CODE BRIGHT

WEB APPLICATION DEVELOPMENT WITH THE LARAVEL FRAMEWORK VERSION 4 FOR BEGINNERS

BY DAYLE REES

Laravel: Code Bright

Web application development for the Laravel framework version 4 for beginners.

Dayle Rees

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

This version was published on 2013-06-17

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 just bought Laravel: Code Bright by @daylerees . http://leanpub.com/codebright

The suggested hashtag for this book is #codebright.

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/#codebright

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 both books! Love you Emma!

Taylor Otwell, the last year has been incredible, thank you for giving me the opportunity to be part of the team, and for your 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 it's development. I have really enjoyed working with you on this new version of Laravel, and hope to work with you again on future projects!

Eric Barnes, Phill Sparks, Shawn McCool, Jason Lewis, Ian Landsman, thanks for all the support with the framework and for being good pals.

Thanks to my parents, who have been supporting my nerdy efforts for close to twenty eight year! Also thanks for buying a billion copies of the first book or so for family members!

Thank you to everyone who bought the first book Code Happy, and all of the Laravel community. Without your support a second title would have never happened.

Errata

This may be my second book and my writing may have improved since the last one, but I assure you that there will be many, many errors. You can help support the title by sending an email with any errors you have found to me@daylerees.com[1] along with the section title.

Errors will be fixed as they are discovered. Fixes will be released within 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] or send a tweet to @daylerees. I will endeavour to reply to all mail that I receive.

[2] mailto:me@daylerees.com

Translations

If you would like to translate Code Bright into your language please send an email to me@daylerees.com[3] with your intentions. I will offer a 50/50 split of the 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.

[3] mailto:me@daylerees.com

Introduction

Well, it's sure been a long time since I've written a book chapter. Code Happy was released almost 12 months ago and amassed a total of nearly three thousand sales. Let's see if I still remember how to do this "writing" thing.

If you have read the previous title you will already know that I am firstly a developer, and secondly a writer. For this reason you won't find any long words in this book. Nothing that would impress Shakespeare (apart from the spelling errors?). What you **will** get is straight talking, simple to understand information about the Laravel framework. You will also get passion! Not the sweaty bed sheets type of passion, but enthusiasm for the Laravel framework which cannot be rivalled. I like to write my books as if I were standing right in front of you having a conversation. In fact, if you really want a conversation then come and see me in the Laravel IRC channel!

Now it's time for one of those little 'about the author' paragraphs. No one really wants to read it, but it never hurts to stroke the ego a little, does it?

My name is Dayle Rees (it says so on the cover!) and I am a web developer and a design enthusiast. I come from a little town on the coast of Wales called Aberystwyth. At the time of writing my last book 'Code Happy' I worked for the National Library of Wales in Aberystwyth, which is one of three copyright libraries in the United Kingdom.

I have since moved to Cardiff, which is the capital city of Wales and have started working with BoxUK. BoxUK is an internet consultancy and development organisation in which I get to work with a team of developers who are passionate about the world of web development.

Web development isn't just my work, it's also my hobby. I enjoy finding useful and interesting pieces of code or beautiful designs. I believe that our skills let us do wonderfully creative things, and I love seeing ideas come to life.

A little over a year ago I began helping the Laravel community with bundles of code, web designs, and helping out any way that I could. Since then my involvement has increased. Laravel is now my primary open source project and I am now a member of the core development team for the framework.

With Laravel 4 (codenamed Illuminate) my involvement has hit a new high. I have been working alongside Taylor Otwell to make this release the best framework that you will have ever used. Don't take my word for it! Start using it and thank us later when you can't stop smiling while coding.

Laravel is an example of how even a development tool can be creative. Laravel's beautiful syntax straight from the genius mind of Taylor Otwell cannot be rivalled. It enables us to write code that will read like nerd poetry, and will allow us to enjoy our coding tasks.

> So Dayle, what has changed since the last release of the framework?

The simple yet confusing answer is everything and nothing!

Laravel 4 has been rewritten from the ground up, allowing for increased flexibility and testability along with a billion (not entirely accurate - don't count them) new features. Where Laravel 3 gave you some freedom with how to structure your code, Laravel 4 will allow hackers to go wild and change the framework to suit their needs.

When I hear that something has improved I am always looking for a catch, but with Laravel 4 there isn't one. It still has the beautiful and expressive syntax you love; you might even find that you love it more!

So why did you write a new book?

Code Happy covered the version stretch between 3.0 and 3.2.x, and with nearly three thousand copies sold I must have done something right. Sure, I could have probably re-worked the entire book to work with Laravel 4. However, this version of the framework is reinvention. If I was to update the book you would lose all the information about version 3 which I believe is still great framework. Many people will have large projects based on Laravel 3 and I think that they should have access to the information in Code Happy if they need it.

Then there's my own experiences. I have learned a lot about writing a book since finishing Code Happy. I have learned about my common mistakes, which I can now avoid. I can improve on what I have already done, and I hope to do so.

I didn't read Code Happy! Should I read that first?

You can if you want to, I put some funny jokes in there. However this book is one for beginners, and so we will start from the very basics. If you have already been using Laravel go ahead and skip to the interesting bits to see what has changed. If you are new to the framework then I would suggest you stick with me and read it from cover to cover. Don't worry! I will try to keep it interesting. Soon you will be building wonderfully expressive PHP applications with Laravel.

When will the book be complete?

As with my previous title, this book is a published while in progress title. It means that you get each chapter as I write it. In its current state the book may not be complete, but as I add additional chapters you will receive an email and will be able to download the updates for free.

I feel that this method of writing offers a great deal of flexibility. I can be relaxed about my writing knowing that I can change something easily if I have got it wrong. By not rushing to hit a deadline I can write a book that I feel will be of greater quality. I can update the title for future versions, or to highlight additional information. You will be able to access the content faster. It also allows me to release the title alongside the launch of the new version of the framework.

> I have run out of questions..

Good! By now you should be eager to start the learning process. Jump right in and start enjoying Laravel. Feel free to send me a tweet or a message on IRC if you want a chat!

The Primers

> Hey this chapter wasn't in Code Happy!?

As a true fan of Code Happy I just can't put anything past you! Well done loyal reader!

You see, Laravel 4 uses a number of new technologies regularly. These technologies can easily be taught on their own and parallel to the framework. With this in mind, I thought it might be best to open with a chapter about this new tech to 'prime' you for your learning experience.

Experienced web developers may have already come across these technologies, or have already been using them. You are welcome to skip any of the primer chapters if you want to. I won't be upset. No really… go on. You don't need me anymore..

If you are still reading I'm going to assume you are my friend. Not like those traitors who skipped straight to the routing chapter!

Let's jump right in to our first primer to talk about PHP namespaces.

Namespaces

In PHP version 5.3 a new feature known as namespacing 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.

In PHP 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 really simple class.

```php
<?php

// app/models/Eddard.php

class Eddard
{

}
```

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

```php
<?php

// app/routes.php

$eddard = new Eddard();
```

> Dayle, I know some PHP...

Okay, okay sorry. Basically, 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 Namespacing

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

```php
<?php

namespace Stark;

// app/models/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.

Now, when we try to use the 'Eddard' class once again.

```php
<?php

// app/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.

```php
<?php

// app/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:

```php
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.

```php
<?php

namespace Stark;

// app/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. Now because we are inside the same namespace as the one we put 'Eddard' into, this time we receive the namespaced 'Eddard' class. See how it's all relative?

Now that we have changed 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 within the global namespace, we simply prefix them with a backward (\) slash.

```php
<?php

// app/routes.php

$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?

```php
<?php

namespace Stark;

// app/routes.php

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

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.

```php
<?php

namespace Stark;

use Tully\Edmure;

// app/routes.php

$edmure = new Edmure();
```

Using the `use` statement, we can bring one class from another namespace, into the current namespace. Allowing us to instantiate it by name only. Now don't ask me why it doesn't need the backward slash prefix, because I just don't know. This is the only exception that I know of. Sorry about that! You can prefix it with a slash if you want to though, you just don't need to.

To make up for that horrible inconsistency, let me show you another neat trick. We can give our imported classes little nicknames, like we used to in the PHP playground. Let me show you.

```php
<?php

namespace Stark;

use Tully\Brynden as Blackfish;

// app/routes.php

$edmure = new Blackfish();
```

By using the ‘as‘ keyword, we have given our ‘Tully/Brynden’ class the ‘Blackfish’ nickname, allowing 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:

```php
<?php

namespace Targaryen;

use Dothraki\Daenerys as Khaleesi;

// app/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 are able to 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.

```php
<?php

namespace Targaryen;

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

Structure

Namespaces aren't just about avoiding conflicts, we can also use them for organisation, and for 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 can avoid causing hassle for the wonderful, open source embracing, individual.

```
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 a number of sub-namespaces to organise 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 really a bug.

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

In Java for example, you are able to import a number of 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.

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

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

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

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

```
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:

```
<?php

namespace Baratheon;

use Dayle\Blog as Cms;

// app/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, but JSON strings. Just turn the page and you'll see what I mean!

JSON

What is JSON?

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 normally achieved using JavaScript with an AJAX request.

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

Since version 5.2.0, PHP has been able to serialize 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() to transform the string into a new instance containing the original value.

It's roughly what we will be using JSON for. However, the advantage is that JSON can be parsed by a variety of different languages, where as `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 serialized strings will look like garbage.

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

JSON Syntax

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

Yay JSON! Okay, so 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 which might make it a little more difficult to read.

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

```
{
    "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 from prying eyes.

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

```
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:

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

Well, 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 just heard that JSON doesn't have variables. This is fine since we can use strings as keys, and that's exactly 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 with 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.

```
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 before, 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 once again look very similar to their JavaScript counterpart.

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


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

***Note** 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. Once again 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!

```
{
    "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 I mentioned earlier, since version 5.2.0 PHP has provided support for serializing and unserializing data to and from the JSON format. Let's go ahead and have a look at that in action.

Serialize a PHP array to JSON

To serialize a PHP value we need only use the `json_encode()` method. Like this:

```
<?php

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

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

```
{"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.

Unserialize a PHP array from a JSON string

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

```php
<?php

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

Awesome! We go... wait, what?

```
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.

```php
<?php

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

// Awesome!
```

Great! That's what we wanted. If we wanted an array then PHP provides a number of 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()`!

```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 a new package manager for PHP. When we start looking at Composer you will understand why a knowledge of JSON is so important.

Composer

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 specific version and it would be installed on your system. However, the dependency is linked to PHP rather than your application itself. This means 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?

First of all, let's forget 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, and describe it.

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

Configuration

You learned JSON in the last chapter, right? So 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? Well, for that reason, why can't we use it within flat files to provide configuration? That's exactly 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 along with 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.

```
{
    "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. Normally you would 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 anyway. 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 I would recommend checking out the Composer website[4] which contains a wealth of information and documentation.

I also found this handy cheat sheet[5] online, which may be useful for newcomers to Composer when creating new packages. Mouse over each line to discover more about the configuration items.

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

```
"name": "marvel/xmen",
```

This is the package name. If you have used Github[6] 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 tend to use

[4] http://getcomposer.org/

[5] http://composer.json.jolicode.com/

[6] http://github.com

their Github username as the owner, and I fully agree with this notion. You can, however, use whatever 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 simple and descriptive. Once again, many developers choose to use the repository name for the package when hosted on Github, and once again I fully agree with this system.

```
"description": "Mutants saving the world for people who hate them.",
```

Provide a brief description of the functionality of the package. Remember, 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.

```
"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.

```
"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? Whatever 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.

```
"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 of the fact that 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!

```
"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 a great deal 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.

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

Use an object to represent each individual 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!

```
"name": "Stan Lee",
```

I don't really 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.

```
"email": "stan@marvel.com",
```

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

```
"homepage": "http://marvel.com",
```

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

```
"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, now let's look at something more interesting. Dependency management!

Dependency Management

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

```
{
    "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 really lives though!

We know that the X-Men will depend on:

- Wolverine

- Cyclops
- Storm
- Gambit

There are loads of others, but these guys are pretty 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 changes. That could get really boring. Let's add them to the `require` section so that Composer will manage them for us.

```
"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[7] or Bitbucket[8]. 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 command:

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

With this we have created version `1.0.0` of our application. This is a stable release which people can depend on.

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

```
"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` user.

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

[7] http://github.com
[8] http://bitbucket.org

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?

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.

Even better news, is that 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 a number of 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:

```
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.

```
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`.

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

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

```
"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:

```
"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 sky diving, then you might want to use bleeding edge versions. Composer is able to target `branches` of a repository using the following syntax.

```
"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.

```
"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.

```
"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. I might come back to this chapter later and tackle that topic, but for now I'd suggest looking at the Composer documentation for package versions[9] to find additional information on the topic.

[9] http://getcomposer.org/doc/01-basic-usage.md#package-versions

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 your back 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[10] 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`.

```
"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 above, the `require-dev` section uses exactly 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.

```
"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.

```
"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.

[10] http://codeception.com/

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

This section indicates packages that have been provided within the codebase of your package. If the Gambit packages source was 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.

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

Your package might have a number of 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.

Well 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 for 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 own autoload, which can be used by your application to load class definitions.

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

```
"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 simplest of loading mechanisms, the `files` method.

```
"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 larger amounts of files.

```
"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 a final loading mechanism, and the best of all, which is intelligent enough to not require a map. Let's first learn about PSR-0 class loading.

PSR-0 class loading was first described in the PSR-0 PHP standard, and provides a simple way of mapping PHP name-spaced 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:

```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` namespaced `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.

```php
<?php

namespace Super\Happy\Fun;

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

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

This class would be located at the following 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 exactly 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.

```
"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 a 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 actually need the trailing slash, but many examples like to use it to denote a directory.

This next bit is very important, 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.

```
<?php

namespace Super\Happy\Fun;

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

Well, 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.

```
"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.

```
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!

You might wonder why I put my namespaces files in a `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.

```
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 its own locations for classes which we will describe 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

Now you might be wondering why I chose to cover installation and usage at the end of this chapter? I feel that having a good knowledge of the configuration will help you understand what Composer is doing behind the scenes as we are using it. Let me know!

The following installation methods will be specifically for unix based development environments such as Linux or Mac OSX. I'm hoping that Taylor might be able to edit this chapter and provide information on installing Composer in a Windows environment, since I avoid that particular operating system like the plague.

Composer is a PHP based application, so you should have the CLI client for PHP installed before attempting to use it. Double check this by running the following command.

```
php -v
```

If PHP has been installed correctly, you will see something similar to..

```
$ php -v
PHP 5.4.4 (cli) (built: Jul  4 2012 17:28:56)
Copyright (c) 1997-2012 The PHP Group
Zend Engine v2.4.0, Copyright (c) 1998-2012 Zend Technologies
    with XCache v2.0.0, Copyright (c) 2005-2012, by mOo
```

If the output states that you are using anything less than PHP version 5.3.2, then you aren't going to be able to use Composer until you upgrade your PHP version. In fact, if you are using anything less than PHP 5.3.7 then you won't be able to use Laravel at all.

You can use `CURL` to download Composer. Mac OSX comes with it by default. Many Linux distributions will have CURL within their software repositories, if it has not already been installed as standard. Let's use CURL to download Composer's executable.

```
curl -sS https://getcomposer.org/installer | php
```

Experienced linux users may be concerned that CURL is piping the installation script into PHP. This is a fair concern, but the Composer installation has been used by thousands of developers and has been proven to be secure. Don't let it put you off using this lovely piece of software!

Assuming the installation process was completed successfully (it will tell you), you will now have a `composer.phar` file in your application directory. This is the executable that can be used to launch Composer, for example..

```
php composer.phar
```

..will show you a list of commands that are available to you.

Now, you could continue to use Composer this way, but I would suggest installing it globally. That way it can be used across all of your Composer projects, and will have a shorter command to execute it.

To install Composer globally, simply move it to a location within your `PATH` environmental variable. You can see these locations by using the following command.

```
echo $PATH
```

However, `/usr/local/bin` is an acceptable location for most systems. When we move the file, we will also rename it to `composer` to make it much easier to launch. Here is the command we need:

```
sudo mv composer.phar /usr/local/bin/composer
```

Having installed Composer globally, we can now use the following shorter syntax to see the same list of commands. This command is also executable from any location on the system.

```
composer
```

Hey, that's a lot cleaner isn't it? Let's start using this thing.

Usage

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

```
{
    "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 setup our autoloader.

```
composer install
```

The output we get from Composer will be similar to:

```
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!

So why are there seven packages installed when I only listed four? Well you are forgetting 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 to? 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 own 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:

```
<?php

require 'vendor/autoload.php';

// Your awesome application bootstrap here!
```

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

```
<?php

$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 is able to load it.

```
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.

```
{
    "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.

```
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.

```
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 versions 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, why not find out about updating them? For example if we had the following requirement in our `composer.json` file.

```
"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?

Well, 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.

```
$ 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:

```
composer update xmen/gambit
```

Wait, what's that `(including require-dev)` bit mean? Well you must remember the `require-dev` section in the `composer.json` file where we list our development only dependencies? Well, Composer expects the update command to be only executed within a safe 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.

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

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

```
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.

```
sudo composer self-update
```

Well, 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, be sure to let me know!

Architecture

I want to make Code Bright a more complete learning experience, and for that reason, I have decided to include this chapter. If you are busting to get started with your coding, feel free to skip it. However, I think it will be really useful down the line to learn how Laravel is constructed.

In Laravel 3, the IoC container was a component that confused a lot of people. In Laravel 4, it is the part of the framework that holds everything together. I can't begin to express its importance. Well, actually, I need to for this chapter to make sense. Let's give it a go.

The Container

In Laravel 4, the container gets created early during the bootstrap process. It's an object called `$app`. So when we talk about a container, what comes to mind?

Let's have a look at the dictionary definition. Why? I'm not sure… I've seen other authors do it though. I don't want them to look down on me. Here we go.

> **Noun** - Container. An object used for or capable of holding, esp for transport or storage, such as a carton, box, etc.

I think the description fits the Laravel container component rather well. It's certainly used to store things. Let's be honest… if it didn't store things, then 'Container' would be a terrible name. As for transport, it certainly makes it easier to 'move' other components around the framework when we need to access them.

It's not really a carton, or a box. It's more of a keyed storage system.

Once the `$app` object has been created, you can access it from anywhere using the `app()` helper method. In all honesty, you probably won't need to. This is due to a feature that we will explain later in this chapter.

Let's get a hold of the app object and take a look at it. We can do this in our `app/routes.php` file.

```php
<?php

// app/routes.php

var_dump($app);

die();
```

Note that we don't need to use the `app()` helper within the `app/routes.php` file. It's available in the global scope, so we can access it directly.

Woah, that's a huge object! Well I guess it should be... it's the core of the framework. You will notice that it's an instance of `Illuminate\Foundation\Application`. All of the Laravel 4 components live within the `Illuminate` namespace. It was a codename used during the early stages of the framework's design, and we just don't have the heart to remove it. Think of it as Laravel "illuminating" the world of PHP!

If we take a look at the source for the `Application` class, we will see that it extends `Container`. Go ahead, you can find the source at:

https://github.com/laravel/framework/blob/master/src/Illuminate/Foundation/Application.php[11]

The `Container` class is where all the magic happens. It also implements the `ArrayAccess` interface. This is a special interface that allows the attributes of an object to be accessed as an array in a similar fashion to that of JavaScript object literals. For example, the following two methods of accessing object attributes would both be possible:

```php
<?php

$object->attribute = 'foo';
$object['attribute'] = 'foo';
```

During the bootstrap of the framework, a number of service provider classes are executed. These classes serve the purpose of registering the framework's individual components within the application container.

So what do I mean by a component? Well, a framework is made up of many different parts. For example, we have a routing layer, a validation system, an authentication layer, and many more packages of code which handle a specific function. We call these our framework components, and they all fit together within the container to create the complete framework.

If you are a first time user of Laravel, then this next example won't make a great deal of sense to you. Don't dwell on it! It's just something for Laravel 3 users to think about. Take a look at this.

[11] https://github.com/laravel/framework/blob/master/src/Illuminate/Foundation/Application.php

```php
<?php

// app/routes.php

$app['router']->get('/', function()
{
    return 'Thanks for buying Code Bright!';
});
```

Here we are accessing the routing layer within the container to create a route that responds to a 'GET' HTTP request. This will be familiar to Laravel 3 users, although the syntax will look a little strange.

As you can see, we are accessing the routing component using an array syntax on our container. This is the magic provided by the `ArrayAccess` interface. If we wanted to, we could also access the routing layer like this:

```php
<?php

// app/routes.php

$app->router->get('/', function()
{
    return 'Seriously, thanks for buying Code Bright!';
});
```

Accessing the component as an attribute of our container works exactly the same as the method we used earlier.

It's not very pretty though, is it? I mean, if you are new to the framework, you have probably heard about Laravel's beautiful syntax. If you used Laravel 3 previously, then you have already been enjoying it.

Surely we aren't going to ruin that reputation? Of course not! Let's have a look at some more magic in action.

Facades

My buddy Kenny Meyers gave the original book 'Code Happy' a lovely review on The Nerdary[12]. I'd love to repay the favor. He told us at the last Laravel conference that he has difficulty with the

[12] http://www.thenerdary.net/

pronunciation of certain words, and I bet this one is tricky. So for Kenny, and those who don't speak English as their first language, here's how you pronounce Facade.

"fah-sahd"

In Laravel 3, most components were accessed using a static method. For example, the routing example in the last chapter would look like this:

```
1  <?php
2
3  // app/routes.php
4
5  Route::get('/', function()
6  {
7      return 'Thanks Kenny, we love you!';
8  });
```

It looks beautiful... we have a descriptive name for the component, and often a verb describing the action performed on that component.

Unfortunately, experienced developers will cringe at the sight of a static method. You see, static methods can be rather difficult to test. I don't want complicate things so early in the book by explaining why that is, so you will have to trust me.

This created a big problem during the design of Laravel 4. We love our beautifully simple syntax, but we also want to embrace coding best practice, and this includes writing many, many, tests to ensure that our code is robust.

Fortunately, Taylor came up with the wonderful idea of Facade classes, named after the 'Facade' design pattern. Using Facades can have the best of both worlds. Pretty static methods, yet instantiated components with public methods. This means that our code is beautiful, and highly testable. Let's see how it works.

In Laravel 4, there are a number of classes that are aliased to the root namespace. These are our Facade classes, and they all extend the `Illuminate\Support\Facades\Facade` class. This class is very clever. I'm going to explain its cleverness with some code samples. You see, we can use a static routing method, just like in good old Laravel 3, like this:

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return 'Thanks for Facades, Taylor!';
});
```

The Facade in this example is the 'Route' class. Each Facade is linked to an instance of a component in the container. The static methods of a Facade are shortcuts, and when called, they call the appropriate public method of the object that they represent within the container. This means that the `Route::get()` method will actually call the following method:

```php
<?php

// app/routes.php

$app['router']->get('/', function()
{
    return 'Thanks for Facades, Taylor!';
});
```

Why is this important? Well, you see, it offers a great deal of flexibility.

Flexibility

What does flexibility mean? I'm not going to refer to the dictionary this time. I know that it means the ability to change. That's a rather good explanation of the benefit of the container. You see, we can change, or rather, replace objects and components that have been registered within the container. This offers us a great deal of power.

Let's imagine for a moment that we don't really like how the routing layer works (don't worry, you will love it, just imagine that you don't).

Since we are all epic coders, we'll go ahead and write our own routing layer, with the primary class being called `SuperRouter`. Due to the flexibility of the container, we can replace the routing layer with our own. It's honestly as easy as you might imagine. We just assign our router to the existing router index. Let's see this in action. Oh, please note that I don't recommend doing this right now, especially if you are a beginner. It just makes for a great example.

```php
<?php

// app/routes.php

$app['router'] = new SuperRouter();
```

Now, not only can we use our router by accessing it directly within the container, but our `Route` Facade will continue to access this component.

In fact, it gets even more impressive. When you replace Laravel's components, the framework will attempt to use your replacement component for its tasks, just as if it was its own.

Allowing access to core framework components in this way offers a great deal of power to you the user. You sure are spoiled aren't you!

Robustness

As I mentioned a few times earlier, Laravel is made from a set of individual components. Each component is responsible for its own little bit of functionality. This means that they are very respectful of the single responsibility principle.

In fact, many of the Laravel components can act alone, completely decoupled from the framework. For this reason, copies of the components can be found under the Illuminate organisation on github[13]. This set of components is also available on Packagist[14] so that you can take advantage of them in your own projects that leverage Composer.

So why is the chapter called Robustness? Well you see, every component used by the Laravel framework is well tested. The entire framework contains a suite of over 900 tests and 1700 assertions.

Thanks to the tests, we can accept contributions and make changes that will provide a fruitful future for the framework, without worrying about whether a change has broken something, or removed existing functionality.

Well, that's enough about architecture. I bet you're ready to get stuck into some development right? Let's get started.

[13] https://github.com/illuminate
[14] https://packagist.org/

Getting Started

Laravel is a framework for the PHP programming language. While PHP is known to have a less than desirable syntax, it is easy to use, easy to deploy, and can be found powering many of the modern web sites you use day to day. Laravel not only provides useful shortcuts, tools, and components to help you achieve success with all of your web based projects, but it also aims to fix some of PHP's flaws.

Laravel has a beautiful, semantic, and creative syntax that allows it to stand out among a large number of frameworks available for the language. This makes PHP a joy to use, without sacrificing power and efficiency. Laravel is a great choice for amateur projects and enterprise solutions alike, and whether you are a seasoned pro with the language, or a newcomer, Code Bright will help you turn the ideas you have into fully functional web applications.

Let's have a quick look at the requirements for both the framework and this book.

Requirements

- ***A Computer*** Reading is great, but you will learn more from playing with the examples that you find in the book.
- ***A Webserver*** Laravel needs a web server. It doesn't matter which one you use but I find the majority of the community use either Apache or Nginx and doing the same will make it much easier to find support if needed.
- ***PHP: Hypertext Preprocessor 5.3 or greater*** Laravel is a PHP framework, it requires the PHP programming language. Trust me you are going to need this one. Since Laravel uses some modern features of the language you will also need version 5.3.7 or greater. You can find out the PHP version used by most web servers by typing `php -v` at the console, or using the `phpinfo()` method.
- ***A Database Server*** While not a requirement of the framework, many of the examples in the book interact with a database. For this reason I would recommend setting up a database server supported by the PDO connector. While I would recommend using the flexible and free `MySQL` by Su.. Oracle, other popular database servers include SQL Server, Postgres and SQLite.
- ***A Text Editor*** You will need this to play with the examples found within the book. I highly recommend Sublime Text 2, and while it's not free, it is extremely sexy. There are however millions of editors and IDEs available, find one that suits the way you work.

Now before we can start working on our Laravel projects, we must first download a copy of the framework.

Installation

I'm not really sure if installation is the right title for this section. However, I couldn't really think of anything better.

You see, Laravel has a Github repository that acts as a kind of 'template' for your new Laravel application. Let's copy that down to our local machine.

We will use git to 'clone' the repository down to a folder on our development web server. Here is the command.

```
git clone git@github.com:laravel/laravel.git my_project
```

Now you will have a template application for Laravel within the `my_project` folder. You might find others referring to this template as the 'app package'. This package will contain your entire application and the entire directory will likely be versioned.

Some experienced Laravel users may remember a directory called `laravel` which used to contain the files that would power the framework. We sometimes would refer to this as the frameworks 'core' files. Well this directory can no longer be found. Leveraging the power of Composer, the core files for the framework now exist as a separate package that is a dependency of our template package.

Let's take a look at the `composer.json` file within our `my_project` directory to see what's new.

```
{
    "require": {
        "laravel/framework": "4.0.*"
    },
    "autoload": {
        "classmap": [
            "app/commands",
            "app/controllers",
            "app/models",
            "app/database/migrations",
            "app/database/seeds",
            "app/tests/TestCase.php"
        ]
    },
    "scripts": {
        "post-update-cmd": "php artisan optimize"
    },
    "minimum-stability": "dev"
}
```

Note that the versions of the dependencies may have changed since this section was last updated. The result however, will remain the same.

Our application package depends only upon the `laravel/framework` package, which contains all of the components needed to power your application. This `composer.json` file is ours. It is for our application and we may edit it however we please. However, some sensible defaults have been provided for you. I also wouldn't recommend removing the `laravel/framework` package. Very bad things will happen.

Right now we only have a template. Let's run `composer install` to install the frameworks core.

```
Loading composer repositories with package information
Installing dependencies
    - Installing doctrine/lexer (dev-master bc0e1f0)
      Cloning bc0e1f0cc285127a38c6c8ea88bc5dba2fd53e94

    - Installing doctrine/annotations (v1.1)
      Loading from cache

    ... Many more packages here. ...

    - Installing ircmaxell/password-compat (1.0.x-dev v1.0.0)
      Cloning v1.0.0

    - Installing swiftmailer/swiftmailer (dev-master e77eb35)
      Cloning e77eb358a1aa7157afb922f33e2afc22f6a7bef2

    - Installing laravel/framework (dev-master 227f5b8)
      Cloning 227f5b85cc2201b6330a8f7ea75f0093a311fe3b

predis/predis suggests installing ext-phpiredis (Allows faster serializatio\
n and deserialization of the Redis protocol)
symfony/translation suggests installing symfony/config (2.2.*)
symfony/translation suggests installing symfony/yaml (2.2.*)
symfony/routing suggests installing symfony/config (2.2.*)
symfony/routing suggests installing symfony/yaml (2.2.*)
symfony/event-dispatcher suggests installing symfony/dependency-injection (\
2.2.*)
symfony/http-kernel suggests installing symfony/class-loader (2.2.*)
symfony/http-kernel suggests installing symfony/config (2.2.*)
symfony/http-kernel suggests installing symfony/dependency-injection (2.2.*\
)
symfony/debug suggests installing symfony/class-loader (~2.1)
monolog/monolog suggests installing mlehner/gelf-php (Allow sending log mes\
```

```
34 sages to a GrayLog2 server)
35 monolog/monolog suggests installing ext-amqp (Allow sending log messages to\
36  an AMQP server (1.0+ required))
37 monolog/monolog suggests installing ext-mongo (Allow sending log messages t\
38 o a MongoDB server)
39 monolog/monolog suggests installing doctrine/couchdb (Allow sending log mes\
40 sages to a CouchDB server)
41 monolog/monolog suggests installing raven/raven (Allow sending log messages\
42  to a Sentry server)
43 Writing lock file
44 Generating autoload files
```

Once again, the package versions may have changed, but the result will still be the same.

Well that sure was a long list of packages! What are they all for? Well, you see, Laravel leverages the power of open source, and the wealth of packages that exist on Composer. These packages are dependencies of the framework itself.

You should definitely take after Laravel and check out the packages that are listed on the Packagist website[15]. There's no point in re-inventing the wheel when it comes to boilerplate code.

Since you have read the Composer chapter, you will now know that the core files for Laravel have been installed to the vendor folder. You don't really want to version your dependencies along the code, so Laravel has provided a sample `.gitignore` file to ignore the `vendors` folder, along with a few other sensible defaults.

We're almost done setting up our Laravel development environment. Only one topic remains. How to set up your webserver.

Web Server Configuration

This section was always going to be a difficult one to write. You see, everyone has a slightly different setup, and there are a number of different web servers available.

So here's what I'm going to do. I will cover the basics of where the web server needs to be pointed. I will also provide some sample configurations for common web servers. They will however be very generic, and require many tweaks to fit into every situation. Still, you can't say I didn't try!

Let's have a look at what the goal is here. Laravel has a directory called `public` which contains its bootstrap code. This code used to launch the framework, and handle all requests to your application.

The `public` folder also contains all of your public assets, such as JavaScript, CSS and images. Essentially, anything that can be accessed via a direct link should exist within the `public` directory.

[15] http://packagist.org

What does this mean for our web server configuration? Well our first task is to make sure that the web server is looking in the right place. We need to edit its configuration so that it will be looking at the `public` directory, and not the root of our project.

The next task will be to let the web server know how to handle pretty URLs.

> I chose a really cool domain name, don't I already have a pretty URL?

Sadly, that's not quite how it works. Life isn't all cakes and pies you know? The Laravel bootstrap exists within a file called `index.php` within the `public` folder. All requests to the framework are going to go through this file. This means that by default, the URL to our guestbook page will look like this:

```
1 http://islifeallcakesandpies.com/index.php/guestbook
```

Our websites users don't really need to know that everything is being directed through `index.php`. It's also not a great feature for search engine optimisation. For that reason our web server should be configured to remove the `index.php` from the URL, leaving only the 'pretty' segment remaining. Normally this is achieved by a piece of config crafted by a regular expression wizard.

Let's now take a look at some sample web server configurations that will allow us to achieve the goals mentioned above. Once again I must remind you that these configurations are to be used as rough guidance. For more detailed information on server configuration I would recommend visiting the documentation for the web server that you have chosen.

Let's start with nginx.

Nginx

Nginx, pronounced 'engine-X', is a wonderful web server that I have recently started using. For me the choice was simple. It performed much faster than apache, and didn't require XML(ish) configuration. It all made sense to me.

On a Debian based linux distribution, like Ubuntu, you can install nginx and PHP by running the following command.

```
1 sudo apt-get install nginx php5-fpm
```

The second package is PHP-FPM, a FastCGI module that will allow nginx to execute PHP code.

On Mac these packages are available on Macports[16]. The required packages can be installed with the following command.

[16] http://www.macports.org/

```
sudo port install php54-fpm nginx
```

Your Nginx site configuration files are normally located at `/etc/nginx/sites-enabled`. Here is a template that you can use to setup a new site.

```
server {

    # Port that the web server will listen on.
    listen          80

    # Host that will serve this project.
    server_name     app.dev

    # Useful logs for debug.
    access_log      /path/to/access.log;
    error_log       /path/to/error.log;
    rewrite_log     on;

    # The location of our projects public directory.
    root            /path/to/our/public;

    # Point index to the Laravel front controller.
    index           index.php;

    location / {

        # URLs to attempt, including pretty ones.
        try_files   $uri $uri/ /index.php?$query_string;

    }

    # Remove trailing slash to please routing system.
    if (!-d $request_filename) {
        rewrite     ^/(.+)/$ /$1 permanent;
    }

    # PHP FPM configuration.
    location ~* \.php$ {
            fastcgi_pass                    unix:/var/run/php5-fpm.sock;
            fastcgi_index                   index.php;
            fastcgi_split_path_info         ^(.+\.php)(.*)$;
            include                         /etc/nginx/fastcgi_params;
```

```
38              fastcgi_param                   SCRIPT_FILENAME $document_root$\
39 fastcgi_script_name;
40     }
41
42     # We don't need .ht files with nginx.
43     location ~ /\.ht {
44             deny all;
45     }
46
47 }
```

Now, not all web servers are going to be the same. This means that providing a generic configuration to suit all situations would be an impossible task. Still, it's enough to get you started.

I've shared the standard configurations used in this chapter on Github so that everyone can contribute to them. You can find them in the daylerees/laravel-website-configs[17] repository.

While nginx is a great choice of web server, the Apache web server is also widely used. Let's take a look at how to configure it.

Apache

The Apache web server can be installed on Debian based systems using the following command.

```
1 sudo apt-get install apache2 php5
```

Here's an Apache VirtualHost configuration that will fit most situations, feel free to contribute to the repository on github[18] if any amendments are needed.

```
1  <VirtualHost *:80>
2
3      # Host that will serve this project.
4      ServerName      app.dev
5
6      # The location of our projects public directory.
7      DocumentRoot    /path/to/our/public
8
9      # Useful logs for debug.
10     CustomLog       /path/to/access.log common
11     ErrorLog        /path/to/error.log
```

[17] http://github.com/laravel-website-configs
[18] http://github.com/laravel-website-configs

```
12
13      # Rewrites for pretty URLs, better not to rely on .htaccess.
14      <Directory /path/to/our/public>
15          <IfModule mod_rewrite.c>
16              Options -MultiViews
17              RewriteEngine On
18              RewriteCond %{REQUEST_FILENAME} !-f
19              RewriteRule ^ index.php [L]
20          </IfModule>
21      </Directory>
22
23  </VirtualHost>
```

Project Structure

I had a chapter in the previous book that covered the directory structure of the package, and where everything likes to live. I think there was a lot of value in this chapter, so I'd like to reiterate it along with the directory structure changes that have happened since Laravel 3.

This section of the getting started will assume that you have already ran `composer install` on a fresh Laravel 4 project.

Project Root

Let's start by taking a look at the root folder structure.

- app/
- bootstrap/
- vendor/
- public/
- .gitattributes
- .gitignore
- artisan
- composer.json
- composer.lock
- phpunit.xml
- server.php

Why don't we have a run through these root items? Sounds like a great idea to me!

app

First up we have the `app` directory. App is used to provide a default home for all of your projects custom code. This includes classes that provide application functionality, config files, and more. The `app` folder is quite important, so rather than giving it a poor summary in a single paragraph, we will cover it in detail at the end of this section. For now, just know this is where your project files live.

bootstrap

- autoload.php
- paths.php
- start.php

The bootstrap directory contains a few files that relate to the startup procedures of the framework. The `autoload.php` file contains most of these procedures, and should only be edited by experienced Laravel users.

The `paths.php` file builds an array of the common filesystem paths that are used by the framework. If for some reason you decide to alter the directory structure of the framework packages, you may need to alter the contents of this file to reflect your changes.

The `start.php` file contains more startup procedures for the framework. I don't want to dig into these in detail right now as that may cause unnecessary confusion. Instead you should probably take note that framework `environments` can be set here. If you don't know what the environments are used for then don't worry. We will cover that later!

Simply put, the contents of the `bootstrap` directory should only be edited by experienced Laravel users who need to alter the shape of the framework on the filesystem. If you are new to Laravel, then just ignore it for now, but don't delete it! Laravel needs this directory to function.

vendor

The `vendor` directory contains all of the composer packages that are used by your application. This, of course, includes the Laravel framework package. For more information about this directory please refer back to the Composer primer chapter.

public

- packages/
- .htaccess
- favicon.ico
- index.php
- robots.txt

The `public` directory should be the only web facing directory of a Laravel application. It's normally where your assets such as CSS, Javascript files and images will live. Let's have a closer look at the contents.

The `packages` directory will be used to contain any assets that need to be installed by third party packages. They are kept in a separate directory so that they don't conflict with our applications own assets.

Laravel 4 ships with a standard `.htaccess` file for the Apache web server. It contains some standard configuration directives that will make sense for most users of the framework. If you use an alternative web server you can either ignore this file or delete it.

By default, web browsers will try to look at the directory index of a site to find a `favicon.ico` file. This is the file that controls the little 16 x 16px image that is displayed in your browsers tabs. The trouble is, when this file doesn't exist the web server likes to complain about it. This causes unnecessary log entries. To counteract this problem Laravel has provided a default, blank `favicon.ico` file, which can be later replaced if so desired.

The `index.php` file is the Laravel framework front controller. It's the first file that the web server hits when a request is received from the browser. This is the file that will launch the framework bootstrap and start the ball rolling so to speak. Don't delete this file, it wouldn't be a good idea!

Laravel has included standard `robots.txt` file that allows all hosts by default. This is for your convenience, feel free to alter this file as necessary.

.gitattributes

Laravel has provided some default Git version control configuration for use out of this box. Git is currently the most popular choice of version control. If you don't intend to version your project with Git then feel free to remove these files.

.gitignore

The `.gitignore` file contains some defaults to inform the Git version control system of which folders should not be versioned. You should note that this also includes both the `vendor` directory and the application `storage` directory. The `vendor` directory has been included to prevent third party packages being versioned.

The `composer.lock` has also been included, although you may wish to remove this entry, and version your lock file to allow for your production environment to install the exact same dependency versions as your development environment. You will find more on this topic within the Composer primer chapter.

artisan

The `artisan` file is an executable which is used to execute the Artisan command line interface for Laravel. Artisan contains a number of useful commands to provide shortcuts or additional functionality to the framework. Its commands will be covered in detail in a later chapter.

composer.json and composer.lock

Both the `composer.json` and `composer.lock` file, contain information about the composer packages used by this project. Once again, you can find more information about these files within the Composer primer chapter.

phpunit.xml

The `phpunit.xml` file provides a default configuration for the PHP Unit testing framework. It will handle the loading of composer dependencies, and execute any tests located in the `app/tests` folder. Information on testing with Laravel will be revealed in a later chapter, stay tuned!

server.php

@todo: This will require more research.

The Application Directory

Here is where your application will take its shape. It is the directory in which you will spend most of your time. For that reason, why don't we get better acquainted with it?

- commands/
- config/
- controllers/
- database/
- lang/
- models/
- start/
- storage/
- tests/
- views/
- filters.php
- routes.php

commands

The commands directory contains any custom artisan command line interface commands that are required by your application. You see the Artisan CLI not only provides default functionality to help you build your project, but you may also create custom commands to do your bidding.

config

The configuration for both the framework and your application are kept within this directory. Laravel's configuration exists as a set of PHP files containing key-value arrays. This directory will also contain sub directories which allow for different configurations to be loaded in different environments.

controllers

As the name suggests, this directory will hold your controllers. Controllers can be used to provide application logic, and to glue the separate parts of your application together. This directory has been added to the default `composer.json` as a classmap autoload location for your convenience.

database

Should you choose to use a database as a method of long term storage, then this directory will be used to hold the files that will create your database schema, and methods for seeding it with sample data. The default SQLite database is also located in this directory.

lang

The `lang` directory contains PHP files with arrays of strings that can be used to provide localisation support to your application. Sub folders named by region allow for string files to exist for multiple languages.

models

The models directory will contain your models. Surprised? Models are used to represent your business model, or provide interaction with storage. Confused? Don't worry. We will cover models in detail in a later chapter. Know that a `User` model has been provided for you to enable application authentication 'out of the box'. Like the `controllers` directory, this has been added to the classmap autoload section of the default `composer.json`.

start

Where the `bootstrap` directory contains the startup procedures that belong to the framework, the `start` directory contains startup procedures that belong to your application. As always, some sensible defaults have been provided for you.

storage

When Laravel needs to write anything to disk, it does so within the `storage` directory. For this reason your web server must be able to write to this location.

tests

The `tests` directory will contain all of the unit and acceptance tests for your application. The default PHP Unit configuration that has been included with Laravel, will look for tests within this directory by default.

views

The views directory is used to contain the visual templates for your application. A default `hello` view has been provided for your convenience.

filters.php

The `filters.php` file is used to contain the route filters for your application. You will learn more about filters in a future chapter.

routes.php

The routes file contains all of the routes for your application. You don't know what routes are? Well, let's not waste any more time then. Onwards to the next chapter!

Basic Routing

You are on a long, straight mud path out in the wilderness of… What country has a wilderness? Australia! That will do.

Don't freak out! This section requires a little imagination, so work with me on this one. I suppose our main character should have a name because books seem to do that. Hmm… let's call him Browsy Mc'Request.

Browsy Mc'Request is walking along a straight, muddy, path somewhere in kangaroo country. Browsy isn't a native to this land and has found himself lost with no idea where he is heading. He does, however know where he wants to be. A friend had earlier told him about "Jason Lewis' Shelter for Moustached Marsupials". He knows that Movember is fast approaching and the Kangaroo stylists will need all the help they can get.

All of a sudden, Browsy spots what looks like a fork in the road way off in the distance. Excited at seeing anything other than his old familiar straight, muddy path, he sprints towards the fork hoping he might find some kind of direction.

Upon reaching the fork, he searches high and low for any indication of which path might lead to the shelter, but finds nothing.

Fortunately for Browsy, at that exact moment, a wild web developer appears. The strong and handsome web developer slams a signpost into the tough, dry dirt with ease, and dashes away in the blink of an eye.

Browsy is left dumbfounded by both the speed and rugged handsomeness of the web developer. Did I mention he was handsome? The web developer… not Browsy. Browsy looks kind of like THAT GUY from THAT BOOK who keeps saying 'my precious' a lot. At least that's how I see it.

After catching his breath, Browsy begins to study the signpost.

To the left you will find "Bob's Bobcat Riding Stables", and to the right… ah! Here we go. To the right you will find "Jason Lewis' Shelter for Moustached Marsupials".

Browsy eventually found his way to the Marsupial Shelter and lived a long and happy life learning to groom the many types of moustaches that kangaroos are able to grow.

You see, the moral of the story is that without a (HANDSOME) developer showing him the way, Browsy Mc'Request wouldn't have found his way to our applic… to the Marsupial Shelter.

Just like in the story, a web browser request wouldn't be directed to your application logic without some form of routing.

Basic Routing

Let's take a look at a request being made to the Laravel framework.

```
http://domain.com/my/page
```

In this example, we are using the `http` protocol (used by most web browsers) to access your Laravel application hosted on `domain.com`. 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/routes.php` file, so let's go ahead and create a route that will listen for the request we mentioned above.

```
<?php

// app/routes.php

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

Now enter `http://domain.com/my/page` into your web browser, replacing `domain.com` with the address to your Laravel application. This is likely `localhost`.

If everything has been configured correctly, you will now see the words 'Hello world!' in wonderful Times New Roman! Why don't we have 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, 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 gets 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:

```php
<?php

// app/routes.php

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

All of these methods 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 situation you are using it in to make the application more transparent.

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

```php
<?php

// app/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:

```php
<?php

// app/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:

```php
<?php

// app/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.

```
http://domain.com/first/page
http://domain.com/second/page
http://domain.com/third/page
```

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

```
http://domain.com
```

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

```
<?php

// app/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?!

CALM YOURSELF READER! You are becoming frantic.

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.

```
http://domain.com
http://domain.com/
```

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:

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

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

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

```php
<?php

// app/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 minimise 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...

```php
<?php

// app/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:

```
http://domain.com/books/crime
```

You would be greeted with this text response:

```
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:

```
<?php

// app/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:

```php
<?php

// app/routes.php

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

Now if we visit the following the following URL:

```
http://domain.com/books
```

We will receive this response:

```
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.

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 give them at least a nod in return...some form of response.

Requests made to a web server are no different than 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:

```
<?php

// app/routes.php

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

So here we have the string "Yeh am alright guv." sent back to the browser. The string is our response and is always `returned` 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?

```
<?php

// app/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 a number of `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 `app/views` directory. This means that `PHP` code can also be parsed within your views. Let's just create a very simple view for now.

```
<!-- app/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 `app/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`.

```php
<?php

// app/routes.php

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

> Oh, I see! You were on 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 `app/views/simple.php`. This is because Laravel is clever. It will by default assume that your views are located in `app/views` and will look for a file with an appropriate view extension.

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 your 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 we can 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.

```php
<?php

// app/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!

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.

```
<?php
array('name' => 'Taylor Otwell', 'status' => 'Code Guru')
```

In the resulting view template we are able to access these values like this:

```
<?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.

```
<!-- app/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` reponse 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.

```
<?php

// app/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.

```
<?php

// app/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 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 really 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.

```
<?php

// app/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 filters, 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` inherit from the 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 generally 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.

```php
<?php

// app/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.

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 are able to define custom headers as we please. Let's have a look at how we can set some header values.

```php
<?php

// app/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.

```php
<?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. Hrrm... let's instead 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 can.

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.

```php
<?php

// app/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, in this case ***some bold text***, 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 by receiving the `text/x-markdown` value it will know to use a markdown transformer to handle the body.

Now 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 actually 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 actually inherits most of its content from the `Response` object belonging to the `Symfony HTTPFoundation` 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[19]. 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?

---

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

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

Parameters:

integer $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 kind of cruel James Bond villain.

[19] http://api.symfony.com/2.2/Symfony/Component/HttpFoundation/Response.html"TheSymfonyResponseObject"

```php
<?php

// app/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.

Response Shortcuts

Laravel is your friend. It's true... no actually it's not true. Laravel is more than a friend. It loves you. It really does. Because of this, it has provided a number of response shortcuts to make your life easier. Let's have a look at what's available to us.

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 a number of details that are specific to JSON results. For example an appropriate HTTP Content-Type header.

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.

```php
<?php

// app/routes.php

Route::get('markdown/response', function()
{
    $data = array('iron', 'man', 'rocks');
    return Response::json($data);
});
```

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 infact a JSON string. The web browser will receive the following body:

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

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

Download Responses

Serving 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.

```php
<?php

// app/routes.php

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

Now if we navigate to the `/file/download` URI the browser will intiate a download instead of displaying a response. The `Response::download()` method received 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 HTTP status code and an array of headers. For example:

```php
<?php

// app/routes.php

Route::get('file/download', function()
{
    $file = 'path_to_my_file.pdf';
    return Response::download($file, 418, array('iron', 'man'));
});
```

Here we will serve our file with the HTTP status code of 418 (I'm a Teapot) and a header value of `iron=man`.

Well 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.

In the next chapter we will take a look at route filters, which will allow us to protect our routes or perform actions before/after they are executed.

Filters

I recall a couple of years back when Jesse O'brien was planning a private event where he and his buddies could watch the local hockey team play their latest match against the Laravel Pandas.

Now we all know that the mighty Laravel Pandas could never be beaten by the London Knights, but Jesse refused to listen. Insisting that this could be the start of the road to glory for the Knights.

The event was planned to take place at the Hoser Hut in central London. A friendly welcoming place for anyone born in 'very-north' America. (Maple syrup land.)

Unfortunately the Hoser Hut had a reputation for being not so welcoming to those who visit from over the border. It was a known fact that Americans were thrown out of the windows of the Hoser Hut on a regular basis. It was for that reason that Jesse decided he needed some kind of door filter to keep the nasty Americans out. Of course the good ol' british chap Dayle Rees was always welcome at the Hoser Hut. He was welcome anywhere.

Jesse employed a bouncer to stand at the front of the Hoser Hut and ask to see ID to confirm whether the guest visiting the hut was Canadian or not.

You see, what Jesse did was implement a filter. Those passing the requirements of the filter would be granted entrance to the warm and cozy Hoser Hut to watch the Laravel Pandas destroy the London Knights. However Americans attempting to enter the bar would not meet the criteria of the filter, and would be shown the shiny side of a boot.

Let's leave Jesse to his game and see how we can use filters to protect our application routes.

Basic Filters

Filters are certain sets of rules or actions that can be applied to a route. They can be performed before or after a route's logic is executed, however, you will find before filters to be more useful. Using before filters we can alter the flow of the application if a certain set of rules or criteria are not met. It's a great way of protecting our routes.

As always, an example speaks a thousand words. Let's have a look at a filter, but first we need something else. Let's see:

```
1  <!-- app/views/birthday.php -->
2
3  <h1>Happy Birthday!</h1>
4  <p>Happy birthday to Dayle, hurray!</p>
```

Great! Now that we have a happy birthday view, we can create our first filter. Here we go:

```php
<?php

// app/filters.php

Route::filter('birthday', function()
{
    if (date('d/m/y') == '12/12/84') {
        return View::make('birthday');
    }
});
```

Here we have our first filter. Laravel has provided a file at `app/filters.php` as a generic location for our filters, but we can actually put them wherever we like.

We use the `Route::filter()` method to create a new filter. The first parameter is a friendly name that we will later use to assign the filter to a route. In this example I have named the filter 'birthday'. The second parameter to the route is a callback, which in the example is a Closure.

The callback is a function that is called when the filter is executed. If it returns a response type object, just like those that we use within our route logic, then that response will be returned and will be served instead of the result of the route logic. If no response is returned from the filter callback, then the routes logic will continue as normal.

This gives us a great deal of power, so go ahead and practice your evil mastermind laugh. Seriously, this is important business.

> Muahahahah!

Well, I suppose that will have to do. You see we can either alter the flow of the application, or perform an action and allow the route logic to continue its execution. For example, we might want to only show a certain type of content on our website, to a certain type of user. This would mean returning a redirect response to another page. Alternatively, we could write a log each time the filter is executed, to see which pages have been visited. Perhaps I am getting ahead of myself, let's have another look at our example filter.

```php
<?php

// app/filters.php

Route::filter('birthday', function()
{
    if (date('d/m') == '12/12') {
        return View::make('birthday');
    }
});
```

Looking closely at our closure, we can see that we have a condition, and a response. In our filter, if the current date is equal to '12/12/84', which is of course the date that the most important person in the universe was born, then the closure will return the response. If the response is returned from the Closure, then we will be redirected to the happy birthday view. Otherwise our route logic will continue as normal.

Of course, before the filter will become useful we need to attach it to a route. However, before we can do that we need to change the route's structure a little. Do you remember how I told you that routing methods accept a closure as the second parameter? Well I told a little white lie again. Sorry.

You see the route methods can also accept an array as the second parameter. We can use this array to assign additional parameters to the route. Let's have a look at how a route looks with an array as a second parameter.

```php
<?php

// app/routes.php

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

You see, it's pretty similar. What we have done is shifted the Closure into the array. It functions just as it did before. In fact, as long as we keep the closure in the array, we can include other values. That's how we are going to attach our filter. Let's start by taking a look at the 'before' filter option.

```php
<?php

// app/routes.php

Route::get('/', array(
    'before' => 'birthday',
    function()
    {
            return View::make('hello');
    }
));
```

As you can see, we have created another option within our array. The index 'before' tells the framework that we want to run our 'birthday' filter before the routes logic is executed. The value 'birthday' matches up with the nickname that we gave to our filter.

Let's go ahead and execute our route by visiting `/`. Now, assuming today isn't the 12th of December then you will see the Laravel welcome page. This is because the filter conditional logic failed, and no response was returned.

Right, so let's wait until the 12th of December so that we can see what happens when the filters condition passes and the response is returned.

Just kidding, let's change the filter to force it to pass. We can change the condition to the boolean value `true`.

```php
<?php

// app/filters.php

Route::filter('birthday', function()
{
    if (true) {
        return View::make('birthday');
    }
});
```

There we go, now let's visit `/` to see if anything has changed. Hurray, it's my birthday! Let's all sing happy birthday to me. Actually, let's just wait until December. So we can see that the birthday filters logic has succeeded, and the happy birthday view has been returned.

We can attach a filter using the 'after' option of a route array, this way the filter will be executed after your route logic. Here's an example:

```php
<?php

// app/routes.php

Route::get('/', array(
    'after' => 'birthday',
    function()
    {
       return View::make('hello');
    }
));
```

You need to remember, however, that the after filter cannot replace the response. Thus, our birthday filter is a little pointless when using 'after'. You could however perform some logging, or a cleanup operation. Just remember that it's there if you need it!

Multiple Filters

One other thing you should know is that you can apply as many filters as you like to a route. Let's have a look at some examples of this in action. First, let's attach multiple before filters:

```php
<?php

// app/routes.php

Route::get('/', array(
    'before' => 'birthday|christmas',
    function()
    {
       return View::make('hello');
    }
));
```

Here we have attached both the 'birthday' and 'christmas' before filters to the route. I'll let your imagination decide what the 'christmas' filter does, but make sure it's something magical.

The pipe `|` character can be used to separate a list of filters. They will be executed from left to right, and the first that returns a response will end the request, and that response will be delivered as the result.

If you like, you can use an array instead to provide your multiple filters. this might strike you as being more 'phpish'.

```php
<?php

// app/routes.php

Route::get('/', array(
    'before' => array('birthday', 'christmas'),
    function()
    {
       return View::make('hello');
    }
));
```

Use whichever suits the way you code, personally I like the arrays. If you want, you can also assign a 'before' and 'after' filter at the same time, like this:

```php
<?php

// app/routes.php

Route::get('/', array(
    'before'    => 'birthday',
    'after'     => 'christmas',
    function()
    {
       return View::make('hello');
    }
));
```

Naturally, the before filter will run first, then the route logic, and finally the after filter.

So, you think you're done with filters? Don't get carried away!

Filter Parameters

Just like PHP functions, filters can accept parameters. This is a great way to avoid repetition, and allow for increased flexibility. Let's lead with an example, as always.

```php
<?php

// app/filters.php

// before

Route::filter('test', function($route, $request)
{

});

// after

Route::filter('test', function($route, $request, $response)
{

});
```

> Wait, why are there two filters?

Well spotted! Well, actually they are the same filter, but still, your question is valid. You see, Laravel provides a different set of parameters to 'before' and 'after' filters. You will notice that both filters receive `$route` and `$request` variables. You can actually call them whatever you like, but I named them this way for a reason.

If you were to `var_dump()` the first parameter, you will see that it is an instance of `Illuminate\Routing\Route`. You will remember that 'Illuminate' is the codename used for Laravel 4 components. The 'Route' class represents a route used by the routing layer. This instance represents the current route that is being executed. Clever right? The 'Route' instance is gigantic, go ahead and `var_dump()` it if you don't believe this shifty welshman. You could interrogate it for the detailed information contained within, or even alter some of its values to manipulate the framework. However, this is an advanced topic, and outside the scope of this chapter, so let's look at the next parameter instead.

As you might have guessed, the next parameter is an instance of the current request object. The `Illuminate\Http\Request` instance represents the state of the request being sent to your web server. It contains information about the URL, and data passed with the request, along with a great wealth of additional information.

The after filter receives an additional parameter, an instance of response object that is returned from the route the filter is acting on. This instance is due to be served as the response of the current request.

Right, those parameters Laravel gave us might be useful to advanced users of the framework, but wouldn't it be great if we could provide our own parameters to our route filters? Let's take a look at how we can do that.

First we need to add a placeholder variable to our filter Closure, it should come after the ones that laravel provides, like this:

```
<?php

// app/filters.php

Route::filter('birthday', function($route, $request, $date)
{
    if (date('d/m') == $date) {
        return View::make('birthday');
    }
});
```

Our birthday filter has been altered to accept a `$date` parameter. If the current date matches the date provided then the birthday filter is executed.

Now all we need to know is how to provide the parameters to our route filter. Let's take a look.

```
<?php

// app/routes.php

Route::get('/', array(
    'before'    => 'birthday:12/12',
    function()
    {
       return View::make('hello');
    }
));
```

The parameter we pass to our filter comes after the colon : character when we assign it to the route. Go ahead and test it, change the date to the current day and watch the filter fire.

If we want to provide additional parameters, then we need to provide extra placeholder variables with the Closure. It will look something like this.

```php
<?php

// app/filters.php

Route::filter('birthday', function($route, $request, $first, $second, $thir\
d)
{
    return "{$first} - {$second} - {$third}";
});
```

We can accept as many parameters as we like. To provide multiple parameters we must first add a colon : between the filter name and its parameters. The parameters themselves must be separated by a comma ,. Here's an example:

```php
<?php

// app/routes.php

Route::get('/', array(
    'before'    => 'birthday:foo,bar,baz',
    function()
    {
       return View::make('hello');
    }
));
```

The values 'foo', 'bar', and 'baz' will be passed to the placeholders we added to the filter. It's worth noting that just like functions, filter parameters can be assigned default values, making them optional. Here's an example:

```php
<?php

// app/filter.php

Route::filter('example', function($route, $request, $optional = 'Yep!')
{
    return $optional;
});
```

Provide the optional parameter or don't. It's up to you, it's your framework!

Feel free to use as many parameters as you like to make your filter more efficient. Take advantage of this great feature.

Filter Classes

Closures are great. They are really convenient, and work great in my examples. However, they are strapped to the logic that we are writing. We can't instantiate them, this makes them difficult to test.

For that reason, any Laravel feature that requires a Closure will also have an alternative. A PHP Class. Let's have a look at how we can use a class to represent our filters.

Before we make the class, we need somewhere to put it. Let's create a new folder in `/app` called `filters`, and update our `composer.json` classmap to include the new folder.

```
"autoload": {
    "classmap": [
        "app/commands",
        "app/controllers",
        "app/models",
        "app/filters",
        "app/database/migrations",
        "app/database/seeds",
        "app/tests/TestCase.php"
    ]
}
```

Now let's create a new class for our birthday filter. Here we go:

```
<?php

// app/filters/Birthday.php

class BirthdayFilter
{
    public function filter($route, $request, $date)
    {
        if (date('d/m') == $date) {
            return View::make('birthday');
        }
    }
}
```

I have called my class 'BirthdayFilter', you don't need the 'Filter' suffix, but I like to do it anyway, it's up to you. What you do need however, is the `filter()` method. It works just like the Closure. In fact, because it works just like the Closure I don't need to explain it again. Instead, let's take a look at how we can hook up a filter to a route.

First we need to create a filter alias, once again we will use the `Route::filter()` method. However, this time we will pass a string instead of a closure as the second parameter. Like this:

```
<?php

// app/routes.php

Route::filter('birthday', 'BirthdayFilter');
```

The second parameter for the method is a string that identifies the filter class to use. If the filter class is located within a namespace, go ahead and supply the namespace too.

Now that the filter alias has been created, we can add it to the route just as we did before.

```
<?php

// app/routes.php

Route::get('/', array(
    'before'    => 'birthday',
    function()
    {
       return View::make('hello');
    }
));
```

Remember that you will need to run `composer dump-autoload` before Composer, and Laravel, will be able to find our filter class.

If you intend to fully test your code, then writing filters as classes is the best way to go about your business. We'll discover more about testing in a later chapter.

Global Filters

If you take a look inside `/app/filters.php` you will notice two strange looking filters. These are the global filters and are executed before, and after, every request to your application.

```php
<?php

// app/filters.php

App::before(function($request)
{
        //
});


App::after(function($request, $response)
{
        //
});
```

They work exactly like normal filters, except that they apply to all routes by default. This means there is no need to add them to the `before` and `after` array indexes of our routes.

Default Filters

In the `app/filters.php` there are some filters that have already been created for you. Let's have a look at the first three.

```php
<?php

// app/filters.php

Route::filter('auth', function()
{
        if (Auth::guest()) return Redirect::guest('login');
});


Route::filter('auth.basic', function()
{
        return Auth::basic();
});

Route::filter('guest', function()
{
    if (Auth::check()) return Redirect::to('/');
});
```

All of these filters relate to the authentication layer of Laravel. The can be used to restrict route access to users who are, or are not, logged in to the web application at present.

In a later chapter we will be taking a closer look at the authentication layer, and the content of these filters will make more sense. For now, just know that they are there waiting for you!

The fourth filter is the cross site request forgery filter, and looks like this:

```
<?php

// app/filters.php

Route::filter('csrf', function()
{
        if (Session::token() != Input::get('_token'))
        {
                throw new Illuminate\Session\TokenMismatchException;
        }
});
```

You can attach it to your routes to protect requests being posted from an origin other than your own application. This is a very useful security measure that is used primarily to protect routes that are the target of forms, or data submission.

Feel free to take advantage of the filters Laravel has provided, they have been put there to save you time.

Pattern Filters

Don't want to attach the filter manually to all your routes? No I can't blame you. Tired fingers happen, I'm writing a book, I know this. Let's try to find a way to save your poor little phalanges some effort. Here's a pattern filter.

The pattern filter will allow you to match a before filter to a number of routes by supplying a routing pattern with a wildcard. Let's see this in action.

```
<?php

// app/routes.php

Route::when('profile/*', 'birthday');
```

The `Route::when()` method above, will run the 'birthday' filter on all route URIs that start with 'profile/'. The star within the first parameter will act as a wildcard. This is a great way of attaching a before filter to a number of different routes at once.

Controllers

Creating Controllers

In the basic routing chapter we were taught how to link routes to closures, 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 housing application logic is the Controller.

The Controller is a class used to house routing logic. Normally, the Controller will contain a number of 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. So let's dive right in and look at a controller. Here's one I made earlier! Cue Blue Peter theme. Actually 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, controllers..

```
1  <?php
2
3  // app/controllers/ArticleController.php
4
5  class ArticleController extends BaseController
6  {
7      public function showIndex()
8      {
9          return View::make('index');
10     }
11
12     public function showSingle($articleId)
13     {
14         return View::make('single');
15     }
16 }
```

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 either `BaseController` or `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/controllers` directory which Laravel has created for us. This directory is being 'classmap file loaded' from our composer.json by default. If `app/controllers` doesn't suit your workflow, then you can put the file wherever you like. However, you must make sure that the class can be autoloaded by Composer. For more information on this topic, please refer back to the Composer primer.

The class methods of our Controller are what actually contain our logic. Once again you can name them however you like, but personally I like to use the word 'show' as a prefix if the result is that they display a web page. 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.

```
1  public function showIndex()
2  {
3      return View::make('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.

```
1  Route::get('index', function()
2  {
3      return View::make('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!

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 it's 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 actually 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.

```
<?php

// app/routes.php

Route::get('index', 'ArticleController@showIndex');
```

In order to link a URI to a Controller we must define a new route within the `/app/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.

```
<?php

// app/controllers/ArticleController.php

class ArticleController extends BaseController
{
    public function showIndex()
    {
        return View::make('index');
    }

    public function showSingle($articleId)
    {
        return View::make('single');
    }
}
```

So we can see from the example that the class name is `ArticleController`, and the action we wish to route to is called `showIndex`. Let's put them together, with an at (`@`) in the middle.

```
ArticleController@showIndex
```

It really is as simple as that. Now we can 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.

```php
<?php

// app/routes.php

Route::post('article/new', 'ArticleController@newArticle');
```

You guys are pretty smart chaps, 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`.

Here's a neat thing to know. You can namespace your controller and Laravel won't bat an eyelid. Just make sure to include the namespace within your route declaration and everything will be just dandy! Let's see this in action.

```php
<?php

// app/controllers/Article.php

namespace Blog\Controller;

use View;
use BaseController;

class Article extends BaseController
{
    public function showIndex()
    {
        return View::make('index');
    }
}
```

So here we have a similar Controller to that used in the other example. This time however, it is contained within the `Blog\Controller` namespace. Since it's located within the `Controller` section of the namespace, I have omitted the `Controller` suffix from the class name. This is my own personal preference, I leave it up to you to decide whether you keep it or not.

Let's see how this namespaced controller can be routed to. You've probably already guessed it!

```php
<?php

// app/routes.php

Route::post('index', 'Blog\Controller\Article@showIndex');
```

Just as before, only this time the name of the controller has been prefixed with its namespace. See, namespaces don't have to complicate things! You can even store your namespaced Controller in a nested directory, or elsewhere in a PSR-0 loading scheme. Laravel doesn't care, as long as Composer knows where to find your class, then Laravel will be able to use it.

RESTful Controllers

Laravel offers solutions, we know this much. It also provides options, RESTful Controllers are a prime example of this.

We know that we can define the HTTP request verb that we want to match using the routing methods. This is really convenient when routing to Closures. However, when routing to Controllers you might want to keep the defining of the request verb close to your application logic. Well the good news is that Laravel provides this alternative configuration.

Let's alter our Controller a little shall we?

```php
<?php

// app/controllers/Article.php

namespace Blog\Controller;

use View;
use BaseController;

class Article extends BaseController
{
    public function getCreate()
    {
        return View::make('create');
    }

    public function postCreate()
    {
        // Handle the creation form.
```

```
20      }
21  }
```

Here we have our Article controller once more. The intention of the actions are to provide a form to create and handle the creation of a new blog article. You'll notice that the names of the actions have been prefixed with `get` and `post`. Those are our HTTP request verbs.

So, in the above example you might think that we have represented end points for the following URLs:

```
1  GET  /create
2  POST /create
```

You might also be wondering how we route to our RESTful controller. You see, using the verb methods on the `Route` class wouldn't make much sense here. Well, say goodbye to routing to individual actions. Let's take a look at another routing method.

```
1  <?php
2
3  // app/routes.php
4
5  Route::controller('article', 'Blog\Controller\Article');
```

This single method will route all of the actions represented by our RESTful controller. Let's take a closer look at the method signature.

The first parameter is the base URL. Normally RESTful routing is used to represent an object, so in most circumstances the base URL will be the name of that object. You can think of it as some what of a prefix to the actions that we have created within our RESTful controller.

The second parameter you will already be familiar with. It's the controller that we intend to route to. Once again, Laravel will happily accept a namespaced controller as a routing target so feel free to organise your Controllers however suits your needs.

As you can see, using this method to route to your controllers provides a distinct advantage over the original routing method. With this method, you will only have to provide a single routing entry for the Controller, rather than routing to every action independently.

Blade

In this chapter, we will learn how to master the Blade. You will need it. In order 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 Phil Sturgeon's 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 honour of Laravel 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'? Well 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.

Creating Templates

I know, I know, I already taught you how to create views right? They are really useful for separating the visual aspect of your application from its logic, but, it doesn't mean 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 out of place within our neat HTML templates. They soil 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.

```
<!-- app/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, looks kinda 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 show 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::make()` method, but that would be incorrect.

```
<?php

// app/routes.php

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

See? Laravel knows what a Blade template is, and how to look for one. For this reason the `View::make()` 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.

PHP Output

A lot of your templates will involve echoing out data provided by the logic portion of your application. Normally it would look something like this:

```
<!-- app/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.

```
<!-- app/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.

```
<!-- app/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.

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

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

What a nasty piece of Code! It would 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.

We have the power to protect our templates from the output though. If we were to use {{{ three curly brackets }}} instead of {{ two }} then our output will be escaped, the angular brackets will be converted to HTML entities, and the Javascript will show as text within the page. Harmless!

Let's see this in action.

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

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

All we have changed is the number of brackets around our nasty Javascript snippet. Let's view the source of the page to see how it looks.

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

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

As you can see, the HTML tags and some other characters have been replaced with the equivalent HTML entities. Our website is saved!

Moving on!

Control Structures

PHP has a number of control structures. If statements, while, for and foreach loops. If you haven't heard of these before then this book really isn't the one for you!

Within templates you will most likely be familiar with using the alternative syntax for control structures using colons `:`. Your `if` statements will look like this:

```
<!-- app/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 looks within a Blade template.

```
<!-- app/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 have 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.

```
<!-- app/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?

```
<!-- app/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.

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

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

Enjoy your cookie! As you can see, we 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.

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

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

Simple, and exactly what 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.

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

@while (isPretty($kieraKnightly))
    <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 dunno if PHP ha..

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

```
<!-- app/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.

Templates

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 5?

- php artisan project:complete
- Allow view composers to construct hair styles.
- Have views write themselves.

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

With PHP, you are able to `include()` a file into the current file, executing its contents. You could do that with views to break them apart into separate files for organisational 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 have a look at an example of this in action.

Here's `header.blade.php` file containing the header for our page, and possibly even other pages.

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

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

Here's the associated footer template.

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

<small>Information provided based on research as of 3rd May '13.</small>
```

Now here's our primary template. The one that is being displayed by our routed Closure or Controller action.

```
<!-- app/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::make()` method that we used earlier. Let's have a look at the resulting document.

```
<!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 makes 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. There are some parts of a web page that don't really 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 if you like. Here's an example:

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

We're gonna 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:

```
<!-- app/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 look at the one within the body first, that's the easy one. It looks like this:

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

This statement tells blade to create a section here that we can fill in content for later. We give it the nickname 'body' so we can refer back to it later.

The other section looks like this:

```
@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.

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

```
<!-- app/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:

```
@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::make()`, so in this situation we are referring to the 'base.blade.php' file in the 'layouts' directory within 'app/views'. Remember that a period (.) character represents a directory separator when dealing with views.

In Laravel 3 this function was called '@layout()', but it has been renamed 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:

```
@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::make()` response to render the child template. Like this:

```
<?php

// app/routes.php

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

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

```
<!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.

So you see, we could have as many child templates as we want inheriting 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:

```
<!-- app/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 can imagine, the head section has been injected with our additional CSS file, and the source for our page now looks like this:

```
<!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:

```
<!-- app/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 actually quite simple. Let's have a look at how the source has changed for some clarity.

```
<!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.

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

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

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

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

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

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

<!-- app/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.

```
<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 probably know already, HTML has its own method of including comments. They look like this.

```
<!-- 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 really 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:

```
<?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.

```
{{-- 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.

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.

In the Filters chapter you learned about how we can give an array as the second parameter to the routing methods to allow for more information to be included with our route definition. Like this:

```
<?php

// app/routes.php

Route::get('/', array(
    'before' => 'sexyfilter',
    function() {
        return View::make('hello');
    }
));
```

In this example, we are using the array syntax to include information about what filters we want to apply to the route. It doesn't end there though, you can do a lot more with this array. Let's take a look at what we have available.

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 really 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:

```php
<?php

// app/routes.php

Route::get('/my/long/calendar/route', array(
    '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 a number of 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.

```
// app/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 a number of 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:

```
<?php

return new 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:

```
<?php

// app/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.

```
<?php

// app/routes.php

Route::get('/my/long/calendar/route', array(
    'as' => 'calendar',
    'uses' => 'CalendarController@showCalendar'
));
```

Easy right? Now let's look at how we can make our routes more secure.

Secure Routes

You may want your routes to respond to secure HTTP URLs so that they can handle confidential data. HTTPS URLs are layered on top of the SSL or TLS protocol to allow for increased security when you need it. Here's how you can allow your routes to match this protocol.

```php
<?php

// app/routes.php

Route::get('secret/content', array(
    'https',
    function () {
        return 'Secret squirrel!';
    }
));
```

By adding the HTTPS index to our routing array, our route will now respond to requests made to the route using the HTTPS protocol.

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:

```php
<?php

// app/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 really 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.

```php
<?php

// app/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.

Now 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:

```php
<?php

// app/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

Remember how we were able to provide conditions to our routes in the Filters chapter? That was really handy right? It would be a shame to have to attach the same filter to many route definitions though.

Wouldn't it be great if we could encapsulate our routes, and apply a filter to the container? Well, you might have guessed already, but here's an example that does exactly that.

```php
<?php

// app/routes.php

Route::group(array('before' => 'onlybrogrammers'), function()
{

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

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

    // Third Route
    Route::get('/third', function() {
        return 'Come at me bro.';
    });

});
```

In the above example we are using the `group()` method on the 'Route' object. The first parameter is an array. It works just like the ones we have been using within our routing methods. It can accept filters, secure indexes, and many of the other routing filters. The second parameter should be a Closure.

When you define additional routes within this Closure, the routes inherit the properties of the group. The three routes within the group above are all protected by the 'onlybrogrammers' before filter.

Now, we can use the routing array filters we discovered earlier in the chapter on the groups, but we can also use some new features that are specific to route groups. Let's take a look at these new features.

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.

```php
<?php

// app/routes.php

Route::group(array('prefix' => 'books'), function()
{

    // First Route
    Route::get('/first', function() {
        return 'The Colour of Magic';
    });

    // Second Route
    Route::get('/second', function() {
        return 'Reaper Man';
    });

    // Third Route
    Route::get('/third', function() {
        return 'Lords and Ladies';
    });

});
```

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.

```
/books/first

/books/second

/books/third
```

Use route prefixes to avoid repetition within your routes, and to group them by purpose for organisational 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.

```
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 discover how we can use domain routing to serve different content from different domains.

Here's an example of domain based routing:

```
<?php

// app/routes.php

Route::group(array('domain' => 'myapp.dev'), function()
{
    Route::get('my/route', function() {
        return 'Hello from myapp.dev!';
    });
});

Route::group(array('domain' => 'another.myapp.dev'), function()
{
    Route::get('my/route', function() {
        return 'Hello from another.myapp.dev!';
    });
});

Route::group(array('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 are able to provide a host name, that *must* match the current hostname for any of the routes inside to be executed.

The host name can either be a subdomain, 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.

```php
<?php

// app/routes.php

Route::group(array('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:

```
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.

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!

In order 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 a number of 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()` helper. Let's create a simple route to test it.

```
<?php

// app/routes.php

Route::get('/current/url', function()
{
    return URL::current();
});
```

Now if we visit our `/current/url` url, we receive the following response.

```
http://myapp.dev/current/url
```

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.

```
http://myapp.dev/current/url?foo=bar
```

You will see that the result of `URL::current()` strips off the extra request data, like this:

```
http://myapp.dev/current/url
```

Well the `URL::full()` method is a little different. Let's modify our existing route to use it. Like this:

```
<?php

// app/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:

```
http://myapp.dev/current/url?foo=bar
```

You see, the `URL::full()` method also includes additional request data.

This next one isn't really a way of getting the current URL, but I feel that it certainly has its place in this sub heading. 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.

```
<?php

// app/routes.php

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

Route::get('second', function()
{
    return URL::previous();
});
```

So 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:

```
http://demo.dev/first
```

You see, after the redirect, the `URL::previous` method gives the URL for the previous request, which in this instance is the URL to the first route. It's as simple as that!

Generating Framework 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:

```
<?php

// app/routes.php

Route::get('example', function()
{
    return URL::to('another/route');
});
```

The response we receive when we visit `/example` looks like this.

```
http://demo.dev/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.

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:

```php
<?php

// app/routes.php

Route::get('example', function()
{
    return URL::to('another/route', array('foo', 'bar'));
});
```

The resulting string will take the following form.

```
http://myapp.dev/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:

```php
URL::to('another/route', array('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:

```php
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:

```php
URL::secure('another/route', array('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:

```php
<?php

// app/routes.php

Route::get('the/best/avenger', array('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.

```
http://myapp.dev/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.

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.

```php
<?php

// app/routes.php

Route::get('the/{first}/avenger/{second}', array(
    'as' => 'ironman',
    function($first, $second) {
        return "Tony Stark, the {$first} avenger {$second}.";
    }
));

Route::get('example', function()
{
    return URL::route('ironman', array('best', 'ever'));
});
```

If we visit the following URL...

```
http://myapp.dev/example
```

...Laravel will fill in the blanks in the correct order, with the parameters we have provided. The following URL is displayed a response.

```
http://myapp.dev/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.

```
<?php

// app/routes.php

// Our Controller.
class Stark extends BaseController
{
    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 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.

```
http://myapp.dev/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.

```
<?php

// app/routes.php

// Our Controller.
class Stark extends BaseController
{
    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', array('narcissist'));
});
```

Just as in the last example, we supply an array with a single parameter 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.

```
http://myapp.dev/tony/the/narcissist/genius
```

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 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.

```
<?php

// app/routes.php

Route::get('example', function()
{
    return URL::asset('img/logo.png');
});
```

Now, if we visit the URL `/example` then we are greeted with the following response.

```
http://myapp.dev/img/logo.png
```

Laravel has created an absolute asset path for us. If we want to use a the secure HTTPS protocol to reference our assets, then we can pass `true` as a second parameter to the `URL::asset()` method, like this:

```
<?php

// app/routes.php

Route::get('example', function()
{
    return URL::asset('img/logo.png', true);
});
```

Now we receive the following response from the `/example` URL.

```
https://demo.dev/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.

```
<?php

// app/routes.php

Route::get('example', function()
{
    return URL::secureAsset('img/logo.png');
});
```

The response from this route is the same as the previous method.

```
https://demo.dev/img/logo.png
```

Generation Shortcuts

The methods mentioned in the previous subheadings are freely available for you to use in your views. Go ahead and take advantage of all of the features that they have to offer.

However, it's good practice for the logic in your views to be short and neat. Also, it takes some stress off your fingers. This is why Laravel has provided some shortcuts to some of the methods available on the `URL` class. Let's take a closer look at what's available.

First we have the `url()` function. It accepts identical parameters to the `URL::to()` method, so I won't cover them again. Here's an example of it in action.

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

<a href="{{ url('my/route', array('foo', 'bar'), true) }}">My Route</a>
```

Now if we look at the link within the rendered view's source, we see the following.

```
<a href="https://demo.dev/my/route/foo/bar">My Route</a>
```

The URL has been created in the same manner as the `URL::to()` method. As before, there is also a shortcut method that can be used to generate a secure URL. It looks like this:

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

<a href="{{ secure_url('my/route', array('foo', 'bar')) }}">My Route</a>
```

The `secure_url()` function accepts the same signature as the `URL::secure()` method. The first parameter is the route, and the second is an array of route parameters to be appended.

The `route()` function is a shortcut to the `URL::route()` method, and be used for generating URLs to named routes. It looks like this:

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

<a href="{{ route('myroute') }}">My Route</a>
```

As you might have guessed, there is also a shortcut for the third method of route URL generation. The `action()` function can be used as a shortcut to the `URL::action()` method, and can be used to generate links to controller actions.

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

<a href="{{ action('MyController@myAction') }}">My Link</a>
```

Just as with the `URL::action()` method, they can accept second and third parameters for route parameters and secure URL generation.

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

<a href="{{ action('MyController@myAction', array('foo'), true) }}">My Link\
</a>
```

The shortcut to the `URL::asset()` method is the `asset()` function, and as with all the other shortcuts, it accepts identical function parameters. Here's an example:

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

<img src="{{ asset('img/logo.png') }}" />
```

Likewise, the shortcut to `URL::secureAsset()` is the `secure_asset()` function. It looks like this:

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

<img src="{{ secure_asset('img/logo.png') }}" />
```

Feel free to use the shortcuts in your views to simplify their content, and to avoid repetitive strain injury.

Request Data

Data, and its manipulation, is 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.

```
<?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->retrieveCurrentRequestByImaginaryFigur\
e();
        return $request->data->input->getDataByKey($requestDataIndicator);
    }
}

// app/routes.php

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

Right 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 real 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.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $data = Input::all();
    var_dump($data);
});
```

The `Input::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.

```
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.

```
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.

```
<!-- app/views/form.blade.php -->

<form action="{{ url('/') }}" method="POST">

    <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. However, we need to work on the routing before we can test this out. Let's take a look at the routing file.

```
<?php

// app/routes.php

Route::post('/', function()
{
    $data = Input::all();
    var_dump($data);
});

Route::get('post-form', function()
{
    return View::make('form');
});
```

Here we have added an additional 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.

```
array(2) { ["foo"]=> string(3) "bar" ["baz"]=> string(3) "boo" }
```

Great, our post data was retrieved correctly. Although 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.

```
<!-- app/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.

```
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 `Input::get()` method is for. Let's see it in action.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $data = Input::get('foo');
    var_dump($data);
});
```

We have changed the routing to use the `get()` method once more, but this time we are using the `Input::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.

```
string(3) "bar"
```

Great! I wonder what would happen if the data didn't exist? Let's find out by visiting `/`.

```
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 `Input::get()` method.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $data = Input::get('foo', 'bar');
    var_dump($data);
});
```

Now let's take a look at the result from the `/` URI.

```
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 `Input::has()` option. Let's take a look at it in action.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $result = Input::has('foo');
    var_dump($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 `Input::get()` 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 `Input::only()` method. It does exactly what it says on the... tin?

Here's an example.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $result = Input::only(array('foo', 'baz'));
    var_dump($result);
});
```

In the above example we pass the `Input::only()` method an array containing the keys of the request data values we wish to return as an associative array. Actually, the array is optional. We could also pass each key as additional parameters to the method, like this:

```
<?php

// app/routes.php

Route::get('/', function()
{
    $result = Input::only('foo', 'baz');
    var_dump($result);
});
```

Let's test the response by visiting the following 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.

```
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:

```
<?php

// app/routes.php

Route::get('/', function()
{
    $result = Input::except(array('foo', 'baz'));
    var_dump($result);
});
```

Or, we can provide the list of keys as additional parameters, like this:

```
<?php

// app/routes.php

Route::get('/', function()
{
    $result = Input::except('foo', 'baz');
    var_dump($result);
});
```

Now let's visit the following 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 `Input::only()` method.

```
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.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Redirect::to('new/request');
});

Route::get('new/request', function()
{
    var_dump(Input::all());
});
```

Here we have a route that redirects to the `new/request` route. Let's hand some `GET` data to the first route and see what happens. Here's the URL we will try.

```
http://myapp.dev/?foo=one&bar=two
```

Here's the response we receive after the redirect.

```
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 `Input::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 `Input::flash()` method.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    Input::flash();
    return Redirect::to('new/request');
});

Route::get('new/request', function()
{
    var_dump(Input::all());
});
```

Now if we hit the same URL we receive... oh hold on.

```
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 complicate things. Laravel and Taylor are clever. Together, they have stored the request data in another collection.

Let's alter our routes again.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    Input::flash();
    return Redirect::to('new/request');
});

Route::get('new/request', function()
{
    var_dump(Input::old());
});
```

The `Input::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.

```
array(2) { ["foo"]=> string(3) "one" ["bar"]=> string(3) "two" }
```

Great! That's exactly what we are looking for. We've now got hold of the data that we `flash()`ed from the previous request. As with the `Input::get()`, we can also retrieve a single value from the array of old data. Bring on the code!

```
<?php

// app/routes.php

Route::get('/', function()
{
    Input::flash();
    return Redirect::to('new/request');
});

Route::get('new/request', function()
{
    var_dump(Input::old('bar'));
});
```

By passing a string to the `Input::old()` method, we can specify a key to return a single value. It works just like the `Input::get()` method, except that it will act upon the 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 nice, descriptive example using Laravel's clean syntax? I completely agree!

```
<?php

// app/routes.php

Route::get('/', function()
{
    Input::flashOnly('foo');
    return Redirect::to('new/request');
});

Route::get('new/request', function()
{
```

```
13      var_dump(Input::old());
14  });
```

This time, we have told Laravel to only 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.

```
1  array(1) { ["foo"]=> string(3) "one" }
```

Laravel has provided only the value for `foo`. That's the only value that was saved for the redirected request, and it's exactly 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.

```
1  <?php
2
3  // app/routes.php
4
5  Route::get('/', function()
6  {
7      Input::flashExcept('foo');
8      return Redirect::to('new/request');
9  });
10
11 Route::get('new/request', function()
12 {
13     var_dump(Input::old());
14 });
```

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.

```
1  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:

```
Input::old('first', 'second', 'third');
Input::flashOnly('first', 'second', 'third');
Input::flashExcept('first', 'second', 'third');

Input::old(array('first', 'second', 'third'));
Input::flashOnly(array('first', 'second', 'third'));
Input::flashExcept(array('first', 'second', 'third'));
```

It really is up to you! The second option might be really 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 actually 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.

```
<?php

// app/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 nice of it, isn't it? Laravel loves you. It really 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.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    Input::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 `Input::flashOnly()`.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Redirect::to('new/request')->withInput(Input::only('foo'));
});
```

By passing the result of the `Input::only()` method to the `withInput()` chained method, we can limit the request data to the set of keys identified within the `Input::only()` method.

Similarly, we can pass the `Input::except()` method to the `withInput()` method to limit the request data to the inverse of the above example. Like this:

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Redirect::to('new/request')->withInput(Input::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 really attach a file to the current URL. Let's setup a simple form instead. We will start with a view.

```
<!-- app/views/form.blade.php -->

<form action="{{ url('handle-form') }}"
      method="POST"
      enctype="multipart/form-data">

    <input type="file" name="book" />
    <input type="submit">
</form>
```

So here we have a form with an 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?

```
<?php

// app/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.

```
<?php

// app/routes.php

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

Route::post('handle-form', function()
{
    var_dump(Input::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 a wonderful 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. It's a week into the release and I've already had to send 5 copyright infringement emails! Let's not add to that total!

Right, select the Code Bright PDF and hit that wonderful submit button! What response do we get?

```
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.

```
<?php

// app/routes.php

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

Route::post('handle-form', function()
{
    var_dump(Input::file('book'));
});
```

We have changed the second route to instead, dump the value of the `Input::file()` method, 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.

```
object(Symfony\Component\HttpFoundation\File\UploadedFile)#9 (7) {
  ["test":"Symfony\Component\HttpFoundation\File\UploadedFile":private]=>
  bool(false)
  ["originalName":"Symfony\Component\HttpFoundation\File\UploadedFile":priv\
ate]=>
  string(14) "codebright.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.

```
<?php

// app/routes.php

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

Route::post('handle-form', function()
{
```

```
    return Input::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 Bright PDF once more to see the result.

```
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.

```
<?php

// app/routes.php

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

Route::post('handle-form', function()
{
    return Input::file('book')->getClientOriginalName();
});
```

This time we will try the `getClientOriginalName()` method instead. Let's see what result we receive after uploading our book.

```
codebright.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 then the file name. Let's have a look at how we can find the file size.

```
<?php

// app/routes.php

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

Route::post('handle-form', function()
{
    return Input::file('book')->getClientSize();
});
```

What we receive after uploading our book is a number.

```
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.

```
<?php

// app/routes.php

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

Route::post('handle-form', function()
{
    return Input::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 Bright book.

```
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 it's mime type. Here's the `guessExtension()` method in action.

```
<?php

// app/routes.php

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

Route::post('handle-form', function()
{
    return Input::file('book')->guessExtension();
});
```

Uploading our book once more yields the following response.

```
pdf
```

Excellent, that's definitely 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.

```
<?php

// app/routes.php

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

```

```
Route::post('handle-form', function()
{
    return Input::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 Bright.

```
/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.

```
<?php

// app/routes.php

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

Route::post('handle-form', function()
{
    Input::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 own choosing. If we retrieve the file's real name and pass it as the second parameter to the `move()` method, it should arrive in the destination with a more sensible name.

Let's take a look at an example.

```php
<?php

// app/routes.php

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

Route::post('handle-form', function()
{
    $name = Input::file('book')->getClientOriginalName();
    Input::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 'codebright.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[20] 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 really. They are a method of storing some data on the client side; in the browser. They can be really handy for lots of things. For example, you might want to show your users a message the first time they visit your site. When 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.

[20] http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/File/UploadedFile.html

Setting and Getting Cookies

```
<?php

// app/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? Nice work 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 lets 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:

```
<?php

// app/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.

```
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.

```
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 `Input::get()`, the `Cookie::get()` method will accept a default value as the second parameter, like this:

```
$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.

```
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:

```php
<?php

// app/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 we wish to forget. Here's an example.

```php
<?php

// app/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 clever little monkey.

A user can edit the cookies stored within the browser. If your cookie was somehow important 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?

Forms

It was a cold, dark night in Los Angeles, but the lights of the Dolby Theatre lit the street so bright that you could be forgiven for thinking that it was daytime. You see, the venue had been the setting for the Academy Awards back in February, where the Iron Man movies had won all of the Oscars presented that night for being absolutely, completely, AWESOME.

In fact, if this book sells well, I might buy a mansion in California; perhaps build a super-lab underneath it. There I could work on an exoskeletal suit fashioned in the likeness of a red panda. Unlike Tony Stark, I'd have to leave the supermodels alone. I don't think Emma would stand for any of that funny business.

I'd call myself 'The Fire Fox', and use my powers and gadgets to settle any tabs versus spaces arguments that arise within the PHP community. I'd also have the power of receiving lawsuits from Mozilla. I'd probably not drive an Audi though. Instead, I… well. Sorry, I got a little carried away. Where were we?

Ah yes, the Dolby Theatre. You see, the Oscars were only paving way to a much more high profile event, the Laravel four premiere. PHP developers who had been saved by Laravel 3 gathered by the millions to await the launch of the new framework. Crowds littered the streets, reporters everywhere! Picture it, no really. You will have to imagine most of this, I'm not a fictional writer you see. Just picture it how you see it on the telly.

A long, black stretch limousine approaches the curb. The door bursts open and a tall figure steps out onto the red carpet. The crowd goes wild! Ladies rip off their tops, men rip off each other's tops, reporters rip off their tops, everyone is topless. I don't know where I'm going with this, but it's my story, and I say everyone is topless. Anyway, the figure of course is none other than Taylor Otwell, the creator of Laravel.

Before Taylor has taken a step towards the entrance of the theatre he is bombarded with a pack of reporters hungering for the inside scoop on Laravel four. He tries to shove the topless reporters away but there are too many of them.

> How long have you been working on Laravel four?

"A year or so now." says Taylor, realising that his only chance of proceeding would be to answer some of the questions. The topless reporter seems satisfied with having her question answered, and allows Taylor to pass.

Another topless reporter quickly steps in front of Taylor, pouncing with another question.

> How long have you been working on Laravel four?

Wait a second. Didn't he already get asked that? "Not again." thought Taylor, who hated repeating himself. "About a year." he said with a stern voice. He pushed forward through the crowd only to have his path blocked by yet another topless reporter.

> Laravel is quite a large project. How long did it take you to write it?

Something in Taylor snapped, he reached into his suit pocket and pulled out a keyboard. Taylor is kind of tall, long pockets... oh who cares how the keyboard fit in there, I'm making this up. He struck the reporter across the face with the keyboard, sending the topless chap flying into a nearby dumpster.

"Did you not read the documentation!?" Taylor screamed at the startled reporter, who was struggling to climb out of the greasy dumpster. "I built the form builder so that you can define your forms within templates. If you had all built a form before hand we could have avoided all this repetition. I HATE REPETITION", snarled Taylor. He spun around and darted for the door where the Laravel team were waiting for him backstage.

Yes. That was all just to build up to the form builder. Don't look at me so disappointed! I'm a technical writer, not a fictional one. What were you expecting... A Song of Ice and Fire? Right, let's just get on with it!

This chapter posed a real dilemma. Well, not just this chapter. Let's see if you can solve the problem better than I can. Here are topics that intertwine.

- Request Data
- Forms
- Validation

You see, they are all big topics, but they also rely on each other. You can't test the response from a form without knowing how to retrieve request data. You can't show form repopulation without first teaching validation. It's a real tangle.

Well, this is how I am going to resolve the problem. We are going to learn how to build forms and target them to framework routes. However, some of our forms explanation will take place in the next chapter, where we will learn how forms can be repopulated after validation.

Let's get started shall we?

Opening Forms

Before we start submitting data, we need to decide where we are going to send it. We took a look at how to generate framework URLs within the URL generation chapter. We could easily build a form open tag using the skills we picked up from that chapter, right?

Let's build a simple view:

```
<!-- app/views/form.blade.php -->

<form action="{{ url('our/target/route') }}" method="POST">

</form>
```

Here, we use the `url()` helper method to provide a framework URL for the action attribute of our form. Let's also add a routed closure to display our `form.blade.php` blade template.

```
<?php

// app/routes.php

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

Great! Now let's go ahead and visit the `/` url to see the source that was rendered. It looks like this.

```
<!-- app/views/form.blade.php -->

<form action="http://demo.dev/our/target/route" method="POST">

</form>
```

Our form now has a destination for the data that it collects. This is how I tend to construct my forms, but Laravel is a framework of options. Here's another method of building form opening HTML that you might prefer. Let's take a look.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'our/target/route')) }}

{{ Form::close() }}
```

To create our opening form tag, we use the `Form::open()` generator method. This method accepts only a single parameter, with many parameters. Confused? I'm talking about an array!

In the above example we are using the `URL` index, with a value of the route that we wish to be the target of the form. We also use the `Form::close()` method to close the form, although I see no reason why you would need to do this. I suppose it looks neater next to the other source generator methods? It's up to you whether you choose to use it, or `</form>` instead.

Here's the resulting source from our rendered form view.

```
<!-- app/views/form.blade.php -->

<form method="POST" action="http://demo.dev/our/target/route" accept-charse\
t="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
</form>
```

Wonderful, our form has been generated to the same target.

I'm going to ask you to do something really important here, could you please ignore the `_token` hidden attribute? Don't worry, we will come back to it later! For now, I'm going to pretend that it doesn't exist.

> Wait, we only provided a URL. Why are there other attributes?

Well done for changing the topic away from the `_token` input.

> What `_token`?

Great! Well, let's get back to your question. You see, Laravel knows that `POST` forms are the most popular choice when building web applications. For that reason it will use the `POST` method by default, overriding the default `GET` method that HTML assumes.

The `accept-charset` attribute is another useful attribute that Laravel provides for us. It will ensure that 'UTF-8' will be used for character encoding when submitting the form. It's a sensible default for most situations.

Perhaps sensible defaults don't suit our situation? What if we want to provide our own?

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'url' => 'our/target/route',
    'method' => 'GET',
    'accept-charset' => 'ISO-8859-1'
)) }}

{{ Form::close() }}
```

Here we have used the `method` index of our `Form::open()` array to specify that we wish to use `GET` as the method for our form. We have also used the `accept-charset` index to provide a different character set for encoding.

Here's the new HTML source.

```
<!-- app/views/form.blade.php -->

<form method="GET" action="http://demo.dev/our/target/route" accept-charset\
="ISO-8859-1">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
</form>
```

Great! While we value Laravel's opinion on acceptable defaults, we have all the flexibility we need to overrule them. Since we have so much power, why don't we create a form that uses the 'DELETE' HTTP verb as its method attribute.

We know that we can override the form method using the `method` array index. Let's give it a go.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'url' => 'our/target/route',
    'method' => 'DELETE'
)) }}

{{ Form::close() }}
```

That looks perfect to me, now let's check out the generated HTML source.

```
<!-- app/views/form.blade.php -->

<form method="POST" action="http://demo.dev/our/target/route" accept-charse\
t="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <input name="_method" type="hidden" value="DELETE">
</form>
```

Wait, what's going on there? We didn't want a 'POST' method. What's the extra input? Oh yes, I remember now. HTML forms are a little dumb. You see, HTML4 will only support 'POST' and 'GET' methods on forms. HTML5 does support additional methods, but we don't want to limit our application to browsers that fully support HTML5.

Don't worry, Laravel has another trick up its sleeve. By providing a hidden input called '_method' we can supply a value to represent our non HTML5 compatible HTTP verbs. Laravel's routing will look at this POST request, see the included '_method' data and route to the appropriate action. This will work on both HTML4 and HTML5, isn't that great?

Remember how we learned how to upload files and retrieve information about them in the last chapter? You might also remember that the form requires an 'enctype' attribute of value 'multipart/form-data' for files to be uploaded correctly.

Well Laravel provided an easy way of enabling the 'enctype' attribute. Let's see how it works.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'url'       => 'our/target/route',
    'files'     => true
)) }}

{{ Form::close() }}
```

By providing a new index named `files` with a boolean value of `true`, Laravel will add the necessary attribute to allow the uploading of files. Here's the generated source from the above example.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/our/target/route"
      accept-charset="UTF-8"
      enctype="multipart/form-data">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
</form>
```

I have formatted the response a little to make it more readable, but I haven't changed the content. As you can see, Laravel has supplied the correct encoding type on our behalf. Thanks again Laravel!

In the URL generation and routing chapters you may have learned that URLs to named routes and controller actions can be generated using special methods. Let's find out how we can target these routes and actions using special indexes on the `Form::open()` method.

First let's take a look at how to target named routes. Here's an example.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'route'       => 'my_route'
)) }}

{{ Form::close() }}
```

Instead of using the `url` index, we use the `route` index and provide the name of the route as a value. It's as simple as that! In fact, it's just as easy to target controller actions. Let's take a look at another example.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'action'       => 'MyController@myAction'
)) }}

{{ Form::close() }}
```

To target our form at a controller action, instead of the `url` and `route` indexes, we use the `action` index and the controller-action pair as its value.

Well, now that we know how to open forms, I think it's about time that we started adding some fields. Let's get going!

Form Fields

Forms aren't very useful unless they provide a means of collecting data. Let's look at some of the form fields that we can generate using Laravel's form builder library.

Field Labels

Wait, there's something else we need other than the fields themselves. We need labels so that people know what values to enter. Labels can be generated using the `Form::label()` method. Let's take a look at an example.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('first_name', 'First Name') }}
{{ Form::close() }}
```

The first parameter of the `Form::label()` method matches the `name` attribute of the field that they are describing. The second value, is the text of the label. Let's take a look at the generated source.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <label for="first_name">First Name</label>
</form>
```

Great! We should use labels whenever we can to describe our fields. I'll lead by example by including them with the other field examples within the chapter.

It's worth noting that we can include other attribute and value pairs within our generated label element by providing a third parameter to the `label()` method. The third parameter is an array of attribute and value pairs. Here's an example.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('first_name', 'First Name',
                    array('id' => 'first_name')) }}
{{ Form::close() }}
```

In the above example we have added an 'id' of value 'first_name' to the label element. Here's the generated source.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <label for="first_name" id="first_name">First Name</label>
</form>
```

Brilliant! Well, now that we know how to label our fields, it's about time to start creating some of them.

Text Fields

Text fields can be used to collect string data values. Let's take a look at the generator method for this field type.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('first_name', 'First Name') }}
    {{ Form::text('first_name', 'Taylor Otwell') }}
{{ Form::close() }}
```

The first parameter for the `Form::text()` method is the `name` attribute for the element. This is what we will use to identify the field's value within our request data collection.

The second parameter is an optional one. It is a default value for the input element. Let's see the source that has been generated by the method.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <label for="first_name">First Name</label>
    <input name="first_name" type="text" value="Taylor Otwell" id="first_na\
me">
</form>
```

Our input element has been created, along with its 'name' attribute and default value. Just like the `Form::label()` method, our `Form::text()` method can accept an optional third parameter to supply an array of HTML attribute and value pairs, like this:

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('first_name', 'First Name') }}
    {{ Form::text('first_name', 'Taylor Otwell',
                        array('id' => 'first_name')) }}
{{ Form::close() }}
```

The truth is, every single one of the input generators will accept an optional array of attributes as a final parameter. They all work in the same way, so I won't describe this feature in every section. Just keep it in mind!

Textarea Fields

The textarea field functions in a similar way to the text input type, except that they allow multiline data. Let's see how they can be generated.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('description', 'Description') }}
    {{ Form::textarea('description', 'Best field ever!') }}
{{ Form::close() }}
```

The first parameter is the `name` attribute for the element, and the second parameter once again is the default value. Here's the form element as it appears in the generated HTML source.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <label for="description">Description</label>
    <textarea name="description"
              cols="50"
```

```
            rows="10"
            id="description">Best field ever!</textarea>
</form>
```

As you can see from the generated source, Laravel has provided some default values, such as the amount of columns and rows for the field. We could easily override these values by adding new values for them to the optional final parameter. I hope you haven't forgot about the attributes array!

Password Fields

Some things are secret. Actually, you can tell me your secrets if you like. I'm not the type of person who would publish them as jokes in a technical book for extra laughs.

However, if we want our users to be able to type their innermost secrets into our application securely, then we need to make sure that their characters don't display on screen. The 'password' input type is perfect for this. Here's how you can generate one.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('secret', 'Super Secret') }}
    {{ Form::password('secret') }}
{{ Form::close() }}
```

The first parameter for the `Form::password()` method is the `name` attribute. As always, there is a final optional parameter for an array of additional element attributes.

Here's the generated source from the above example.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <label for="secret">Super Secret</label>
    <input name="secret" type="password" value="" id="secret">
</form>
```

Checkboxes

Checkboxes allow for your form to capture boolean values. Let's take a look at how we can generate a checkbox form element.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('pandas_are_cute', 'Are pandas cute?') }}
    {{ Form::checkbox('pandas_are_cute', '1', true) }}
{{ Form::close() }}
```

The first parameter to the `Form::checkbox()` method is the field `name` attribute, and the second parameter is the field `value`. The third, optional value is whether or not the checkbox will be checked by default. The default is for the checkbox to be unchecked.

Here's the source generated by the above example.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <label for="pandas_are_cute">Are pandas cute?</label>
    <input checked="checked"
           name="pandas_are_cute"
           type="checkbox"
           value="1"
           id="pandas_are_cute">
</form>
```

Radio Buttons

Radio buttons have an identical method signature to that of checkboxes. The difference is that you can use a number of radio input types to provide a method of selection within a small group of values. Here's an example.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('panda_colour', 'Pandas are?') }}
    {{ Form::radio('panda_colour', 'red', true) }} Red
    {{ Form::radio('panda_colour', 'black') }} Black
    {{ Form::radio('panda_colour', 'white') }} White
{{ Form::close() }}
```

In the above example you will notice that we have a number of radio buttons with the same name attribute. Only one of them will be able to be selected at once. This will allow our users to choose the colour of a panda.

Here's how our generated source looks when the form view is rendered.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <label for="panda_colour">Pandas are?</label>
    <input checked="checked"
           name="panda_colour"
           type="radio"
           value="red"
           id="panda_colour"> Red
    <input name="panda_colour"
           type="radio"
           value="black"
           id="panda_colour"> Black
    <input name="panda_colour"
           type="radio"
           value="white"
           id="panda_colour"> White
</form>
```

Select Boxes

Select boxes, or drop-downs, are another way of providing a choice of values to the user of your application. Let's take a look at how they can be constructed.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('panda_colour', 'Pandas are?') }}
    {{ Form::select('panda_colour', array(
        'red'       => 'Red',
        'black'     => 'Black',
        'white'     => 'White'
    ), 'red') }}
{{ Form::close() }}
```

The `Form::select()` method accepts a `name` attribute as the first parameter, and an array of key-value pairs as a second parameter. The key for the selected option will be returned within the request data. The third, optional, parameter is the key for a value that will appear as selected by default. In the above example, the 'Red' option will be selected when the page loads.

Here's the generated source.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <label for="panda_colour">Pandas are?</label>
    <select id="panda_colour" name="panda_colour">
        <option value="red" selected="selected">Red</option>
        <option value="black">Black</option>
        <option value="white">White</option>
    </select>
</form>
```

If we want to, we can organise our select box options by category. To enable this option, all we need to do is provide a multidimensional array as the second parameter. The first level of the array will be the category, and the second level will be the list of values just as before.

Here's an example.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('bear', 'Bears are?') }}
    {{ Form::select('bear', array(
        'Panda' => array(
            'red'       => 'Red',
            'black'     => 'Black',
            'white'     => 'White'
        ),
        'Character' => array(
            'pooh'      => 'Pooh',
            'baloo'     => 'Baloo'
        )
    ), 'black') }}
{{ Form::close() }}
```

We have added a new dimension to our values array. The top level of the array is now split between 'Panda' and 'Character'. These two groups will be represented by 'optgroup' element types within the drop-down.

Here's the generated source code.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <label for="bear">Bears are?</label>
    <select id="bear" name="bear">
        <optgroup label="Panda">
            <option value="red">Red</option>
            <option value="black" selected="selected">Black</option>
            <option value="white">White</option>
        </optgroup>
        <optgroup label="Character">
            <option value="pooh">Pooh</option>
            <option value="baloo">Baloo</option>
        </optgroup>
    </select>
</form>
```

Email Field

The email field type has the same method signature as the `Form::text()` method. The difference is that the `Form::email()` method creates a new HTML5 input element that will validate the value provided client side to ensure that a valid email address will be provided.

Here's another example.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::label('email', 'E-Mail Address') }}
    {{ Form::email('email', 'me@daylerees.com') }}
{{ Form::close() }}
```

The first parameter to the `Form::email()` method is the `name` attribute for our input type. The second parameter is a default `value`. As always, and I hope you haven't forgotten, there's always a last, optional, array of additional element attributes.

Here's how the page source for the above example would be rendered.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <label for="email">E-Mail Address</label>
    <input name="email"
           type="email"
           value="me@daylerees.com"
           id="email">
</form>
```

File Upload Field

In the previous chapter, we learned how to handle uploaded files. Let's see how we can generate a file upload field.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'url'   => 'my/route',
    'files' => true
)) }}
    {{ Form::label('avatar', 'Avatar') }}
    {{ Form::file('avatar') }}
{{ Form::close() }}
```

The first parameter for the `Form::file()` method is the `name` attribute of the file upload element, however, in order for the upload to work, we need to ensure that the form is opened with the `files` option to include the multipart encoding type.

Here's the generate page source.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8"
      enctype="multipart/form-data">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <label for="avatar">Avatar</label>
    <input name="avatar" type="file" id="avatar">
</form>
```

Hidden Fields

Sometimes our fields aren't meant to capture input. We can use hidden fields to supply extra data along with our forms. Let's take a look at how hidden fields can be generated.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::hidden('panda', 'luishi') }}
{{ Form::close() }}
```

The first parameter to the `Form::hidden()` method is the `name` attribute, I bet you didn't see that coming? The second parameter is of course the `value`.

This is how the generated page source looks.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <input name="panda" type="hidden" value="luishi">
</form>
```

Form Buttons

Our forms are no good if we can't submit them. Let's have a close look at the buttons that we have available to us.

Submit Button

First up is the submit button. There's nothing quite like a classic! Here's how it looks.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::submit('Save') }}
{{ Form::close() }}
```

The first parameter to the `Form::submit()` is the `value`, which in the case of a button is the label used to identify the button. As with all of the input generation methods, the button generation methods will accept an optional last parameter to provide additional attributes.

Here's the generated HTML source code from the above example.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <input type="submit" value="Save">
</form>
```

Great! We can now submit our forms. Let's have another look at an alternative form of button.

Normal Buttons

When we need a button that won't be used to submit our form, we can use the `Form::button()` method. Here's an example.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::button('Smile') }}
{{ Form::close() }}
```

The parameters for the `Form::button()` method are exactly the same as the `Form::submit()` method mentioned previously. Here's the generated source from the example.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <button type="button">Smile</button>
</form>
```

Image Buttons

Instead of a native button, you could use the HTML5 image button type to submit your form. Here's an example.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::image(asset('my/image.gif', 'submit')) }}
{{ Form::close() }}
```

The first parameter to the `Form::image()` method is the URL to an image to use for the button. I have used the `asset()` helper method to generate the URL. The second parameter is the value of the button.

Here's the generated source code.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <input src="https://demo.dev/my/image.gif" type="image">
</form>
```

Reset Button

The reset button can be used to clear the contents of your form. It's used by your application's users when they have made a mistake. Here's an example of how to generate one.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'my/route')) }}
    {{ Form::reset('Clear') }}
{{ Form::close() }}
```

The first parameter to the `Form::reset()` method is the label that you would like to appear on the button. Here's the generated source from the above example.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/my/route"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
    <input type="reset" value="Clear">
</form>
```

Form Macros

In the previous subsection we discovered the various form input generator methods. They can be used to save you a lot of time, but perhaps you have a custom type of form input that is specific to your own applications.

Lucky for us, Taylor already thought about this. Laravel comes equipped with a method that will allow you to define your own form generators, let's see how it works.

So, where shall we put our form macros? Ah I know, let's make a file called `app/macros.php`, then we can include it from within our routes file. Here's the contents:

```
<?php

// app/macros.php

Form::macro('fullName', function()
{
    return '<p>Full name: <input type="text" name="full_name"></p>';
});
```

To define a macro we use the `Form::macro()` method. The first parameter is the name that will be used by the method that generates our form field. I'd suggest making it `camelCase` for this very reason.

The second parameter to the method is a Closure. The value returned from this closure must be the source code required to construct our field.

Let's create a simple route to show the result from our new form macro. Here it is:

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Form::fullName();
});
```

Our route is returning the value of the `fullName()` method on the `Form` class. This isn't your average static method, it's a bit of magic that Laravel provides to call our own form macros. The name of the method matches up to the nickname that we gave to our form macro. Let's take a look at what is returned from the route we just created.

```
<p>Full name: <input type="text" name="full_name"></p>
```

As you can see, the result of our Closure has been returned.

What if we want to supply parameters to our form macros? Sure, no problem! Let's first alter our form macro.

```php
<?php

// app/macros.php

Form::macro('fullName', function($name)
{
    return '<p>Full name: <input type="text" name="'.$name.'"></p>';
});
```

By providing placeholders within our macro closure, we can use their values within our rendered source. In the above example we have added a method to provide a name attribute for our new input type.

Let's take a look at how this new parameter can be provided.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Form::fullName('my_field');
});
```

Well, that was easy wasn't it? We just pass the value for our parameter to the magical method that Laravel has provided for us. Let's take a look at the new result of our `/` route.

```
<p>Full name: <input type="text" name="my_field"></p>
```

Awesome, the new name value has been inserted into our form field! There's another great way to avoid repetition. Enjoy!

Form Security

It's great when we receive data from our forms. It's what they are there for right? The trouble is, if our routes can be a target of our own forms, then what's to stop an external source posting some dangerous data to our site?

What we need is a way of making sure that the data that is sent to us belongs to our own website. Not a problem. Laravel can handle this for us. You should probably know that this form of attack is known as 'Cross Site Request Forgery' or CSRF for short.

Now then, do you remember that `_token` hidden field that I told you not to worry about? Well now I want you to start worrying about it.

> ARGH. We are all gonna die!

Right, well done. Let me explain what this token is. It's kind of like a secret passphrase that Laravel knows about. It's been created using the 'secret' value from our application config. If we tell Laravel to check for this value within our input data, then we can ensure that the data has been provided by a form that belongs to our application.

We could check for the token ourselves by comparing its value to the result of the `Session::token()` method, but Laravel has provided a better option. Do you remember the defaults from our filters chapter? Well, Laravel has included the `csrf` filter. Let's attach it to a route.

```
<?php

// app/routes.php

Route::post('/handle-form', array('before' => 'csrf', function()
{
    // Handle our posted form data.
}));
```

By attaching the `csrf` filter as a `before` filter to the route that will handle our form data, we can ensure that the `_token` field is present and correct. If our token is not present, or has been tampered with, then a `Illuminate\Session\TokenMismatchException` will be thrown, and our route logic will not be executed.

In a later chapter we will learn how to attach error handlers to certain types of exceptions in an effort to handle this circumstance appropriately.

If you used `Form::open()` to create the wrapping form markup, then the security token will be included by default. However, if you want to add the token to manually written forms, then you must simply add the token generator method. Here's an example.

```
<!-- app/views/form.blade.php -->

<form action="{{ url('handle-form') }}" method="POST">
    {{ Form::token() }}
</form>
```

The `Form::token()` method will insert the hidden token field into our form. Here's the source that will be generated.

```
<!-- app/views/form.blade.php -->

<form action="http://demo.dev/handle-form" method="POST">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NE\
cAwFoRFTq6u">
</form>
```

To summarise, unless you want external applications and users to be able to post data to your forms, you should use the 'CSRF' filter to protect the routes which handle your form data.

Validation

A few months ago, I was getting really stressed out. Work was mounting for the release of Laravel 4, which meant designing and building the new site, responding to documentation alterations, along with the typical support role that any framework developer will have to assume. They were hard times. On top of all of that work, I also had the pressure of trying to build Code Bright into as much of a success as Code Happy was so that I could bring a bunch of new developers into the community.

What I should have done is taken a short break at one of Phill's Parks. You see, Phill's Parks are a number of holiday parks located around the world that are holiday destinations for developers. They were originally opened by CEO of Phill's Parks, Phill Sparks, back in the year two thousand, and are now the only place that developers can go to unwind, and feel comfortable in a bathing suit.

Phill's Parks offer a wide variety of activities for developers wishing to wind down after a hard year of work. These events include competitive FizzBuzz tournaments, tabs versus spaces debates, and neck-beard pageants.

If I had known about Phill's Parks, I would have been able to chill out, and enjoy the weeks leading up to the Laravel four launch. Oh well, at least you know about them right? As a lucky reader of Code Bright, you have access to a coupon that will allow a 90% discount off entry to the parks. To take advantage of this deal, simply track down Phill Sparks on IRC, Twitter, E-Mail or in person, and quote coupon code 'CODE_BRIGHT_TROLLS_YOU'. If he doesn't respond right away, just keep spamming him with it. Sooner or later he will hook you up with a coding holiday.

> So what does this have to do with Validation?

Well, when I started writing this chapter, I had a wonderful, magnificent plan in mind. Sadly, I have gotten completely distracted by the fun and games offered by Phill's Parks and now I have no idea. I'll just make something up, it's turned out fine so far right?

Phill's Parks are a very special place for developers, and developers only. The problem with developers, is that without care an attention, they can be hard to determine from other roles within the web development industry, for example, designers.

We don't want designers sneaking into Phill's Parks posing as developers. They will start judging the choice of decoration, and complaining about the use of Comic Sans on the cafeteria menus. It will completely ruin the atmosphere! No, we don't want them in the park at all. What we need is some way of validating our visitors, to ensure that they are developers. See, I told you I could bring it back.

What we validate, is a set of attributes. Let's think about some of the attributes of a park visitor. Oh, oh, I know. We can create a list! Lists are fun!

- Favourite beverage.
- Choice of web browser.
- Facial hair type.
- Ability to talk to females.

Excellent, we have a number of attributes that can be used to describe visitors to the park. Using some validation constraints or rules, we can be sure that our visitor is a developer. For the park, we will just make sure that the attributes of a park visitor match the following requirements.

- Favourite beverage: Beer.
- Choice of web browser: Google Chrome.
- Facial hair type: Neck-beard.
- Ability to talk to females: None.

Disclaimer: These values are just for fun, for example, I know many female web developers who are fantastic at what they do, and I'm fairly sure they have the ability to talk to other females. Also, I'm sporting a rather fabulous van dyke facial hair combo right now, and not the traditional neck-beard. It does have beer soaked into it though…

Anyway, be ensuring that the attributes of a park visitor, match those of a developer, we can go ahead and let them into the park. However, if a shifty fella steps up to the reception with the following attributes…

- Favourite beverage: Starbuck Frappechino.
- Choice of web browser: Mobile Safari.
- Facial hair type: None.
- Ability to talk to females: Talk at females, sure.

…then oh no, we have a designer, or a Ruby developer, and we can't let them through the gates.

Once more, this is just a bit of fun, please don't send me hate mail, wait until I kill off a main character in a later chapter… erm, I mean, just forget it.

What we have done is implemented validation so that we can be sure that the people who enter the park are developers, and not imposters.

Phill's parks are now safe, but what about your applications? Let's have a look at how we can use validation in Laravel to make sure that we get exactly what we expect.

Simple Validation

Don't trust your users. If there's one thing that I have learned working in the IT industry, is that if you have a point of weakness in your application, then your users will find it. Trust me when I say that they will exploit it. Don't give them the opportunity. Use validation to ensure that you receive good input.

What do we mean by good input? Well let's have a look at an example.

Let's say that we have a HTML form that is used to collect registration information for our application. In fact, it's always great to have a little review session. Let's create the form with the Blade templating engine, and the form builder.

Here's our view.

```
<!-- app/views/form.blade.php -->

<h1>Registration form for Phill's Parks</h1>

{{ Form::open(array('url' => 'registration')) }}

    {{-- Username field. ------------------------}}
    {{ Form::label('username', 'Username') }}
    {{ Form::text('username') }}

    {{-- Email address field. -------------------}}
    {{ Form::label('email', 'Email address') }}
    {{ Form::email('email') }}

    {{-- Password field. ------------------------}}
    {{ Form::label('password', 'Password') }}
    {{ Form::password('password') }}

    {{-- Password confirmation field. -----------}}
    {{ Form::label('password_confirmation', 'Password confirmation') }}
    {{ Form::password('password_confirmation') }}

    {{-- Form submit button. --------------------}}
    {{ Form::submit('Register') }}

{{ Form::close() }}
```

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 textfield 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, 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 own 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.

```
<?php

// app/routes.php

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

Route::post('/registration', function()
{
    $data = Input::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? Well OK then, I guess I'll show you.

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.

```php
<?php

// app/routes.php

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

Route::post('/registration', function()
{
    $data = Input::all();

    $rules = array(
        '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.

```
array(
    '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.

```php
<?php

// app/routes.php

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

```

```
Route::post('/registration', function()
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
        'username' => 'alpha_num'
    );

    // Create a new validator instance.
    $validator = Validator::make($data, $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 checked whether or not the data we provided conforms to the validation constraints that we have provided. Let's set up an example.

```
<?php

// app/routes.php

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

Route::post('/registration', function()
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
        'username' => 'alpha_num'
    );

    // Create a new validator instance.
    $validator = Validator::make($data, $rules);
```

```
22
23      if ($validator->passes()) {
24          // Normally we would do something with the data.
25          return 'Data was saved.';
26      }
27
28      return Redirect::to('/');
29  });
```

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 indicated that the data conforms to all of the validation rules. A `false` indicates that the data does not meet the validation requirements.

We use 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 the validation will pass. We are presented with the following sentence.

```
1  Data was saved.
```

Great, that's what we expected. Now 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.

```
1  <?php
2
3  // app/routes.php
4
5  Route::get('/', function()
6  {
7      return View::make('form');
8  });
9
10 Route::post('/registration', function()
```

```
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
        'username' => 'alpha_num'
    );

    // Create a new validator instance.
    $validator = Validator::make($data, $rules);

    if ($validator->fails()) {
        return Redirect::to('/');
    }

    // 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 are able to accept parameters. Let's swap out our `alpha_num` rule for the `min` one. Here's the source.

```
<?php

// app/routes.php

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

Route::post('/registration', function()
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
```

```
17          'username' => 'min:3'
18      );
19
20      // Create a new validator instance.
21      $validator = Validator::make($data, $rules);
22
23      if ($validator->passes()) {
24          // Normally we would do something with the data.
25          return 'Data was saved.';
26      }
27
28      return Redirect::to('/');
29  });
```

The `min` validation rule ensures that the value is greater than 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 3 characters long. On a numerical value, it will ensure that the value is mathematically greater than our parameter. Finally, on uploaded files, the `min` validation rule will ensure that the uploaded file's size in bytes is greater than the provided parameter.

Let's test out our new validation rule. First visit the `/` page and enter the value 'Jo' into the username field. If you submit the form, 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 soil it with 'she said' jokes.

Right, we have now used two validation constrains, 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.

```php
<?php

// app/routes.php

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

Route::post('/registration', function()
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
        'username' => 'alpha_num|min:3'
    );

    // 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('/');
});
```

As you can see, we are able to pass a number of 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 in work, and a Mac at home, and it just took me 30 seconds to find the pipe key.

Anyway, there's an alternative way to provide multiple validation rules. If you aren't a 60s grandfather then you might not appreciate pipes. If you like, you can use a multidimensional array to specify additional validation rules.

Here's an example.

```php
<?php

// app/routes.php

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

Route::post('/registration', function()
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
        'username' => array('alpha_num', 'min:3')
    );

    // 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('/');
});
```

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. Put the book down and go to sleep. You are 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. It's intended purpose is for when you wish to ensure that the user has agreed to something, for example, a terms and conditions checkbox.

```
array(
    '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.

```
array(
    'field' => 'active_url'
);
```

---

after

The 'after' validation rule accepts a single parameter, a string representation of a 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.

```
array(
    'field' => 'after:12/12/13'
);
```

---

alpha

The 'alpha' validation rule ensures that the provided value consists entirely of alphabetical characters.

```
array(
    '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'.

```
array(
    '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.

```
array(
    'field' => 'alpha_num'
);
```

---

before

The 'before' 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.

```
array(
    'field' => 'before:12/12/13'
);
```

---

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 upon the length of the string in characters. On a file, the comparison will be based upon the size of the file in bytes.

```
array(
    'field' => 'between:5,7'
);
```

---

confirmed

The 'confirmed' validation rule can be used to ensure that another field exists, that matches the current 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'.

```
array(
    '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.

```
array(
    'field' => 'date'
);
```

---

date_format

The 'date_format' validation rule will ensure that our value is 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.

```
array(
    '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.

```
array(
    '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.

```
array(
    '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 of 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.

```
array(
    '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:

```
array(
    '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`.

```
array(
    'field' => 'image'
);
```

---

in

The 'in' validation rule will ensure that the value of the field matches one of the provided parameters.

```
array(
    '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!

```
array(
    'field' => 'integer'
);
```

---

ip

The 'ip' validation rule will check to make sure that the value of the field contains a well formatted IP address.

```
array(
    '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 upon the size of the file in bytes.

```
array(
    '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.

```
array(
    '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 upon the size of the file in bytes.

```
array(
    '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.

```
array(
    'field' => 'not_in:blue,green,pink'
);
```

---

numeric

The ‘numeric’ rule will check to make sure that the field being validated contains a numeric value.

```
array(
    '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 very large 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.

```
array(
    'field' => 'regex:[a-z]'
);
```

---

required

The 'required' validation rule can be used to ensure that the current field exists within the validation data array.

```
array(
    '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.

```
array(
    'field' => 'required_if:username,zoidberg'
);
```

---

required_with

The 'required_if' is used to ensure that the current value is present, only if one or more fields defined by the rule parameters are also present.

```
array(
    'field' => 'required_with:age,height'
);
```

---

required_without

The 'required_without' rule is the direct opposite to 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.

```
array(
    '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.

```
array(
    '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 upon the size of the file in bytes.

```
array(
    '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 users provide username is unique when handling registration forms.

```
array(
    'field' => 'unique:users,username'
);
```

You can provide extra optional parameters to list a number of IDs for rows that will be ignored by the unique rule.

```
array(
    '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.

```
array(
    '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, it doesn't offer the user much in the way of constructive feedback.

Fortunately, Laravel collects a number of error messages describing why the validation attempt failed. Let's take a look at how we can access this information.

```php
<?php

// app/routes.php

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

Route::post('/registration', function()
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
        'username' => 'alpha_num'
    );

    // 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.';
    }

    // Collect the validation error messages object.
    $errors = $validator->messages();

    return Redirect::to('/');
});
```

In the above example, you will see that we can access the validation error messages object using the `messages()` method of our validator instance. Now, 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.

```php
<?php

// app/routes.php

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

Route::post('/registration', function()
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
        'username' => 'alpha_num'
    );

    // 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);
});
```

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.

```php
<?php

// app/routes.php

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

Route::post('/registration', function()
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
        'username'   => 'required|alpha_num|min:3|max:32',
        'email'      => 'required|email',
        'password'   => 'required|confirm|min:3'
    );

    // 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);
});
```

Now let's see how we can access our error messages from the form view.

```
<!-- app/views/form.blade.php -->

<h1>Registration form for Phill's Parks</h1>

{{ Form::open(array('url' => 'registration')) }}

    <ul class="errors">
    @foreach($errors->all() as $message)
        <li>{{ $message }}</li>
    @endforeach
    </ul>

    {{-- Username field. ------------------------}}
    {{ Form::label('username', 'Username') }}
    {{ Form::text('username') }}

    {{-- Email address field. -------------------}}
    {{ Form::label('email', 'Email address') }}
    {{ Form::email('email') }}

    {{-- Password field. ------------------------}}
    {{ Form::label('password', 'Password') }}
    {{ Form::password('password') }}

    {{-- Password confirmation field. -----------}}
    {{ Form::label('password_confirmation', 'Password confirmation') }}
    {{ Form::password('password_confirmation') }}

    {{-- Form submit button. --------------------}}
    {{ Form::submit('Register') }}

{{ Form::close() }}
```

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, so 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, thats 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! Now our application's users are 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.

```
<!-- app/views/form.blade.php -->

<h1>Registration form for Phill's Parks</h1>

{{ Form::open(array('url' => 'registration')) }}

    {{-- Username field. ------------------------}}
    <ul class="errors">
    @foreach($errors->get('username') as $message)
        <li>{{ $message }}</li>
    @endforeach
    </ul>
    {{ Form::label('username', 'Username') }}
    {{ Form::text('username') }}

    {{-- Email address field. -------------------}}
    <ul class="errors">
    @foreach($errors->get('email') as $message)
        <li>{{ $message }}</li>
    @endforeach
    </ul>
    {{ Form::label('email', 'Email address') }}
    {{ Form::email('email') }}

    {{-- Password field. ------------------------}}
    <ul class="errors">
    @foreach($errors->get('password') as $message)
        <li>{{ $message }}</li>
    @endforeach
    </ul>
    {{ Form::label('password', 'Password') }}
    {{ Form::password('password') }}
```

```
33
34      {{-- Password confirmation field. -----------}}
35      {{ Form::label('password_confirmation', 'Password confirmation') }}
36      {{ Form::password('password_confirmation') }}
37
38      {{-- Form submit button. --------------------}}
39      {{ Form::submit('Register') }}
40
41  {{ Form::close() }}
```

We can use the `get()` method on the validation errors object to retrieve an array of errors for a single field. Simply pass the name of the field as the first parameter to the `get()` method.

Let's resubmit the form. This time, place a single exclamation `!` mark within the username field. Let's take a look at the errors that appear above the username field.

- The username may only contain letters and numbers.
- The username must be at least 3 characters.

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.

Let's take a look at how this is done with the Laravel validation errors object.

```
1   <!-- app/views/form.blade.php -->
2
3   <h1>Registration form for Phill's Parks</h1>
4
5   {{ Form::open(array('url' => 'registration')) }}
6
7       {{-- Username field. ------------------------}}
8       <span class="error">{{ $errors->first('username') }}</span>
9       {{ Form::label('username', 'Username') }}
10      {{ Form::text('username') }}
11
12      {{-- Email address field. -------------------}}
13      <span class="error">{{ $errors->first('email') }}</span>
14      {{ Form::label('email', 'Email address') }}
15      {{ Form::email('email') }}
16
17      {{-- Password field. ------------------------}}
18      <span class="error">{{ $errors->first('password') }}</span>
19      {{ Form::label('password', 'Password') }}
```

```
20      {{ Form::password('password') }}
21
22      {{-- Password confirmation field. -----------}}
23      {{ Form::label('password_confirmation', 'Password confirmation') }}
24      {{ Form::password('password_confirmation') }}
25
26      {{-- Form submit button. --------------------}}
27      {{ Form::submit('Register') }}
28
29  {{ Form::close() }}
```

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. This means that you can use it without having to check for the existence of messages. However, if for some reason you wish to check whether or not an error message exists for a field, you can use the `has()` method.

```
1  @if($errors->has('email'))
2      <p>Yey, an error!</p>
3  @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.

```
<!-- app/views/form.blade.php -->

<h1>Registration form for Phill's Parks</h1>

{{ Form::open(array('url' => 'registration')) }}

    <ul class="errors">
    @foreach($errors->all('<li>:message</li>') as $message)
        {{ $message }}
    @endforeach
    </ul>

    {{-- Username field. ------------------------}}
    {{ Form::label('username', 'Username') }}
    {{ Form::text('username') }}

    {{-- Email address field. -------------------}}
    {{ Form::label('email', 'Email address') }}
    {{ Form::email('email') }}

    {{-- Password field. ------------------------}}
    {{ Form::label('password', 'Password') }}
    {{ Form::password('password') }}

    {{-- Password confirmation field. -----------}}
    {{ Form::label('password_confirmation', 'Password confirmation') }}
    {{ Form::password('password_confirmation') }}

    {{-- Form submit button. --------------------}}
    {{ Form::submit('Register') }}

{{ Form::close() }}
```

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.

```
<!-- app/views/form.blade.php -->

<h1>Registration form for Phill's Parks</h1>

{{ Form::open(array('url' => 'registration')) }}

    {{-- Username field. ------------------------}}
    {{ $errors->first('username', '<span class="error">:message</span>') }}
    {{ Form::label('username', 'Username') }}
    {{ Form::text('username') }}

    {{-- Email address field. -------------------}}
    {{ $errors->first('email', '<span class="error">:message</span>') }}
    {{ Form::label('email', 'Email address') }}
    {{ Form::email('email') }}

    {{-- Password field. ------------------------}}
    {{ $errors->first('password', '<span class="error">:message</span>') }}
    {{ Form::label('password', 'Password') }}
    {{ Form::password('password') }}

    {{-- Password confirmation field. -----------}}
    {{ Form::label('password_confirmation', 'Password confirmation') }}
    {{ Form::password('password_confirmation') }}

    {{-- Form submit button. --------------------}}
    {{ Form::submit('Register') }}

{{ Form::close() }}
```

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.

```php
<?php

// app/routes.php

Validator::extend('awesome', function($field, $value, $params)
{
    return $value == 'awesome';
});

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

Route::post('/registration', function()
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
        '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 within 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.

```php
<?php

// app/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 fail.

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 customise your validation rules as required.

If you prefer to define your custom validation rule within a class, rather than than a 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.

```php
<?php

// app/validators/CustomValidation.php

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 own project structure. I chose to put my validation class within the `app/validators` folder, and classmap 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.

```
1  <?php
2
3  // app/routes.php
4
5  Validator::extend('awesome', '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 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 really sure. I guess I think of it as a hobby at this point. Anyway, let's example the array of custom validation messages.

```php
<?php

// app/routes.php

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

Route::post('/registration', function()
{
    // Fetch all request data.
    $data = Input::all();

    // Build the validation constraint set.
    $rules = array(
        'username'   => 'min:3',
    );

    // Build the custom messages array.
    $messages = array(
        '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 big example, let's hit 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 Â±.

> 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.

```
<?php

// Build the custom messages array.
$messages = array(
    '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 normally a good idea to do so.

```
<?php

// Build the custom messages array.
$messages = array(
    '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 specific 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.

```php
<?php

// Build the custom messages array.
$messages = array(
    '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 database migrations.

Coming Soon

> Hey, where's the rest of my book?

As I clearly wrote on the book's description page, this title is published while in progress. Since you are seeing this page, it means it's not done yet.

Don't worry though, I have big plans for this title. I will continue to write chapters, and add fixes and updates until I deem the book a complete source of knowledge for the framework. All of the future chapters and updates are available for free to those who have purchased the title. Just head over to Leanpub whenever you get an email from them about updates. It's as simple as that!

I'd also like to take this time to thank each and every one of you for supporting my writing. I have really enjoyed writing both titles, and intend to write a whole lot more in the future. If you guys hadn't supported me by buying my books, and emailing me with your feedback, I probably wouldn't have found this wonderful hobby (job?) of mine.

If Code Happy and Code Bright have helped you in anyway, then I would really appreciate if you would share the URL to the book with your friends. It's at http://leanpub.com/codebright[21], in case you lost it. :)

My overall plan for the title at present is the following.

- Describe basic framework concepts and components in detail.
- Write a 'build-an-app' chapter for a simple application, building on what we have learned.
- Further explore advanced framework features.
- Several 'build-an-app' chapters based on the advanced features.
- Best practice, and clever tricks.

I'm also considering an FAQ chapter based on the feedback that I get from the title, so if you have a burning question that isn't likely to be covered by a future feature chapter, then please let me know.

If you have anything you want to talk about you can contact me at me@daylerees.com[22] or daylerees on Twitter[23]. You will often find me hanging around in `#laravel` on Freenode IRC too, but please don't be offended if I don't reply right away, I have a day job too!

Thanks once again for being part of Code Bright.

Love,

Dayle, and his faithful army of red pandas.

xxx

[21] http://leanpub.com/codebright
[22] mailto:me@daylerees.com
[23] http://twitter.com/daylerees