the first validation error.

```
1   <!-- resources/views/form.blade.php -->
2
3   <h1>Registration form for our club!</h1>
4
5   <form action="{{ url('registration') }}" method="POST">
6
7       {{-- CSRF Token. ---------------------------}}
8       {{ csrf_field() }}
9
10      {{-- Username field. ------------------------}}
11      {{ $errors->first('username', '<span class="error">:message</span>') }}
12      <label for="username">Username</label>
13      <input type="text" name="username" />
14
15      {{-- Email field. ---------------------------}}
16      {{ $errors->first('email', '<span class="error">:message</span>') }}
17      <label for="email">Email</label>
18      <input type="text" name="email" />
19
20      {{-- Password field. ------------------------}}
21      {{ $errors->first('password', '<span class="error">:message</span>') }}
22      <label for="password">Password</label>
```

```
    <input type="password" name="password" />

    {{-- Password confirmation field. -----------}}
    {{ $errors->first('password_confirmation', '<span class="error">:message</sp\
an>') }}
    <label for="password_confirmation">Password confirmation</label>
    <input type="password" name="password_confirmation" />

    {{-- Form submit button. --------------------}}
    <input type="submit" value="Register" />

</form>
```

Custom Validation Rules

Oh I see, you're not happy with all that Laravel has given you? You wish to have your own validation methods do you? Very well, time to bring out the big guns. Laravel is flexible enough to let you specify your own rules.

Let's have a look at how this can be done.

Example 55: Extend the validator with a custom rule.

```
<?php

// app/Http/routes.php

Validator::extend('awesome', function ($field, $value, $params) {
    return $value == 'awesome';
});

Route::get('/', function () {
    return View::make('form');
});

Route::post('/registration', function (Request $request) {
    // Fetch all request data.
    $data = $request->all();

    // Build the validation constraint set.
    $rules = [
        'username'   => 'awesome',
```

```
    ];

    // Create a new validator instance.
    $validator = Validator::make($data, $rules);

    if ($validator->passes()) {
        // Normally we would do something with the data.
        return 'Data was saved.';
    }

    return Redirect::to('/')->withErrors($validator);
});
```

There's no default location for custom validation rules to be defined, so I have added it to the `routes.php` file to simplify the example. You could include a `validators.php` file and provide custom validations in there if you like.

We have attached our 'awesome' validation rule to our username field. Let's take a closer look at how the validation rule is created.

Example 56: Our custom validation rule.

```
<?php

// app/Http/routes.php

Validator::extend('awesome', function ($field, $value, $params) {
    return $value == 'awesome';
});
```

To create a custom validation rule we use the `Validator::extend()` method. The first parameter to the method is the nickname that will be given to the validation rule. This is what we will use to attach it to a field. The second parameter to the method is a closure. Should the closure return a boolean result of `true` then, the validation attempt will have passed. If a boolean `false` is returned from the closure, then the validation attempt will have failed.

The parameters that are handed to the closure are as follows. The first parameter is a string containing the name of the field that is being validated. In the above example, the first parameter would contain the string 'username'.

The second parameter to the extend closure contains the value of the field.

The third parameter contains an array of any parameters that have been passed to the validation rule. Use them to customize your validation rules as required.

If you prefer to define your custom validation rule within a class, rather than closure, then you won't be able to. Stop wanting things.

Now wait, I'm just kidding. Laravel can do that. Let's create a class that will accomplish this.

Example 57: Custom validation rule as a class.

```php
<?php

// app/Validators/CustomValidation.php

namespace App\Validators;

class CustomValidation
{
    public function awesome($field, $value, $params)
    {
        return $value == 'awesome';
    }
}
```

As you can see, our validation class contains any number of methods that have the same method signature as our validation closure. This means that a custom validator class can have as many validation rules as we like.

Once again there's no perfect location for these classes, so you will have to define your project structure. I chose to put my validation class within the `app/validators` folder and class map that folder with Composer.

Well, that's everything covered.

> Wait, the validation class, doesn't contain the validation nickname.

Ah yes, I almost forgot. Well done observant reader! You see, for the validation rule to be discovered, we need to use the `Validator::extend()` method again. Let's take a look.

Example 58: Register a custom validation class.

```
<?php

// app/Http/routes.php

Validator::extend('awesome', 'App\Validators\CustomValidation@awesome');
```

This time the `Validator::extend()` method is given a string as a second parameter. Just like when routing to a controller, the string consists of the class name of the validation rule class and the action representing the rule separated by an `@` symbol.

In a later chapter, we will learn how to extend the Validation class as an alternative, more advanced method of providing custom validation rules.

Custom Validation Messages

Laravel has provided default validation messages for all the inbuilt validation rules, but what if you don't like the default ones or want to write your application for a region that doesn't use English as its primary language.

Well, don't panic! Laravel will let you override the inbuilt validation messages. We just need to build an additional array and pass it to the `make()` method of the validator instance.

> Hey, we already looked at the `Validator::make()` method?!

That's true, but once again I lied to you.

> Why do you keep tormenting me?

I'm not sure. I guess I think of it as a hobby at this point. Let's take a look at an example array of custom validation messages.

Example 59: Custom validation message.

```php
<?php

// app/Http/routes.php

Route::get('/', function () {
    return View::make('form');
});

Route::post('/registration', function (Request $request) {
    // Fetch all request data.
    $data = $request->all();

    // Build the validation constraint set.
    $rules = [
        'username'   => 'min:3',
    ];

    // Build the custom messages array.
    $messages = [
        'min' => 'Yo dawg, this field aint long enough.'
    ];

    // Create a new validator instance.
    $validator = Validator::make($data, $rules, $messages);

    if ($validator->passes()) {
        // Normally we would do something with the data.
        return 'Data was saved.';
    }

    return Redirect::to('/')->withErrors($validator);
});
```

That's a significant example, let's hit the focus button.

> I can't find the focus button on my keyboard.

Well, if you have a Mac, it's that one above the tab key. It looks like this ±.

> Are you sure?

Well, do you have any other ideas of what that button means?

> Ah, I see your point.

Right, let's focus on the example.

Example 60: Validation message array.

```
<?php

// Build the custom messages array.
$messages = [
    'min' => 'Yo dawg, this field aint long enough.'
];

// Create a new validator instance.
$validator = Validator::make($data, $rules, $messages);
```

The messages array is an optional third parameter to the `Validator::make()` method. It contains any custom validation messages that you wish to provide and will also override the default validation messages. The key for the validation message array represents the name of the validation rule, and the value is the message to display if the validation rule fails.

In the above example, we have overridden the failed validation message for the `min` validation rule.

We can also use this method to provide validation errors for our custom validation rule. Our custom rules won't have validation error messages by default, so it's usually a good idea to do so.

Example 61: Create validator with custom messages.

```
<?php

// Build the custom messages array.
$messages = [
    'awesome' => 'Please enter a value that is awesome enough.'
];

// Create a new validator instance.
$validator = Validator::make($data, $rules, $messages);
```

If we want to provide a custom error message for a particular field, we can do so by providing the name of the field, a period `.` character, and the type of validation as the key within the array.

Example 62: Validation message for a specific field.

```
<?php

// Build the custom messages array.
$messages = [
    'username.min' => 'Hmm, that looks a little small.'
];

// Create a new validator instance.
$validator = Validator::make($data, $rules, $messages);
```

The message in the above example will only be shown when the `min` validation rule fails for the 'username' field. All other `min` failures will use the default error message.

That's all I have to offer on validation right now. In a later chapter we will learn how to extend Laravel's validator class and how to provide translatable validation error messages, but, for now, let's move on to the next chapter to learn about events.

25. Events

```
Dear handsome reader,

You have been formally invited to Dayle Rees' happy panda party. Please dress su\
itably and bring the enclosed coloured flags. More will be explained at the even\
t.

Kind Regards,

Lushui.

Chief panda Butler Esq.
```

Exciting times, right? You arrive at Code Smart Manor wearing your Sunday best with a coloured flag in hand. In the grand hall, you are surrounded by thousands of other handsome developers. On the walls are oil paintings of famous red pandas that have served the Rees family throughout history.

Champagne and caviar are served by red panda butlers wearing their finest orange and white tuxedos. It's a swish event. I mean, I'm a guy of good tastes, right?

I suppose it's time that I address my guests? Let me step up to the panda podium.

Welcome, beautiful people, to the annual panda party. You have all been invited due to your fantastic taste in technical writing.

Now, I know that small-talk at these types of events can be a little awkward for developers. Fear not! I have devised an ingenious system. I hope that you have all brought your coloured flags along with you?

- cheers *

Excellent! I'm glad to see that you're all so enthusiastic.

Here's how it's going to work. If you would like to small-talk about the weather, then please hold up your red flag. If you would like to small-talk about recent sporting events, then please hold up the blue flag. Finally, if you would like to small-talk about video games, then please hold up the green flag.

With this simple system in place, we are all free to enjoy ourselves this evening. Please help yourself to champagne and let's drink to your future projects with the Laravel framework.

Concept

So what was that all about? Well, apart from a little fun, it serves to illustrate the basics of event-driven programming. By holding up a coloured flag we are 'firing' an event. Here's an example.

Example 01: Our first event.

```
Event::fire('raise.blue.flag');
```

Other people can 'listen' or 'observe' this event. Here's how that would look.

Example 02: Listen for our first event.

```
Event::listen('raise.blue.flag', function () {
    return new \App\SmallTalk::make('sports');
});
```

With our event listener in place, as soon as someone raises a blue flag we automatically initiate small talk about sports with the person. It's a very efficient process. We can register as many listeners as we like. We can even register multiple listeners to the same event.

So how do we employ a system such as this within Laravel? Well, it's exactly as in the previous examples. Ok, I lied again, a little. That `SmallTalk` class doesn't exist. The rest is just fine, though! Why don't we take a closer look?

Firing Events

Events are fired with a 'key'. We can use this key to register listeners later. We fire the events using the `fire()` method on the `Event` class. The first and only required parameter is the key itself. Let's take a look at an example.

Example 03: Fire an event.

```
Event::fire('my.event');
```

Any listeners that have been registered will be executed when this line of code has been run.

It's useful to be able to be notified in this way, but we can do much more with this eventing system. We can pass additional information along with our events. Our listeners can respond to this information, or may even modify it.

Let's take a look at another example.

Example 04: Fire an event with parameters.

```
Event::fire('my.event', [$object, $value]);
```

With an optional second parameter, we can pass an array of PHP variables that will be handed to registered listeners. The standard rules of PHP apply. Objects themselves are given directly, and basic types are passed as a copy.

What this means is that if you place an object or class instance into the second parameter array, then the listener will be able to modify it.

Let's take a look at the other side of the equation shall we? It's time to register some listeners.

Listening for Events

Listening for an event is simple. We can use the `Event::listen()` method. The first parameter is the 'key' of the event that we wish to respond to. The second parameter is a Closure that can be used to respond to the event.

As always, let's take a look at an example.

Example 05: Listen for an event.

```
Event::listen('my.event', function () {
    // Perform some action.
    // Update the database?
});
```

Within the Closure, we can perform any action that we need to. We could log that an event occurred. We could update a database model. Anything is possible. Use your imagination!

If our event is fired with additional parameters, then we can capture them by placing placeholder parameters within our Closure. Let's take a look at this in action.

Example 06: Listen for an event with parameters.

```
Event::listen('my.event', function ($first, $second) {
    // Use $first and $second.
});
```

The parameters are provided in the same order as they are represented in the array passed to the event firing method.

As I mentioned earlier, we can register multiple listeners to a single event. All registered listeners will be executed. Here's an example.

Example 07: Multiple listeners for a single event.

```
Event::listen('my.event', function () {
    // First listener.
});

Event::listen('my.event', function () {
    // Second listener.
});

Event::listen('my.event', function () {
    // Third listener.
});

Event::fire('my.event');
```

Within the above example, all listeners will be executed once the `my.event` has been fired.

Maybe we have dealt with the event within the first listener? What if we don't want the other listeners to be processed? Not a problem! If we return a boolean `false` value from our listener, then we will break the event chain and subsequent listeners will not be executed.

For example, within this code snippet:

Example 08: Breaking the event chain.

```
Event::listen('my.event', function () {
    // First listener.
    return false;
});

Event::listen('my.event', function () {
    // Second listener.
});

Event::listen('my.event', function () {
    // Third listener.
});

Event::fire('my.event');
```

... only the first event listener will be executed.

Now I'm sure you're getting tired of me telling you this, but wherever there's a Closure, you could also use a PHP class. The syntax is the same as it always is. First, you define your event listener class.

Example 09: A listener as a class.

```
class MyListener
{
    public function process()
    {
        // Handle the event here.
    }
}
```

Next, you register the listener using a notation that's similar to how you define a controller and action pair. Here's an example.

Example 10: Register a listener class.

```
Event::listen('my.event', 'MyListener@process');
```

The second parameter to the `listen()` method is now the class and method pair. If you don't provide a method, then the event system will automatically look for a method named `handle()`.

Events can also be subscribed to with priority. This way we can change the order used to execute our listeners. Here's an example.

Example 11: Event listener priority.

```
Event::listen('my.event', function () {
    // First listener.
}, 1);

Event::listen('my.event', function () {
    // Second listener.
}, 3);

Event::listen('my.event', function () {
    // Third listener.
}, 5);

Event::fire('my.event');
```

By specifying an integer as the third parameter of the `listen()` method we can change the order that our listeners are processed. Listeners with a higher integer value will be executed first. In the above example, the listeners will be executed in reverse order. Nice and simple!

When registering events, you don't have to be exact about your event key. If you like, you could use a wildcard (*) within your listener to register for a subset of events. Here's an example.

Example 12: Listen to wildcard event names.

```
Event::listen('my.*', function () {
    // ...
});
```

The event listener will now be executed once any event that starts with the prefix `my.` is fired.

Event Subscribers

In the previous chapter, I shared some locations where you could register your code. With events, you have an additional option. An event subscriber can be used to create a class that will handle multiple events.

Here's an example of a simple event subscriber.

Example 13: Our first event subscriber.

```
class MyListeners
{
    public function firstListener()
    {
        // First event listener.
    }

    public function secondListener()
    {
        // Second event listener.
    }

    public function thirdListener()
    {
        // Third event listener.
```

```
    }

    public function subscribe($events)
    {
        $events->listen('first.event', 'MyListeners@firstListener');
        $events->listen('second.event', 'MyListeners@secondListener');
        $events->listen('third.event', 'MyListeners@thirdListener');
    }
}
```

Our event subscriber is similar to our standard event class. It has some listener methods, but also a new method named `subscribe()`. This method accepts an eventing instance, which we have called `$events` in the example above.

In previous examples we used the `Event` class to fire and listen for events, do you remember? Well, this instance is the same. We can use it to register our listeners from within our subscriber class.

To allow for our subscriber class to function correctly, we need only provide a call to the `subscribe()` method with an instance of our subscriber class, like this.

Example 14: Register an event subscriber.

```
Event::subscribe(new MyListeners);
```

Whichever method you choose to register your events, I'm sure it will work fantastically for you. Now have another sip of champagne. It's on me!

Global Events

Want to hear another secret? Now, this is just between you and me. I don't want you sharing this with the other developers, okay? We can spy on Laravel.

That's right. We can intercept its messages if we choose. You see, Laravel fires its own events. For example, when Laravel executes an SQL query the `illuminate.query` event is fired. Remember that `illuminate` is the codename for Laravel's component suite.

The parameters passed with the `illuminate.query` event are related to the query itself. Which includes the SQL query that is to be executed.

Laravel also fires an `illuminate.log` event when a new call to the `Log` class is made. By interrogating the parameters that are passed to the event we can easily intercept any messages that are logged by the system.

There are also several events that are so useful that Taylor has provided some simple short-cuts that can be used to listen for them.

If you pass a Closure to the `App::after()` method, then the Closure will be executed after the framework has completed it's request-response cycle. Just before the response is sent to the client.

The current request and response are sent as parameters to the event, which means you have a 'last-chance' to alter the response before it is sent. Here's an example.

Example 15: The after framework event.

```
App::after(function ($request, $response) {
    $response->headers->set('Access-Control-Allow-Origin', '*');
});
```

In the above example we modify the `$response` parameter to add the 'Access-Control-Allow-Origin' header. This head will now be served along with every response from the system.

You also have access to the `App::before()` method that is an event which is triggered before the routing has been performed. This event only passes the current request as a parameter.

Stubs for these methods are found in the example `filters.php` file that ships with the framework. While they exist in the filters file, they are actually implemented in a fashion that is more similar to an event.

Use Cases

I certainly encourage you to use the event system in a creative manner to solve your problems, but here are some example use cases that I have found useful.

Logging & Audit

When a particular function relating to my application occurs, I like to fire an event. Here are some examples of the events that might be fired.

- `user.created`
- `user.deleted`
- `profile.updated`

Should I wish to log this information, forward it to an external service for analytical purposes, or event audit actions within the system, I need only register an event subscriber to watch for these defined events.

The overhead for firing an event is rather small, so I tend to place them wherever is useful. You never know what kind of actions you wish to track at a later date.

Hooks

With software pages similar to CMS's or task management systems, or "those kind" of re-distributable packages you will often find a way of 'hooking' your own extensions into the code. Or modifying the objects that the system works with.

Events are a fantastic way of allowing developers to mod on top of your system without having to extend classes.

This chapter has been in preparation for an upcoming chapter on the inversion of control principle and Laravel's own service container.

However, the next chapter will also include a little container magic. We'll be looking at dependency injection.

26. Dependency Injection

In an earlier chapter, we learned about how Facades work. In that chapter, I mentioned that while facades were ideal for beginners, they weren't perfect for object oriented programming.

You see, facades are the equivalent of using a global function. When writing clean code and adhering to OOP principles, we want to be working with instances of classes, and not making static calls to them. What this means is that our classes will hold instances of any service or dependency that they use. This way, we know what our class is using, and have the ability to swap them out for different variations. For example, we might swap a dependency out for a "mock" object, which would appear to give the same functionality, but would result in functionality that is more convenient for testing purposes.

This chapter will contain some advanced practices for building Laravel-powered applications. If you have difficulty following the examples or have any other issues with the chapter, then please don't panic! You're welcome to continue using Facades. Simply skip to the next chapter.

Concept

Dependency injection is a scary sounding word, but the technique itself is quite simple. When we create a class, we give it everything that it needs to do its work. In PHP, there are two ways of doing this. Let's take a look at the first example. We'll be using a made-up class.

Example 01: Injection through the constructor.

```
1  <?php
2
3  class Wolverine
4  {
5      protected $claws;
6
7      protected $spandex;
8
9      public function __construct(Claws $claws, Spandex $spandex)
10     {
11         $this->claws = $claws;
```

```
        $this->spandex = $spandex;
    }
}
```

Here we have a class that has a few dependencies. Wolverine needs her claws and a black spandex outfit to be a competent superhero. That's right; I said *her*. If you were expecting Logan, then you need to catch up on your Marvel!

When creating a Wolverine class, we will first create instances of claws and spandex, and then pass them into the class upon instantiation. The constructor for Wolverine will set the two dependencies as class properties. What this means is that our class now holds instances of its dependencies, and can use them in any of its methods.

Note that if we had used Facades for 'claws' and 'spandex,' then they would not be held in the class. They would exist outside the class, and will be called in a similar fashion to a global function.

The method we have used above is called constructor-based dependency injection and is the preferred method. There is, however, another method of injecting our dependencies. Let's take a look at an example.

Example 02: Injection using setters.

```
<?php

class Wolverine
{
    protected $claws;

    protected $spandex;

    public function setClaws(Claws $claws)
    {
        $this->claws = $claws;
    }

    public function setSpandex(Spandex $spandex)
    {
        $this->spandex = $spandex;
    }
}
```

In this example, we can inject our dependencies using setter methods. This technique is less favorable to constructor-based dependency injection because we can make a class instance without having the dependencies set since we aren't relying on the constructor. We depend on the instantiation code to supply our dependencies, instead of enforcing a contract.

We have learned that dependency injection is giving classes instances of the dependencies that they need to function. Quite simple, right?

Dependency Injection with the Container

In an earlier chapter, we learned that the container is where Laravel keeps all of its services, but it's much smarter than that. It also has a bunch of other neat tricks.

The container can create instances of classes. Let's look at an example of this. In this example, `$app` is a reference to our Laravel container.

Example 03: Instantiation with the container.

```
<?php

// Instantiate a class directly.
$deadpool = new Deadpool;

// Instantiate a class through the container.
$deadpool = $app->make(Deadpool::class);
```

In the first example, we instantiate a Deadpool class in the traditional way. In the second example, we use the `make()` method of the container to instantiate our Deadpool class.

The `Deadpool::class` part of the above example is a PHP feature that will return the fully qualified class name as a string.

What's the point in that method?

Well, I'm glad you asked! You see, the Laravel container is intelligent. When a class needs constructor dependencies (like our old Wolverine class), then the container will try to provide them automatically. The container will inject dependencies under the following circumstances:

- The dependencies are simple classes, which require no custom values in their constructors.
- The dependencies were previously bound in the container.

Simply put, if Laravel has the means to instantiate the dependencies, then it will do so. Not only that but dependencies of dependencies will also be satisfied. For example, let's say that Wolverine needs an instance of Claws, and his claws need an instance of Adamantium, then all of these classes will be instantiated and injected where necessary. That's super convenient, right?

Injection within Controllers

Want to hear a little secret? When you use classes that are specific to Laravel, for example, its controllers, then they are automatically instantiated through the container. Let's imagine that we have routed to a controller called `ExampleController`. Here's our class.

Example 04: Controller constructor injection.

```
<?php

namespace App\Http\Controllers;

class ExampleController extends Controller
{
    protected $wolverine;

    public function __construct(Wolverine $wolverine)
    {
        $this->wolverine = $wolverine;
    }

    public function index()
    {
        // Make use of $this->wolverine.
    }
}
```

Because our controller was instantiated through the container, Laravel will automatically inject our Wolverine instance. Once we have set it as a class property, it will be available for use within all of our controller actions.

There are some instances where constructor injection within controllers will be a waste. Consider the following class.

Example 05: Wasteful injection.

```php
<?php

namespace App\Http\Controllers;

use App\Wolverine;

class ExampleController extends Controller
{
    protected $wolverine;

    public function __construct(Wolverine $wolverine)
    {
        $this->wolverine = $wolverine;
    }

    public function index()
    {
        // Make use of $this->wolverine.
    }

    public function other()
    {
        // Do something else.
    }
}
```

In the above controller, the `index()` action method will make use of Wolverine, but our `other()` method will not. However, both routes will end up having Wolverine injected into the controller. This is a little wasteful for the `other()` method don't you think?

That's what Taylor thought too, so he decided to implement action-based dependency injection for Laravel controllers. Here's an example.

Example 06: Action injection.

```php
<?php

namespace App\Http\Controllers;

use App\Wolverine;

class ExampleController extends Controller
{
    public function index(Wolverine $wolverine)
    {
        // Make use of $this->wolverine.
    }

    public function other()
    {
        // Do something else.
    }
}
```

This time, instead of injecting our dependencies into the constructor, we inject them into the action. What this means is that we don't have to set class properties, and each action has *only* the dependencies that it needs to do its work. You can type-hint as many injectable classes as you like into your actions, and feel free to inject through the constructor if there are dependencies that are used by all actions.

If your controller actions accept parameters, just place the parameters after your injected parameters. Here's an example.

Example 07: Action parameters with injection.

```php
<?php

namespace App\Http\Controllers;

use App\Wolverine;

class ExampleController extends Controller
{
    public function index(Wolverine $wolverine, $first, $second)
    {
        // Make use of $this->wolverine.
```

```
12      }
13  }
```

What's great about this method of injection, is that it's clear to see the dependencies of our controllers and their actions. We know what they need to function, and so we can make assumptions about their intended functionality. It's also possible to pass in different instances (for example, mocks) of dependencies when testing our controllers.

Injecting Services

Earlier in the book, I promised that I'd teach you how to be free of Facades, and to use Laravel's services in a manner that is more compatible with OO principles. Good news! That time is now.

Laravel's services are located in the container, this much we know already. Instances that are stored in the container have keys. You can almost think of it as a big intelligent array. The keys are string values that you can use to retrieve those instances when using the `make()` method of the container. Only when no bindings are found for a given key will the container instantiate a matching class instead, just as in the previous sections.

All of Laravel's services are bound in the container using up to three keys. Here are the keys that are used.

- A short name. For example, `router`.
- The instance class. For example `Illuminate\Routing\Router`.
- The contract for a service. For example, `Illuminate\Contracts\Routing\Registrar`.

What's this contract thing? It's simple! All of Laravel's services implement an interface called a 'contract.' These contracts define all of the public methods of that service. So if you'd like to swap a Laravel service out for another implementation, simply implement the service's contract interface, and Laravel will be more than happy!

Illuminate is a codename for the suite of Laravel components. You'll find that most of the framework classes are namespaced under `Illuminate`.

We can use the contracts for Laravel services to inject dependencies into our classes that are instantiated with the controller. Let's take a look at an example.

Example 08: Inject a service.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Contracts\Routing\UrlGenerator;

class ExampleController extends Controller
{
    public function index(UrlGenerator $url)
    {
        $exampleLink = $url->to('/example');
    }
}
```

By type-hinting the `Illuminate\Contracts\Routing\UrlGenerator` contract within our action, the framework will inject an instance of the URL generator. What this means is that we can use this instance instead of the `URL` facade, and we can avoid using global or static calls.

Contracts

In this book, we'll continue to use Facades. I believe that this is because they are much easier to grasp for beginners. However, if you'd like to step up your game, and instead start injecting your dependencies, then you'll need to know which contracts inject which services. Since I'm a nice person, here's a list of contracts you can inject. Please note that there will be some services here that we've not covered yet. All in due time.

`Illuminate\Contracts\Auth\Factory`

This contract will inject the service behind the `Auth` facade.

`Illuminate\Contracts\Bus\Dispatcher`

This contract will inject the dispatcher for the Laravel command bus implementation. It can be used to dispatch jobs.

`Illuminate\Contracts\Broadcasting\Broadcaster`

This contract will inject the broadcaster service, which can be used to issue push notifications.

`Illuminate\Contracts\Cache\Repository`

This contract will inject the service behind the `Cache` facade.

`Illuminate\Contracts\Config\Repository`

This contract will inject the service behind the `Config` facade.

`Illuminate\Contracts\Container\Container`

This contract will inject our application container. That's right, injecting the container, using the container. Inception? You betcha.

`Illuminate\Contracts\Cookie\Factory`

This contract will inject the service behind the `Cookie` facade.

`Illuminate\Contracts\Encryption\Encrypter`

This contract will inject the service behind the `Crypt` facade.

`Illuminate\Contracts\Events\Dispatcher`

This contract will inject the service behind the `Event` facade.

`Illuminate\Contracts\Filesystem\Cloud`

This contract will inject the service that can be used to store files in our cloud files service.

`Illuminate\Contracts\Filesystem\Factory`

This contract will inject the filesystem service.

`Illuminate\Contracts\Filesystem\Filesystem`

This contract will inject the service to interact with the local filesystem.

`Illuminate\Contracts\Foundation\Application`

This contract will inject the service behind the `App` facade. It's also an instance of our application container. More inception!

`Illuminate\Contracts\Hashing\Hasher`

This contract will inject the service behind the `Hash` facade.

`Illuminate\Contracts\Logging\Log`

This contract will inject the service behind the `Log` facade.

`Illuminate\Contracts\Mail\Mailer`

This contract will inject the service behind the `Mail` facade.

`Illuminate\Contracts\Queue\Queue`

This contract will inject the service behind the `Queue` facade.

`Illuminate\Contracts\Redis\Database`

This contract will inject the service behind the `Redis` facade.

`Illuminate\Contracts\Routing\Registrar`

This contract will inject the service behind the `Route` facade.

`Illuminate\Contracts\Routing\ResponseFactory`

This contract will inject the service behind the `Response` facade.

`Illuminate\Contracts\Routing\UrlGenerator`

This contract will inject the service behind the `URL` facade.

`Illuminate\Contracts\Validation\Factory`

This contract will inject the service behind the `Validator` facade.

`Illuminate\Contracts\View\Factory`

This contract will inject the service behind the `View` command.

`Illuminate\Http\Request`

Okay, so this one's not a contract. We're using the Symphony request, so it would be pointless to have a contract for this class. By type-hinting the above class, you'll get a copy of the object behind the `Request` facade.

Have I missed a service? Let me know, and I'll add it!

In the next chapter, we'll be taking a look at middleware.

27. Middleware

Before Laravel 5, when we wanted to restrict access to specific routes based on one or more conditions, we'd use a tool called a 'filter.' In Laravel 5, these filters have been replaced by something much more useful. We now have access to middleware.

Using middleware, we can once again restrict access to routes, but also modify the request and response objects to suit our needs. Let's take a look at how a middleware is constructed.

Middleware Classes

Within the latest iteration of the framework, middleware is defined within PHP classes. Let's take a look at a simple example now.

Example 01: Our first middleware.

```
<?php

// app/Http/Middleware/MyMiddleware.php

namespace App\Http\Middleware;

use Closure;

class MyMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure                  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        return $next($request);
    }
}
```

Here we have an example middleware class. As you can see, there is no need to extend a base class. Our class must implement a single method called `handle()`. Let's take a closer look at the method signature.

Example 02: Handle a middleware request.

```
/**
 * Handle an incoming request.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Closure                  $next
 * @return mixed
 */
public function handle($request, Closure $next);
```

The `handle()` method accepts two parameters. The first parameter is an instance of `Illuminate\Http\Request`. It's the same instance that you've been using in controllers to access your application's request data. The second parameter is `$next`, an instance of a Closure.

The `$next` parameter is a very special one. You see, middleware in Laravel are executed as a chain or pipeline. We pass the `$request` parameter into the `$next` Closure to execute the next middleware in the chain. Middleware chains will only terminate under the following circumstances.

- You break the chain manually, refusing to return `$next($request)`.
- The chain reaches its conclusion, and the framework returns the response object.

Right now, we return the result of the next middleware in the chain, like this:

Example 03: Continue the middleware chain.

```
public function handle($request, Closure $next)
{
    return $next($request);
}
```

This means that our middleware is doing nothing. If we were to return a response object instead of the result of the next middleware, the framework would serve that response instead. This is useful. It means we're able to change the flow of the request from within our middlewares. Let's demonstrate this with an example.

Example 04: Terminate a middleware.

```php
<?php

// app/Http/Middleware/MyMiddleware.php

namespace App\Http\Middleware;

use Closure;

class MyMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure                  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->has('terminate')) {
            return view('terminate')
        }

        return $next($request);
    }
}
```

In the above example, we check for a request parameter called 'terminate.' If this request parameter is present, we'll serve the `terminate.blade.php` view. If the parameter is not present, then the request will continue as before.

We've created the equivalent of a Laravel 3/4 'before' filter. This middleware has an opportunity to terminate a request **before** the response as been served. We can use any condition that we want, and serve any response that we want. For example, let's re-create the `auth` filter from the previous versions of Laravel. The auth filter will redirect to the login screen if a user isn't logged in.

First, we'll need to use the authentication component. Fortunately, we can use the dependency injection tricks that we have learned in previous chapters. You see, middleware is resolved through the Laravel container. Therefore, we can use the constructor to inject our dependencies.

Let's try this now.

Example 05: The authentication middleware.

```php
<?php

// app/Http/Middleware/MyMiddleware.php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Contracts\Auth\Factory as Auth;

class MyMiddleware
{
    /**
     * Authentication factory instance.
     *
     * @var \Illuminate\Contracts\Auth\Factory
     */
    protected $auth;

    /**
     * Inject our middleware dependencies.
     *
     * @param \Illuminate\Contracts\Auth\Factory $auth
     */
    public function __construct(Auth $auth)
    {
        $this->auth = $auth;
    }

    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure                  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($auth->guest()) {
            return redirect()->route('login');
        }

```

```
42          return $next($request);
43      }
44  }
```

First, we use the constructor-based dependency injection to inject our authentication factory. Next, we use the authentication factory to check whether the user is a guest (not logged in). If the user is a guest, we'll return a response object that will redirect to the login form. Otherwise, the request will continue as intended.

We've successfully recreated the authentication filter as a Laravel 5 middleware.

What about `after` filters? Do you remember those? In previous versions of Laravel, after filters were executed *after* the response had been served. This would give you an opportunity to modify the response, or to perform some 'clean up' operations at the end of the request/response cycle.

In Laravel 5, before and after filters are essentially the same thing. A single middleware. In fact, you're able to act *before* and *after* the request is served within a single middleware. Let's take a look at this now.

Example 06: A combined before and after filter in middleware form.

```
1   <?php
2
3   // app/Http/Middleware/MyMiddleware.php
4
5   namespace App\Http\Middleware;
6
7   use Closure;
8
9   class MyMiddleware
10  {
11      /**
12       * Handle an incoming request.
13       *
14       * @param  \Illuminate\Http\Request  $request
15       * @param  \Closure                  $next
16       * @return mixed
17       */
18      public function handle($request, Closure $next)
19      {
20          // Code to be executed *before* the response
21          // has been rendered, should go here.
22
```

```
        $response = $next($request);

        // Code to be executed *after* the response
        // has been rendered, should go here.

        return $response;
    }
}
```

Once again, this middleware will do nothing, but it does serve to identify where our 'before' and 'after' filter equivalents will sit. In the example above, pay close attention to the following line.

Example 07: Retrieve the response object.

```
$response = $next($request);
```

Instead of returning the result of the Closure, we capture its value into a `$response` variable. Why did we call it `$response`? At the end of the middleware chain, the final middleware will return an instance of the response and served by our controller or routed closure. It will be returned all along the chain until it becomes the result of our `$next` Closure.

After retrieving the response, we can modify it, examine it, or do whatever we want with it! We now know that the request has been handled, and we're in the 'after' space of the middleware. We could still terminate the chain by returning a different response object if we wanted, or we could just modify the one we now have access to. Let's add a header to the response object.

Example 08: Modify the response object.

```
<?php

// app/Http/Middleware/MyMiddleware.php

namespace App\Http\Middleware;

use Closure;

class MyMiddleware
{
    /**
     * Handle an incoming request.
```

```
13      *
14      * @param  \Illuminate\Http\Request  $request
15      * @param  \Closure                  $next
16      * @return mixed
17      */
18     public function handle($request, Closure $next)
19     {
20         $response = $next($request);
21
22         $response->header('My-Header', 'present');
23
24         return $response;
25     }
26 }
```

Here we have retrieved the response object, added a new header to it, and then served the response as the result of our middleware as intended. All of our responses will now have the `My-Header` header.

Middleware are far more useful than the filters of old. Within a middleware, we can execute as many 'before' and 'after' response actions as we require. In previous iterations of the framework, we'd have had to apply some individual filters to each route.

Our middleware classes need to be registered before they become active. There are two methods of registering middleware. Let's take a look at these now.

Global Middleware

Global middleware is executed on *every* request to the framework. These are defined on your instance of the HTTP kernel which you'll find tucked away in `app/Http/Kernel.php`.

Within this file, you'll find the following class property.

Example 09: Register global middleware.

```
/**
 * The application's global HTTP middleware stack.
 *
 * These middleware are run during every request to your application.
 *
 * @var array
 */
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
];
```

Our global middleware is defined within this array. Here we've already found the middleware that is responsible for terminating requests when 'maintenance mode' is switched on using the Artisan CLI. We're using the `::class` suffix to add the class names as strings to the array, rather than actual class instances.

We can add as many global middlewares as we like to this array. Don't forget where it lives! Middleware is extremely useful to have on your Laravel tool belt. See? You're pretty much like a PHP Batman now.

Route Middleware

Middleware can also be added to individual or groups of routes. For example, the authentication middleware that we created above wouldn't be useful if it were applied to every route. We'd result in an endless redirect loop when trying to visit our login page. Instead, we're better off applying it to individual routes or route groups.

Before a route middleware can be used, it must be registered. Once again, let's take a look inside our HTTP Kernal at `app/Http/Kernel.php`.

Example 10: Register route middleware.

```
/**
 * The application's route middleware.
 *
 * These middleware may be assigned to groups or used individually.
 *
 * @var array
 */
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
```

```
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
];
```

Here we have another class property array called `$routeMiddleware`. The keys for the array are a short name that we can use to apply our middleware to our routes. The value for each array key is, of course, a string representing our middleware class name.

You'll notice that Laravel has some middleware defined by default. That's right; there's even an `auth` middleware. It's true. I've tricked you once again. We didn't need to create it ourselves.

Forgive me? All right then, let's move on.

Once our middleware has been registered, there are some ways of assigning it to a route or group. Let's take a look at these examples now.

Example 11: Apply middleware using array syntax.

```
Route::group(['middleware' => ['auth']], function () {
    //
});

Route::get('/welcome', [
    'middleware' => ['auth'],
    'uses' => 'HomeController@index'
]);
```

As discussed in the advance routing chapter, our routes and route groups can be defined using an array of options. One such option is `middleware`. This can be used to define an array of middleware to execute, based on the names that we used to register them in our HTTP Kernel.

In the above example, we see a group and a single route being protected by the authentication middleware.

We can also register our middleware by chaining the `middleware()` method onto our routes. Here's an example.

Example 12: Apply the middleware using a method.

```
Route::get('/welcome', 'HomeController@index')
    ->middleware('auth');
```

As a parameter, provide either the name of a middleware or an array of names. Use as many as you like.

Hey, do you remember when I said you needed to register middleware before it can be used? Yeah? Well, I lied. Again. I need to see a psychiatrist. They are expensive, though, aren't they? Let's hope Code Smart sells well so that my next book can be sane and boring.

Anyway, instead of providing the short name for a middleware, you can provide the class name. Here's an example.

Example 13: Apply a middleware as a class name.

```
Route::get('/welcome', 'HomeController@index')
    ->middleware(App\Http\Middleware\MyMiddleware::class);
```

You can use this technique with any of the examples that you've seen above. It makes you wonder why I didn't teach this soon? We may never know.

Middleware Parameters

Would you like to use parameters within your middleware?

> Yes, please!

Tough, you're not allowed. Hah!

> Oh go on. I've had to put up with your lies.

Well, I can't argue with that, can I? Fine! Let's take a look at how we can use parameters in our middleware. First, we'll need placeholder parameters in our middleware method signature. Here's an example.

Note that parameters don't make sense with global middleware. Parameters are intended for use with route middleware.

Example 14: Allow for parameters within middleware.

```
<?php

// app/Http/Middleware/MyMiddleware.php

namespace App\Http\Middleware;

use Closure;

class MyMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure                  $next
     * @param  string                    $first
     * @param  boolean                   $second
     * @return mixed
     */
    public function handle($request, Closure $next, $first, $second = true)
    {
        // Do whatever we need to with $first and $second.

        return $next($request);
    }
}
```

All we need to do is place our parameters after the Closure within our `handle()` method. Here we've provided two parameters; the second one has a default value.

Now that our placeholder variables in place, we can provide parameters when registering our middleware to our routes. For example, if our middleware were registered under the name `mine`, then we'd use the following format to provide our middleware parameters.

Example 15: Apply a middleware with parameters.

```
Route::get('/welcome', 'HomeController@index')
    ->middleware('mine:hello,true');
```

Middleware parameters are separated by the middleware name using a colon `:` character. The parameters themselves are separated using a comma `,` character.

It's as simple as that. You can use parameters for conditionals, or use them to apply properties to the response object. The limit is your imagination.

Middleware Groups

When applying middleware to routes, you might find yourself applying big arrays of middleware to each route. Some middleware is more common than others. For this reason, Laravel has provided a means for grouping middleware together. This way, we can apply the entire group to a route or route-group with a single name.

You'll find the middleware group definitions in our old friend, the HTTP Kernel. Do you remember where it lives? It's over in `app/Http/Kernel.php`.

Example 16: Define middleware groups.

```
/**
 * The application's route middleware groups.
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
    ],

    'api' => [
        'throttle:60,1',
    ],
];
```

The `$middlewareGroups` multi-dimensional array can be used to nest some middleware under a single group name. For example, in the sample above we can see that Laravel has created a 'web' group with many middleware within.

We can apply this (or our own) middleware to routes by specifying the group name instead of a single middleware. Here's an example.

Example 17: Apply a middleware group to a route.

```
Route::get('/welcome', 'HomeController@index')
    ->middleware('web');
```

What's that `throttle` middleware, I hear you asking? Well, that's a middleware provided by the core of the framework to throttle API requests. Don't go looking for the middleware class in your application folder; you won't find it there!

In the next chapter, we'll be taking a look at service providers.

28. Service Providers

Service providers are a really important part of Laravel. At the start of the request, the application container is empty. That's no use to us, is it! We want access to all of those great services that Laravel has to offer. Service providers are what we use to let the container now how to create and retrieve services. They do all the heavy lifting before the framework gets to our routed code.

Registering Providers

Let's take the authentication system for example. Before we can use the `Auth` facade, or inject the `Auth` contract, we'll need the authentication service to be present within the container. It's added to the container during the framework bootstrapping process using a service provider. Don't believe me? That's fine; I have evidence!

Go ahead, open up the `config/app.php` file. Here you'll find a configuration key called 'providers'. Here is where all of the application's service providers are registered.

Example 01: Registered service providers.

```
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
    Illuminate\Bus\BusServiceProvider::class,
    Illuminate\Cache\CacheServiceProvider::class,
    Illuminate\Foundation\Providers\ConsoleSupportServiceProvider::class,
    Illuminate\Cookie\CookieServiceProvider::class,
    Illuminate\Database\DatabaseServiceProvider::class,
    Illuminate\Encryption\EncryptionServiceProvider::class,
    Illuminate\Filesystem\FilesystemServiceProvider::class,
    Illuminate\Foundation\Providers\FoundationServiceProvider::class,
    Illuminate\Hashing\HashServiceProvider::class,
    Illuminate\Mail\MailServiceProvider::class,
    Illuminate\Pagination\PaginationServiceProvider::class,
    Illuminate\Pipeline\PipelineServiceProvider::class,
```

```
20      Illuminate\Queue\QueueServiceProvider::class,
21      Illuminate\Redis\RedisServiceProvider::class,
22      Illuminate\Auth\Passwords\PasswordResetServiceProvider::class,
23      Illuminate\Session\SessionServiceProvider::class,
24      Illuminate\Translation\TranslationServiceProvider::class,
25      Illuminate\Validation\ValidationServiceProvider::class,
26      Illuminate\View\ViewServiceProvider::class,
27
28      /*
29       * Application Service Providers...
30       */
31      App\Providers\AppServiceProvider::class,
32      App\Providers\AuthServiceProvider::class,
33      App\Providers\EventServiceProvider::class,
34      App\Providers\RouteServiceProvider::class,
35
36  ],
```

There it is! We can see the `Illuminate\Auth\AuthServiceProvider::class` is being used to register our authentication services. We can also see all of the other services that belong to the framework. If we wanted to, we could strip services out of this array to increase the performance of the application slightly. I've done this in the past on smaller projects that demand fewer services.

You'll notice that the top half of the service providers exist within the `Illuminate` namespace, and belong to the framework as 'core' service providers. The bottom half of the array belong to our application, and you'll find them in your `app` directory.

Writing Providers

Let's take a look at a service provider class, shall we? Go ahead and open up the file at `app/Providers/AppServiceProvider.php`. Here's what we see.

Example 02: My first provider.

```php
<?php

// app/Providers/AppServiceProvider.php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        //
    }

    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

This service provider has been provided by the framework for our convenience. As you can see, it extends the `Illuminate\Support\ServiceProvider` class, which all service providers must do. We've got two methods that we can fill in.

The `boot()` method is executed first. It's very early into the framework bootstrapping cycle. Therefore, this is a great place to instantiate your service if you need to do things like register events. Defining services in this way is less common, however, so you'll probably find yourself using the `register()` method more.

The `register()` method is the best place for us to define our services. Here we'd bind our service into the container so that we can make use of it later. You'll find out how

to bind objects into the container in a future chapter, but for now, it's great to know where it can be used.

It's worth noting that you don't need to bind things into the container from this location. You can use ServiceProviders for other purposes too. This is because the service provider gives you full access to the container using the `$this->app` variable. As we've learned, the container contains all of our application services, so if we'd like to define a route, we could do it from a service provider. Here's an example.

Example 03: Register a route from a service provider.

```
/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->app->router->get('/foo', 'FooController@index');
}
```

The container allows access to its services via array or property access. So in the above example, we're accessing the `router` binding of the container and using it to define a route.

Of course, we have our own `routes.php` file, so I don't think this is the best place for us to define our routes. Then why did I show you this example?

Well, service providers form the 'entry-point' into your application for any third party packages you install. You'll find that whenever you install a composer package that is made for Laravel, you'll typically have to register a new service provider at the same time.

These packages don't have access to the `routes.php` file, but fortunately, they can access the router from their service provider if they need to define their routes. In fact, with full access to the application object, they can do pretty much anything that we could do within our application code. With great power, comes great responsibility. It's worth keeping this in mind when trusting packages, and registering their service providers.

Deferred Providers

If your provider is used only to register services within the container, you can make it more efficient by making it a deferred provider. This way, the service provider register

method will only be executed when the container looks for a service that we've defined in a `provides()` method.

Let's take a look at an example, shall we?

Example 04: A deferred service provider.

```php
<?php

// app/Providers/AppServiceProvider.php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Indicates if loading of the provider is deferred.
     *
     * @var bool
     */
    protected $defer = true;

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        //
    }

    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        $this->app->bind(\App\My\Service::class, function () {
            return new \App\My\Service;
        });
```

```
38      }
39
40      /**
41       * Get the services provided by the provider.
42       *
43       * @return array
44       */
45      public function provides()
46      {
47          return [\App\My\Service::class];
48      }
49  }
```

Woah, that's a lot to take in, isn't it? Let's run through it step by step.

Example 05: Defer the provider.

```
1  /**
2   * Indicates if loading of the provider is deferred.
3   *
4   * @var bool
5   */
6  protected $defer = true;
```

First, we must override the `$defer` property, and set it's value to try. By default, providers are not deferred, and so this value is set to false in the super class.

Example 06: Register services.

```
1  /**
2   * Register any application services.
3   *
4   * @return void
5   */
6  public function register()
7  {
8      $this->app->bind(\App\My\Service::class, function () {
9          return new \App\My\Service;
10     });
11 }
```

Next, we use the `register()` method to bind our service into the container using the class name as its key. We'll learn more about service binding in the container chapter.

Finally, let's take a look at the `provides()` method.

Example 07: List provided services.

```
/**
 * Get the services provided by the provider.
 *
 * @return array
 */
public function provides()
{
    return [\App\My\Service::class];
}
```

Here, we override the `provides()` method and return an array of all of the service keys that our provider registers. We've only registered one in this example, but we could return as many keys as needed. This is how the framework knows for which services to load your provider.

Well, that's most of what I wanted to cover in this chapter. There are a few other methods that make it easier to register configuration files and commands, but I want to save those for a 'developing packages' type chapter.

In the next chapter, we'll take a look at my favorite Laravel component. That's right; turn the page to learn about the container.

29. The Container

As we have discovered in the previous chapter. The container is the heart of Laravel. Think of it as the toolbox, and inside are all of our web developer goodies. It's the glue that holds the framework together.

A good understanding of the Laravel container will give you an extreme advantage when building things using the framework. You're going to have to take my work for it! This chapter might touch on some difficult concepts, but it's worth staying for the whole ride. Please make sure your arms and legs are inside the cart, and let's get started!

Useful Terms

There are a few terms that are useful to understand when working with the container. Let's look at 'bind' first.

The process of 'binding' or to 'bind' is the act of putting a service into the container. Placing a new tool into the toolbox. A Laravel developer might say "Let's bind this service within the container." Which translates roughly to "Let's put this object into the container".

On the opposite side of binding, we have the ability to 'resolve' things from the container. To resolve a service means to create or retrieve a service from the container.

From this point onwards, we'll be using the terms 'bind' and 'resolve' when talking about storing or retrieving things from the container. We're all on the same page now, aren't we?

The final thing to note is that the `Application` class that sits at the center of Laravel extends from the Container. Therefore, when we speak of Laravel's container, we're talking about the application instance.

Basic Usage

Let's first take a look at how we can bind and resolve basic types from the container. It's quite simple. We can access the container from anywhere using the `App` facade. Let's start simple. Here's an example of binding a class instance into the container.

Example 01: Bind a service within the container.

```
<?php

App::bind('panda', function () {
    return new Panda;
});
```

We use the `bind()` method to create blueprints for our services. This is normally achieved by passing a Closure to the second parameter of the method. The first parameter is, of course, the key that the service will be bound to. The beauty of the Closure is that it can be used to create the service on demand. Our controller will only execute the Closure when the service is requested. This is highly efficient; it means that we only ever bootstrap the blueprints for our services, and our services will only be held in memory when they are resolved for use. The Closure should always return an instance of the service that is to be bound.

In the above example, we've bound the blueprint for our `Panda` service under the name `panda`. We could use this key to resolve the service from the container. However, it's much better to use the name of the service class as a binding key. Here's an example.

Example 02: Bind a service within the container by class name.

```
<?php

App::bind(Panda::class, function () {
    return new Panda;
});
```

Here we use the `::class` suffix to bind the service under the fully qualified class name. Doing this will allow us to inject our service into the constructor of container-resolved classes, just as we discovered in the dependency injection chapter.

Now that we've learned how to bind our services, surely it's time to resolve them, don't you think? Resolving a service from the container is as easy as calling the `make()` method with the binding key. Let's resolve the Panda service that we've bound to our container, shall we?

Example 03: Resolve a bound service.

```
1  <?php
2
3  $panda = App::make(Panda::class);
```

Once again we pass the binding string as the first parameter to the `make()` method. Why is this method not `resolve()`? Great question. I have no answer. Sometimes the world is just like that!

When we call the `make()` method, our bound Closure is executed, and the resulting value is returned. We receive an instance of our `Panda` class. Binding and resolving services are as simple as that. It's not so difficult really, is it?

The best place to bind a service is from within a service provider. All service providers which extend the `Illuminate\Support\ServiceProvider` class have access to a class property called `$app` which is an instance of the application container.

Singletons

In the previous example, each time that we make a call to `make()`, we'd receive a different instance of the Panda class. Sure, it would be the same class blueprint, but each time it would be a new instance, using more memory.

When it comes to software design patterns, the singleton pattern is used to ensure that we have only one instance of a class at any one time. Singletons are great for services since it's unlikely that we need different instances. For example, when creating a database connection, you wouldn't want to set up the database configuration for the instance each and every time, would you? Instead, it would be better to create that service once, and use it whenever we need it.

In PHP, true singletons are created by designing classes with a static method to retrieve or instantiate a static instance. It's a complicated process for beginners and requires a lot of boilerplate. The Laravel container offers a pseudo-singleton method of binding services into the container. Let's take a look at an example.

Example 04: Bind a singleton service.

```
<?php

App::singleton(Panda::class, function () {
    return new Panda;
});
```

The only change that we've made to the initial binding process is to change the name of the method used from `bind()` to `singleton()`. As you can see, the method still accepts both a key and a Closure, and will still return an instance of our Panda service.

We can resolve singleton services from our container the same way that we would create a bound instance, using the `make()` method.

Example 05: Resolve singleton service.

```
<?php

$panda = App::make(Panda::class);
$secondPanda = App::make(Panda::class);
$panda === $secondPanda; // true
```

This time, however, the container will resolve our panda instance, and cache it on the container. The second time that we use `make()` to create the service, we'll receive the *exact* same instance as the first time. This means that we can only ever have one instance of our `Panda` service. It is, in that sense, a singleton.

Bound Instances

You don't have to create a blueprint when binding your services into the container. For example, let's say that your service needs to listen for events. If you were to bind the service using the closure, then it's listeners would only be executed when the service is requested and created. Some events might have already fired by this time. In this circumstance, it would be better to instantiate our service first, and then store it in the container.

Binding instances is much simpler than binding blueprints. Let's take a look at an example now.

Example 06: Bind an instance into the container.

```php
<?php

// First create the service instance.
$panda = new Panda;

// Next, bind it within the container.
App::instance(Panda::class, $panda);
```

This time, we use the `instance()` method, passing our binding key and the instance that we wish to store in the container. It's as simple as that.

Once again, we can use the `make()` method to retrieve the instance from the container. Much like the singleton binding, we will always receive the *exact* same instance from this method.

Example 07: Resolve our service.

```php
<?php

$panda = App::make(Panda::class);
```

Class Resolution

The container is multi-talented. We've already discovered how to resolve bound services from the container, but did you know that it can also be used to instantiate classes?

For example, consider the following.

Example 08: Contained-based instantiation.

```php
<?php

$panda = App::make('My\Red\Panda');
```

Should we try to resolve a binding that doesn't exist within the container, the container will next attempt to determine whether the string represents a class name. If a class with the binding name exists, the container will attempt to instantiate it, and return it from the `make()` method.

So why is this more useful than the traditional method?

Example 09: Traditional instantiation.

```
<?php

$panda = new My\Red\Panda;
```

Well, there are many reasons. First of all, cast your mind back to the dependency injection chapter. We learned that classes that are instantiated from the container have their constructor dependencies injected automatically, and that of their dependencies' dependencies. That's the most useful feature for me.

There is, however, another reason. Consider the following.

Example 10: Our social function.

```
<?php

function getTweets()
{
    $twitter = new Twitter\Api\Client;
    return $twitter->getTweets();
}
```

Here we have a function that uses an instance of a Twitter API client to retrieve our tweets. Of course, the code is imaginary, but it should make perfect sense.

Let's say that we want to test this method. We wouldn't want it to hit the Twitter API would we? Our tests should execute in isolation, and have a determinable outcome. We can't guarantee the existence of the Twitter API, can we?

Our `getTweets()` function is now bound to an instance of the Twitter client. They are inseparable. This is a shame because we'd have to mock the entire function if we wanted to test it, and that's not an easy thing to do.

Let's make some modifications to our little function.

Example 11: Use container class instantiation.

```
<?php

function getTweets()
{
    $twitter = App::make(Twitter\Api\Client::class);
    return $twitter->getTweets();
}
```

Now we're creating our API client using the container. Why is this an advantage, do you think? Let's approach this from a testing angle. To test this function, we'll want to reduce the number of outside dependencies. We'll want to replace our Twitter client for a dummy version; that supplies mock tweets instead of calling the API.

Fortunately, the Laravel container makes it easy to replace bindings. Actually, in this instance, there's no binding to replace, but we can certainly create one. Consider the following code snippet.

Example 12: Swap for a dummy client.

```
<?php

function getTweets()
{
    $twitter = App::make(Twitter\Api\Client::class);
    return $twitter->getTweets();
}

// Create an instance binding for the twitter client.
App::instance(Twitter\Api\Client::class, new DummyTwitterClient);

$tweets = getTweets();
```

We use the `instance()` method to bind our dummy Twitter client under the name of the real Twitter client. This means that when the class is resolved through the container, it will be our dummy client instead. This means that we receive our predictable dummy twitters from the `getTweets()` method, and are now able to test it.

If we had been resolving a bound Twitter service from the container, instead of class resolution, we could have replaced the binding in the same way. Each time you `bind()`, `instance()` or `singleton()`, it will replace the original binding.

Implementation Binding

Now this is a slightly more complicated topic, but it's an extremely powerful one. It requires a little understanding of interfaces. Consider an adapter pattern, where adapters for a number of providers adhere to a common interface.

For example, we might have an `App\SocialProvider` interface, with the `App\TwitterSocialProvider` implementation class. The implantation adheres to a `messages()` method as defined by the interface. This method can be used to retrieve the social messages from Twitter.

Instead of using the `App\TwitterSocialProvider` class, we'll do something a little clever. Let's be forward-thinking, shall we? Instead of binding our Twitter social provider to its class name, we'll bind it to the interface. Here's an example.

Example 13: Bind an implementation to an interface.

```
<?php

// app/Providers/AppServiceProvider.php

namespace App\Providers;

use App\SocialProvider;
use App\TwitterSocialProvider;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        //
    }

    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
```

```
29      {
30          $this->app->bind(
31              SocialProvider::class,
32              TwitterSocialProvider::class
33          );
34      }
35  }
```

We use our application provider to bind our Twitter social provider under the name of the social provider interface. Of course, it's not possible to instantiate an interface, so when we try to resolve the interface from the container, we instead receive an instance of our Twitter social provider.

This means that we can type-hint the interface in our container-resolved classes (such as controllers, for example) to inject the Twitter service. Here's an example.

Example 14: Inject social provider using interface.

```
1   <?php
2
3   namespace App\Http\Controllers;
4
5   use App\SocialProvider;
6
7   class SocialController extends Controller
8   {
9       public function index(SocialProvider $social)
10      {
11          // Use the twitter service.
12      }
13  }
```

The parameter `$social` is an instance of our Twitter social provider.

> Isn't that more confusing. Shouldn't we name things after what they are?

I suppose you might see it that way, but the Twitter instance is an actual social provider instance. It's not as untrue as it initially seems. The context might be a little looser.

However, there's a good reason why we've done this. If you're a programmer, then there's every chance that you have a boss or someone that you report to. The

wonderful attribute about these people, is they have the ability to change their mind at any moment.

"Hey, computer guy. Our product people have just told me that this thing called Facebook is the better social network. Honestly, I've never heard of it before, but, I do like money. Can we use Facebook instead of Twitter? Oh, and by "can we", I mean make it happen or you're fired." - *PHP CEO*

Does that sound familiar? I thought so. Luckily for us, we took the time to use implementation binding. We quickly create a new class called `FacebookSocialProvider` that is an implementation of the `SocialProvider` interface.

Because our new class shares a common implementation, we know that it's entirely compatible with our `TwitterSocialProvider` class. Now that we've created it, we need only update our service provider to use the new implementation. Here's an example.

Example 15: Swap the social provider implementation.

```
<?php

// app/Providers/AppServiceProvider.php

namespace App\Providers;

use App\SocialProvider;
use App\FacebookSocialProvider;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        //
    }

    /**
     * Register any application services.
     *
     * @return void
     */
```

```
28      public function register()
29      {
30          $this->app->bind(
31              SocialProvider::class,
32              FacebookSocialProvider::class
33          );
34      }
35  }
```

The only change that we've made is to bind our `FacebookServiceProvider` to the implementation in the place of our Twitter equivalent. We're now done with our task. You can now play Uncharted 4 with your feet on the desk for a few months, and our CEO will be none the wiser.

You see, we used the interface to inject our service whenever it was used. This means that our Facebook provider will automatically be injected in the place of our Twitter provider all across our application. A single line code change. Isn't that fantastic? Isn't it powerful?!

Contextual Binding

We're now more than aware that the Container can instantiate classes, and resolve their dependencies automatically. But what if it chooses the wrong ones? Perhaps we want to inject a different implementation into our class' controller.

Don't worry; the container has you covered. Let's lead with an example, as always.

Example 16: Contextual binding.

```
1  <?php
2
3  // Contextually bind a dependency.
4  $this->app->when(App\SocialManager::class)
5      ->needs(App\SocialProvider::class)
6      ->give(App\FacebookSocialProvider::class);
```

Let's walk through this chained method sequence. The first method, `when()`, is used to specify the class that we're attempting to resolve using the container. The next method, `needs()`, defines the type-hinted parameter in that classes, constructor. Finally, the `give()` method defines the instance that will be given in place of what was specified within the `needs()` method.

This means that when our container attempts to instantiate the `App\SocialManager` class, an instance of the `App\FacebookSocialProvider` class will be injected where the `SocialProvider` interface was type-hinted.

We can use as many contextual bindings as we need to ensure that our class can be resolved in the form that we intended.

Tagging Bindings

Sets of bindings can be tagged within the container so that the services that they have bound can be retrieved in bulk. This is a great way of registering extensions for your application. For example, let's assume that we're building a CMS. Often, CMS systems use plugins to enhance their functionality.

First, we'll bind our plugins into the container. Here's a simple example.

Example 17: Bind our CMS plugins.

```
<?php

// Bind a markdown parser plugin into the container.
$this->app->bind(Plugin\Markdown::class, function () {
    return new Plugin\Markdown;
});

// Bind a Twitter plugin into the container.
$this->app->bind(Plugin\Twitter::class, function () {
    return new Plugin\Twitter;
});
```

Here we've bound two plugins into the container. There's nothing new here. Let's try something new, shall we?

Example 18: Bind our CMS plugins and tag them.

```
<?php

// Bind a markdown parser plugin into the container.
$this->app->bind(Plugin\Markdown::class, function () {
    return new Plugin\Markdown;
});

// Bind a Twitter plugin into the container.
```

```
$this->app->bind(Plugin\Twitter::class, function () {
    return new Plugin\Twitter;
});

// Tag a single service.
$this->app->tag(Plugin\Markdown::class, 'cms.plugins');

// Tag multiple services.
$this->app->tag([
    Plugin\Markdown::class,
    Plugin\Twitter::class
], 'cms.plugins');
```

Now we've tagged our plugins under the name `cms.plugins`. Our application can use this name to retrieve all instances that are bound to that tag. Here's an example.

Example 19: Retrieve tagged services.

```
<?php

// Retrieve an array of plugins from the container.
$plugins = App::tagged('cms.plugins');
```

We use the `tagged()` method to retrieve an array of all services that are bound under that tag. Using this method, a third party package would be able to register its own plugin by tagging it within the container using the name `cms.plugins`. This would allow the plugin to register itself automatically. Isn't that useful?

Resolution Methods

There are many ways to resolve a service from the container. Let's quickly step through them all, so that we understand them.

The first method is constructor-based (or action-based) injection, which we discovered in the dependency injection chapter. Here's an example.

Example 20: Use dependency injection.

```php
<?php

namespace App\Http\Controllers;

use App\SocialProvider;

class SocialController extends Controller
{
    public function index(SocialProvider $social)
    {
        // Use the twitter service.
    }
}
```

The next method is to use the `App` facade. Here's another example:

Example 21: Use the App facade.

```php
<?php

$service = App::make('service');
```

Next, we have the `$app` class property; that is present on all Service Provider classes.

Example 22: Use the service provider application instance.

```php
<?php

// app/Providers/AppServiceProvider.php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
```

```
15      */
16     public function boot()
17     {
18         //
19     }
20
21     /**
22      * Register any application services.
23      *
24      * @return void
25      */
26     public function register()
27     {
28         $service = $this->app->make('service');
29     }
30 }
```

In the global context, an `$app` variable is available as a reference to the application container. Since the container implements the `ArrayAccess` interface and supports a magic method for `__get()` to access services, we have some equivalent ways of accessing our services. Here are the possibilities.

Example 23: Use the $app global variable.

```
1  <?php
2
3  // Use the make method.
4  $service = $app->make('service');
5
6  // Use array access.
7  $service = $app['service'];
8
9  // Use the magic accessor.
10 $service = $app->service;
```

All of the methods above will result in the resolution of our service.

Finally, we have access to the `app()` global function. This method can be used to retrieve an instance of the application container. It can be used in a number of ways. Let's take a look at some examples.

Example 24: Use the app() function.

```php
<?php

// Use the make method.
$container = app();
$service = $container->make('service');

// Use the function parameter.
$service = app('service');

// Use array access.
$service = app()['service'];

// Use the magic accessor.
$service = app()->service;
```

Where you can, I would suggest favoring dependency injection. It's the most extensible solution. Failing that, I would suggest using either a container instance on the service provider or the `App` facade. However, each of the methods above is useful and have their own reason for existing. Use them to your advantage!

30. Coming Soon

Oh dear, you've gone and reached the end of Code Smart. But wait! This isn't the end at all. You see, Code Smart is being continually updated with new content and topics. You'll receive an email when the next topic is available to read. I'm committed to writing *at least* one chapter a week. You won't have to wait long!

If you'd like to be more involved with the future of the book, then perhaps you could vote on the next topic. Below, you'll find some chapters that I intend to write next. Tweet to @daylerees[1] with the hashtags `#codesmart` and `#vote`, along with a topic title, to register your vote for the next chapter. If you'd like to see me cover a chapter that isn't on the list, then please send an email to me@daylerees.com[2].

- Queued Jobs
- Mail
- Elixir
- Cache
- Session
- Redis
- Task Scheduling
- Localisation
- Authentication

Once again, I'd like to thank you for purchasing and reading Code Smart. Writing these books isn't my day job. By supporting my writing, you're allowing me to spend more time on the books, and to help more people learn development skills.

Dayle.

[1] https://twitter.com/daylerees
[2] mailto:me@daylerees.com