CODE SMART

THE LARAVEL FRAMEWORK *VERSION 5* FOR BEGINNERS

DAYLE REES

Laravel: Code Smart

The Laravel Framework Version 5 for Beginners

Dayle Rees

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

This version was published on 2016-07-03

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

Tweet This Book!

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

The suggested tweet for this book is:

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

The suggested hashtag for this book is #codesmart.

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

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

Also By Dayle Rees

Laravel: Code Happy

Laravel: Code Happy (ES)

Laravel: Code Happy (JP)

Laravel: Code Bright

Code Happy (ITA)

Laravel: Code Bright (ES)

Laravel: Code Bright (SR)

Laravel: Code Bright (JP)

Laravel: Code Bright (IT)

Laravel: Code Bright (TR) Türkçe

Laravel: Code Bright (PT-BR)

PHP Pandas (PHP7!)

Laravel: Code Bright (RU)

PHP Pandas (ES)

PHP Pandas (IT)

PHP Pandas (FR)

PHP Pandas (TR)

PHP: Composer

Contents

Acknowledgements . . . . . **i**

Errata . . . . . **ii**

Feedback . . . . . **iii**

Translations . . . . . **iv**

1. Introduction **1**

2. Installation **2**
Install Software Dependencies 2
Create a Laravel Project 3
Install Homestead 4
Mastering Vagrant 5

3. Lifecycle **7**
Request 7
Services 8
Routing 8
Logic 8
Response 8

4. Configuration **10**
Configuration Files 10
Environmental Variables 11
Configuration Caching 14

5. Basic Routing **15**
Defining Routes 15
Route Parameters 20

Acknowledgements

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

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

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

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

Errata

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

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

[1] mailto:me@daylerees.com

Feedback

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

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

Translations

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

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

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

1. Introduction

Why hello there! Pleased to meet you.

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

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

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

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

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

2. Installation

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

> This is no time for home-making!

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

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

Install Software Dependencies

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

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

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

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

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

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

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

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

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

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

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

Create a Laravel Project

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

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

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

Example 01: Navigate to projects directory.

```
1 cd ~/Projects
```

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

Example 02: Create a Laravel project.

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

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

Install Homestead

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

Example 03: Navigate to the project directory.

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

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

Example 04: Add homestead to composer dependencies.

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

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

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

Example 05: Install homestead.

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

Well, at least that one was super quick!

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

Example 06: Boot the virtual machine.

```
vagrant up
```

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

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

Example 07: Add a local DNS entry.

```
192.168.10.10  homestead.app
```

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

Mastering Vagrant

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

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

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

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

Example 08: Boot the virtual machine.

```
vagrant up
```

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

Example 09: Halt the virtual machine.

```
vagrant halt
```

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

Example 10: Update the virtual machine settings.

```
vagrant provision
```

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

Example 11: SSH into the virtual machine.

```
vagrant ssh
```

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

Example 12: Destroy the virtual machine.

```
vagrant destroy
```

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

3. Lifecycle

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

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

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

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

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

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

Request

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

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

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

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

Services

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

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

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

Routing

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

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

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

Logic

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

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

Response

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

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

Let's summarize, shall we?

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

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

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

4. Configuration

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

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

Configuration Files

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

Example 01: Configuration files.

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

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

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

Example 02: Accessing configuration values.

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

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

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

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

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

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

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

Environmental Variables

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

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

Example 03: Set an environmental variable.

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

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

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

Example 04: Our .env file.

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

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

CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

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

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

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

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

Example 05: The env() function.

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

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

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

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

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

> I guess not!

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

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

Configuration Caching

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

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

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

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

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

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

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

5. Basic Routing

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

Example 01: A URL

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

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

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

Defining Routes

Example 02: Our first route.

```php
<?php

// app/Http/routes.php

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

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

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

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

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

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

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

Example 03: Routing methods.

```
<?php

// app/Http/routes.php

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

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

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

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

Example 04: Our first route, again.

```
<?php

// app/Http/routes.php

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

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

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

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

Example 05: Separate closure.

```
<?php

// app/Http/routes.php

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

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

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

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

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

Example 06: Multiple routes.

```
<?php

// app/Http/routes.php

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

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

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

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

Example 07: Multiple URLs.

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

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

Example 08: No path.

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

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

Example 09: Route with no path.

```
<?php

// app/Http/routes.php

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

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

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

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

Example 10: With or without a slash.

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

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

Consider the following site structure:

Example 11: Imaginary website.

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

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

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

Example 12: Route our imaginary site.

```
<?php

// app/Http/routes.php

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


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

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

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

Route Parameters

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

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

Example 13: Route parameters.

```php
<?php

// app/Http/routes.php

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

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

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

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

Example 14: Parameter in the URL.

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

You would be greeted with this text response:

Example 15: Output.

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

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

Example 16: Optional parameter.

```
<?php

// app/Http/routes.php

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

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

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

Example 17: Default parameter values.

```
<?php

// app/Http/routes.php

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

If we visit the following URL:

Example 18: URL with missing parameter.

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

We will receive this response:

Example 19: Output.

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

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

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