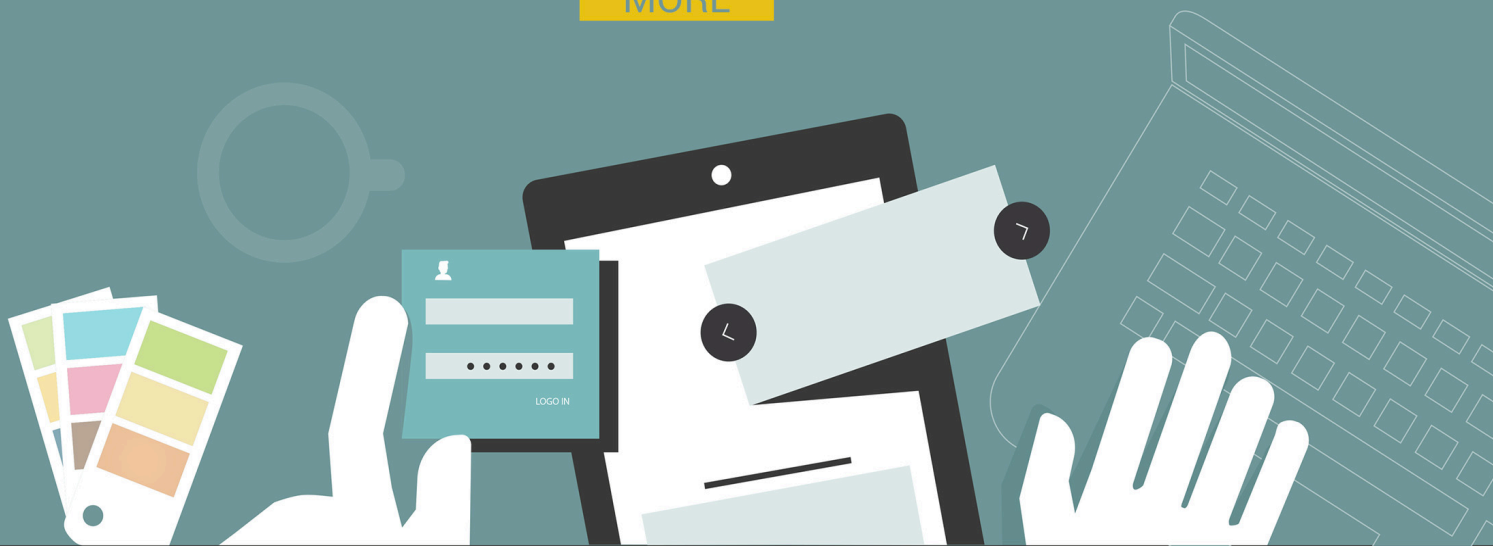


LARAVEL 5.1 BEAUTY

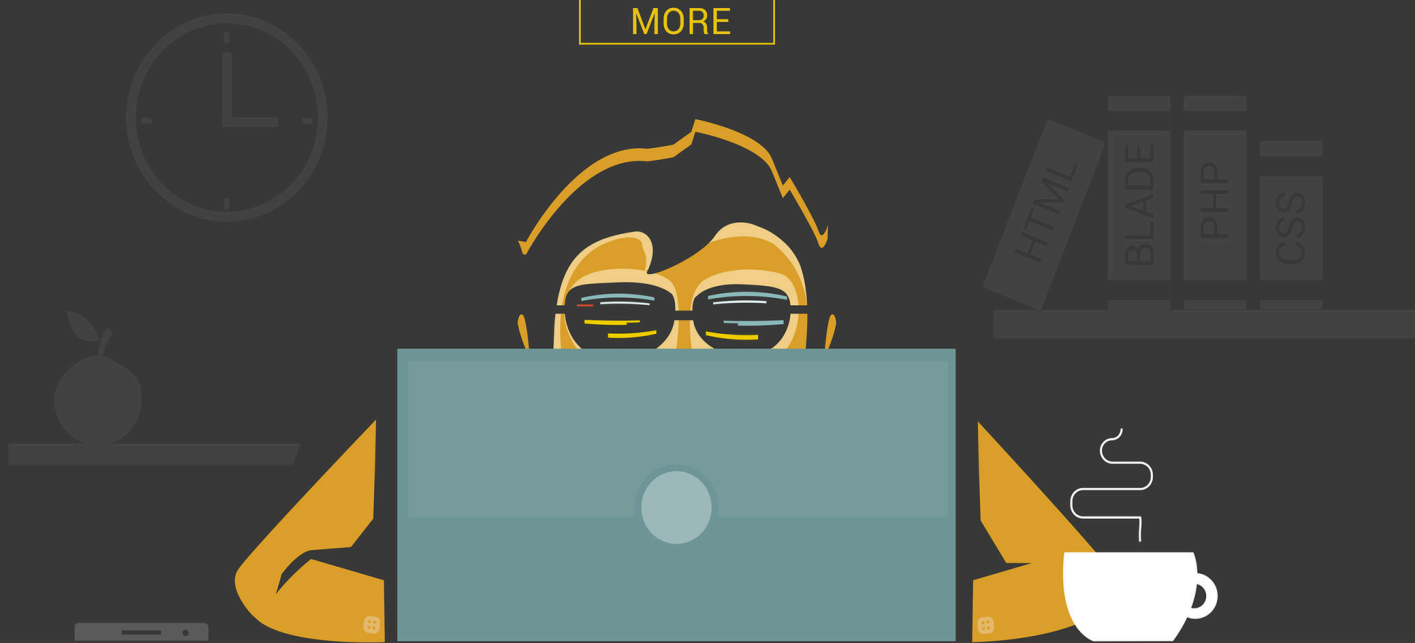
Creating Beautiful Web
Apps in Laravel 5.1

MORE



<BY CHUCK HEINTZELMAN>

MORE



Laravel 5.1 Beauty

Creating Beautiful Web Apps with Laravel 5.1

Chuck Heintzelman

This book is for sale at <http://leanpub.com/l5-beauty>

This version was published on 2015-07-16



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

©2014 - 2015 Chuck Heintzelman

Also By **Chuck Heintzelman**

[Getting Stuff Done with Laravel 4](#)

[Getting Stuff Done with Laravel 4 \(TR\) Türkçe](#)

[Laravel4でこなすプログラム術](#) [Getting Stuff Done](#)

Contents

Thank You	i
The Source Code is on GitHub	i
Feedback	i
Other places to learn Laravel 5.1	i
Chapter 1 - Introduction	1
Chapter Contents	1
Long Term Support	1
Why This Book	1
GitHub and the Blog	1
What is the Application?	2
Conventions Used This Book	2
Have Fun	4
Chapter 2 - Required Software and Components	5
Chapter Contents	5
The Rise of the Virtual Machines	5
About Laravel Homestead	5
Installing Virtual Box	6
Installing Vagrant	7
Where Do I Execute Things?	9
Recap	11
Chapter 3 - Setting up a Windows Machine	12
Chapter Contents	12
Multiple Ways to Setup Windows	12
Step 1 - Installing PHP Natively	12
Step 2 - Install Node.js	14
Step 3 - Install Composer	17
Step 4 - Install GIT and set up SSH Key	18
Step 5 - Adding the Homestead box	19
Step 6. Installing Homestead	20
Step 7 - Bring up the Homestead VM	21
Step 8 - Setting up PuTTY	22

CONTENTS

Step 9 - Installing Laravel's Installer	23
Recap	24
Chapter 4 - Setting up an OS X or Linux Machine	25
Chapter Contents	25
Slight Variations with Linux	25
Step 1 - Installing PHP	25
Step 2 - Install Node.js	26
Step 3 - Install Gulp	28
Step 4 - Install Composer	29
Step 5 - Adding SSH Keys	30
Step 6 - Adding the Homestead box	30
Step 7 - Installing Homestead	31
Step 8 - Bring up the Homestead VM	32
Step 9 - Installing the Laravel Installer	33
Recap	34
Chapter 5 - Homestead and Laravel Installer	35
Chapter Contents	35
The Homestead Tool	35
Overview of Common Homestead Commands	36
Examining Homestead.yaml	37
Adding Software to the Homestead VM	39
Daily Workflow	40
Six Steps to Starting a New Laravel 5.1 Project	40
Other Homestead Tips	44
Recap	44
Chapter 6 - Testing	45
Chapter Contents	45
Creating the l5beauty Project	45
Running PHPUnit	48
Using Gulp for TDD	54
Creating a Markdown Service	56
Other Ways to Test	62
Recap	63
Chapter 7 - The 10 Minute Blog	64
Chapter Contents	64
Pre-work before the 10 Minute Blog	64
0:00 to 2:30 - Creating the Posts table	65
2:30 to 5:00 - Seeding Posts with test data	67
5:00 to 5:30 - Creating configuration	69
5:30 to 7:30 - Creating the routes and controller	70

CONTENTS

7:30 to 10:00 - Creating the views	72
Recap	75
Chapter 8 - Starting the Admin Area	76
Chapter Contents	76
Establishing the Routes	76
Creating the Admin Controllers	79
Creating the Views	80
Testing logging in and out	87
Recap	91
Chapter 9 - Using Bower	92
Chapter Contents	92
Stealing Code	92
Installing Bower	93
Pulling in Bootstrap	94
Creating admin.less	96
Gulping Bootstrap	96
Running gulp	98
Updating the admin layout	98
Adding FontAwesome and DataTables	100
Recap	103
Chapter 10 - Blog Tags	104
Chapter Contents	104
Creating the Model and Migrations	105
Implementing admin.tag.index	108
Implementing admin.tag.create	112
Implementing admin.tag.store	118
Implementing admin.tag.edit	121
Implementing admin.tag.update	125
Finishing the Tag System	127
Recap	129
Chapter 11 - Upload Manager	130
Chapter Contents	130
Configuring the File System	130
Adding a Helpers file	132
Creating an Upload Manager Service	134
Implementing UploadController index	138
Finishing the Upload Manager	144
Setting Up Your S3 Account	155
Configuring L5Beauty to Use S3	158
Installing an Additional Package	159

CONTENTS

Test The Upload Manager	160
Fixing Bucket Permissions	161
Recap	163
Chapter 12 - Posts Administration	164
Chapter Contents	164
Modifying the Posts table	164
Updating the Models	167
Adding Selectize.js and Pickadate.js	171
Creating the Request Classes	175
Creating the PostFormFields Job	177
Adding to helpers.php	180
Updating the Post Model	181
Updating the Controller	182
The Post Views	186
Removing the show route	196
Recap	196
Chapter 13 - Cleaning Up the Blog	197
Chapter Contents	197
Using the Clean Blog Template	197
Creating the BlogIndexData Job	199
Updating the BlogController	202
Building the Assets	203
The Blog Views	208
Adding a Few Model Methods	215
Updating the Blog Config	218
Updating our Sample Data	219
Recap	224
Chapter 14 - Sending Mail and Using Queues	225
Contents	225
Setting Up for Emails	225
Adding a Contact Us Form	230
About Queues	237
Queuing the Contact Us Email	238
Automatically Processing the Queue	240
Queing Jobs	242
Recap	243
Chapter 15 - Adding Comments, RSS, and a Site Map	244
Contents	244
The Problem with Comments	244
Adding Disqus Comments	245

CONTENTS

Adding Social Links	249
Creating a RSS Feed	251
Create a Site Map	256
Recap	259
Chapter 16 - General Recap and Looking Forward	260
Contents	260
Testing	261
Eloquent Models and the Fluent Query Builder	261
Advanced Routing	261
Migrations, Seeding, and Model Factories	262
Dependency and Method Injection	262
Facades vs. helpers vs. IoC objects	263
Laravel Elixir	263
Tinker	264
Artisan Commands	264
Events	265
Form Requests	265
Blade Template Engine	266
Flysystem	267
Queues	267
Blog Features to Add	267
Final Recap and Thank You	268

Thank You

Thank you for purchasing this book. I hope you find it informative and useful.

The Source Code is on GitHub

The source code for each chapter having code in this book is on GitHub. Go to [ChuckHeintzelman/l5beauty](https://github.com/ChuckHeintzelman/l5beauty)¹ and then select the branch for the chapter you want to view.

Feedback

Feedback is encouraged!

If you find a typo, have a correction, or just want to comment on something you've found useful please drop by LaravelCoding.com² and comment on the appropriate chapter where you've found an issue.

Other places to learn Laravel 5.1

- [The Laravel Web Site](http://laravel.com)³ - The documentation there is a great place to start.
- [Laracasts](http://laracasts.com)⁴ - The video tutorials created by Jeffrey Way are unparalleled.

¹<https://github.com/ChuckHeintzelman/l5beauty>

²<http://LaravelCoding.com/blog/?tag=L5+Beauty>

³<http://laravel.com>

⁴<http://laracasts.com>

Chapter 1 - Introduction

Chapter Contents

- [Long Term Support](#)
- [Why This Book](#)
- [GitHub and the Blog](#)
- [What is the Application?](#)
- [Conventions Used This Book](#)
- [Have Fun](#)

Long Term Support

Laravel version 5.1 is the first LTS (long term support) Laravel release. This means bug fixes are provided for 2 years and security fixes are provided for 3 years.

This is important because the applications you build today will still be supported by the framework tomorrow.

Why This Book

My previous book on Laravel, *Getting Stuff Done with Laravel 4*⁵ was well received. Now that Laravel 5.1 is available, I briefly thought of updating my previous book to work with Laravel 5.1. The new version of Laravel implements big changes from Laravel 4, but Laravel 5.1 is mostly backwards compatible.

But the *Getting Stuff Done with Laravel 4* book isn't really a manual covering every aspect of Laravel 4. It's a process and design book. The principles discussed within that book still are valid in Laravel 5.1, even if the implementation may vary slightly.

Instead of updating my previous book, I've created a new book, *Laravel 5.1 Beauty*, to highlight some of the new features. This book is bigger and better than *Getting Stuff Done with Laravel 4*.

GitHub and the Blog

I'm publishing *Laravel 5.1 Beauty* simultaneously, as it is being built, on my web site [LaravelCoding.com](http://laravelcoding.com)⁶ and on [Leanpub](http://leanpub.com)⁷.

⁵<https://leanpub.com/gettingstuffdonelaravel>

⁶<http://laravelcoding.com>

⁷<http://leanpub.com>



The Source Code is on GitHub

The source code for the application built in this book is available on GitHub at [Chuck-Heintzelman/l5beauty](https://github.com/ChuckHeintzelman/l5beauty)⁸. Just switch the branch at github to the chapter you're working on.

This book has a different tone than my previous book. No lame attempts to be funny. (*I guess we all can't be Dayle Rees.*)

Laravel 5.1 Beauty goes through the process of creating, designing and coding a real-world application while focusing on the the architecture that makes Laravel the number one PHP framework available today.

What is the Application?

Throughout this book we'll build a simple, clean and beautiful blogging application along with the administration required to maintain the blog.

My own Laravel blog, LaravelCoding.com, uses the same blogging application developed here.

Conventions Used This Book

There are a few conventions used throughout this book.

Code is indented two spaces

The standard indentation for PHP code is 4 spaces. Since this book is available in a variety of eBook formats and some devices with small screens don't have much horizontal space, code within this book is indented 2 spaces instead of 4 to save space.

```
for ($i = 1; $i <= 10; $i++) {  
    echo "I can count to $i\n";  
}
```

Lines that end with backslash (\) should be continued

If you see any line ending with a backslash, that means the code should continue uninterrupted with text from the next line.

⁸<https://github.com/ChuckHeintzelman/l5beauty>

```
$ here_is_a_really_really_long_command that_has_a_long list of_arguments\  
which should continue
```

In the above line, even though two lines are shown you should type in everything, excluding the backslash into one line.



Be Careful of This One

When you're typing code and miss this it can cause an issue. When in doubt check the GitHub [l5beauty](#) repository.

Different prompts used for Windows, OS X (or Linux), and Homestead

Whenever a Windows command prompt is used, the prompt always begins with C: and ends with the > symbol.

```
C: \some\path>
```

Whenever the OS X Console or Linux console is used, the prompt also ends with the > symbol, but slashes are used instead of backslashes. Often there's a tilde (~) in the path.

```
~/some/path>
```

Whenever the console is generic (meaning it could be Windows, OS X, or Linux console depending on your host operating system) a prompt ending with a percent sign % is used.

```
/some/path%
```

Finally, whenever the console for the Homestead Virtual Machine is used, the standard dollar sign \$ prompt is used. (*The majority of the book uses the Homestead Virtual Machine.*)

```
~/somepath$
```

With the Homestead Virtual Machine, your prompt actually shows the username and hostname before the path. For example: `vagrant@homestead:~$`, but the username and hostname are only occasionally illustrated.



Sometimes the path is missing

Whenever the path is omitted from the prompt in one of the versions of the console windows, it is assumed you are in the current project directory.

Have Fun

I hope you enjoy this book and learn Laravel 5.1 with it. Be sure and follow along, set up your development machine, and create the application step-by-step, chapter-by-chapter.

Above all. Have fun. Coding in Laravel 5.1 is great fun.

Chapter 2 - Required Software and Components

This chapter discusses what software and components are required to develop applications with Laravel 5.1 and why they're required. Instructions to install VirtualBox and Vagrant are provided.

Chapter Contents

- [The Rise of the Virtual Machines](#)
- [About Laravel Homestead](#)
- [Installing Virtual Box](#)
- [Installing Vagrant](#)
- [Where Do I Execute Things?](#)
- [Recap](#)

The Rise of the Virtual Machines

Over the last few years, virtual machines have come into their own. Virtual Machines (or VMs) allow one computer system (the host operating system) to emulate another one. Sure, VMs have been around for a while, but now with increased processor speed and cheap memory VMs can be on every developer's desktop.

Laravel embraces VM technology and packages it's own "box" with the most common requirements for web applications. This pre-packaged development environment is called [Laravel Homestead](#)⁹.

About Laravel Homestead

One of the driving philosophies behind Laravel is to make PHP development both fun and easy. To this end Laravel provides a development environment called Laravel Homestead. [Vagrant](#)¹⁰ is used to manage the virtual machine. Under the hood [VirtualBox](#)¹¹ provides the interface to the host operating system.

⁹<http://laravel.com/docs/5.1/homestead>

¹⁰<http://vagrantup.com>

¹¹<http://virtualbox.org>

A **car** is the perfect metaphor for how this all works together. **Homestead** is the driver's seat of the car, **Vagrant** is the car's frame, and **VirtualBox** is the engine. Once Vagrant and VirtualBox are installed, there's no need to worry about them again. All interaction with the VM occurs through Homestead. *(Just like when driving a car, there's no need to worry about the frame or engine.)*

Laravel Homestead allows you to use a virtual Ubuntu Linux machine, pre-installed with the software required for web development. This VM includes:

- Ubuntu 14.04
- PHP 5.6
- HHVM
- Nginx
- MySQL
- PostgreSQL
- Node (With Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd
- Laravel Envoy
- Fabric + HipChat Extension

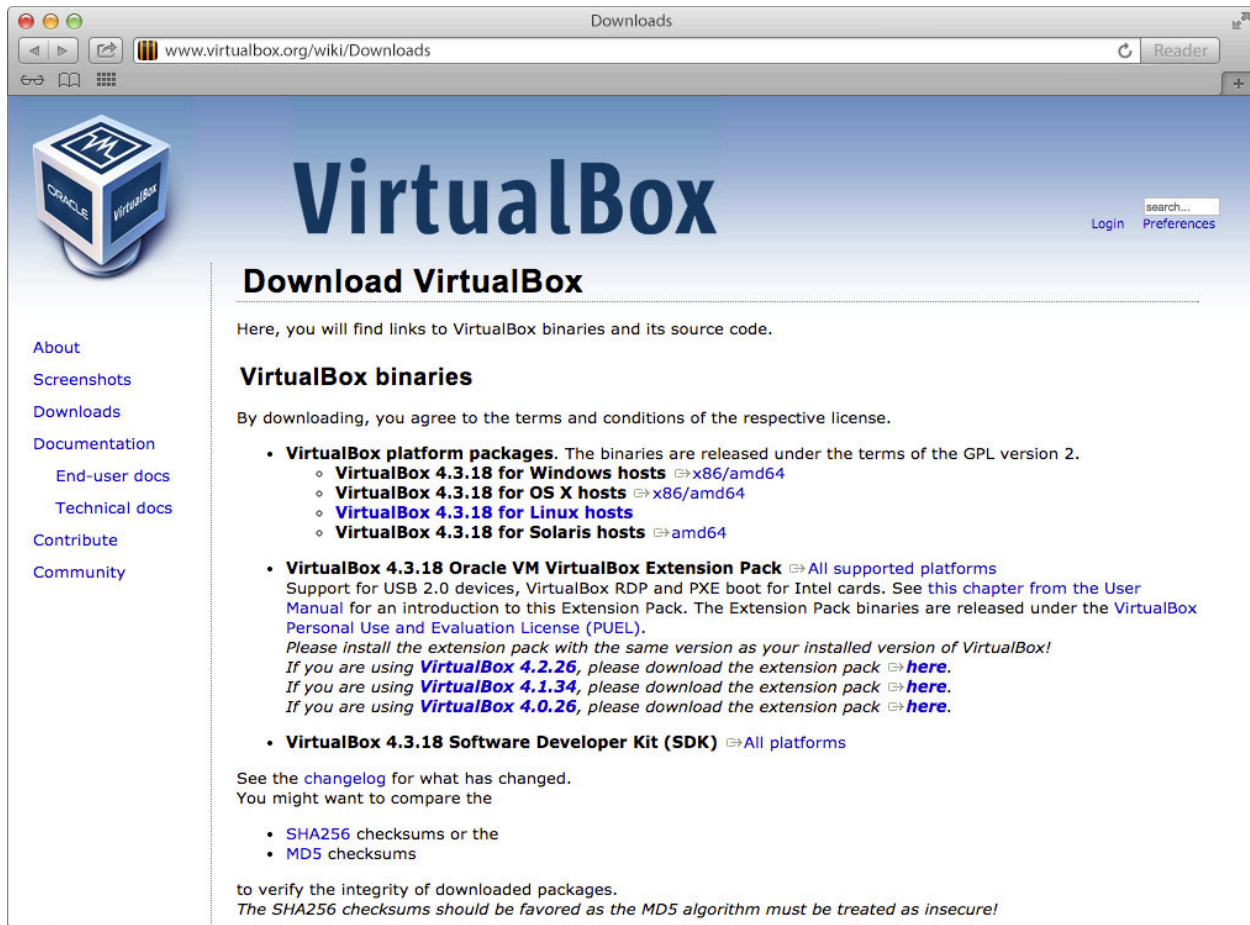
And best of all, Laravel Homestead allows the same development environment to be used on Windows, OS X, or Linux systems without worrying about conflicting software on the host machine.

Installing Virtual Box

Vagrant requires a back-end provider to provide the virtual machine it will manage. If you already have VirtualBox, VMWare, or another compatible [provider](https://docs.vagrantup.com/v2/getting-started/providers.html)¹² you can skip this step.

But if you don't yet have a back-end installed, use the VirtualBox platform package. It's free and works on every major platform.

¹²<https://docs.vagrantup.com/v2/getting-started/providers.html>

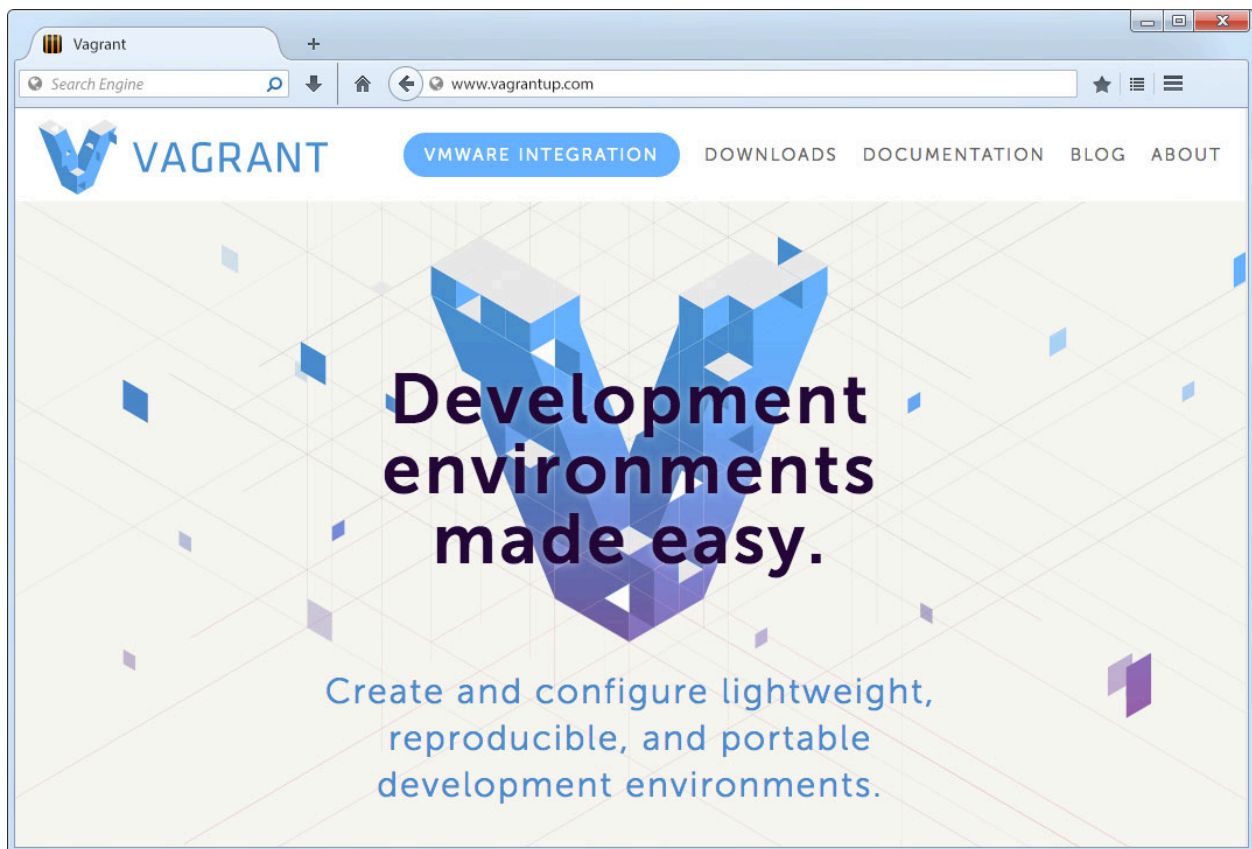


VirtualBox Download Page

Go to 'www.virtualbox.org'](https://www.virtualbox.org/wiki/Downloads), download and install the package for your operating system.

Installing Vagrant

Once you have VirtualBox (or another back-end provider) installed, you need to install Vagrant.



Vagrant Home Page

Go to www.vagrantup.com, download and install the package for your operating system.

When the Vagrant installation is complete, you may need to reboot your machine. After the reboot, verify Vagrant is installed by opening the console (command prompt in Windows, terminal in OS X or Linux) and checking the version.

Checking the Vagrant Version

```
% vagrant --version
Vagrant 1.6.5
```



Vagrant Windows Install Location

Unlike most software installed within Windows, Vagrant is not available in the Windows Start menu. It installs into the `C:\HashiCorp` directory and adds `C:\HashiCorp\Vagrant\bin` to your Windows system path.

Where Do I Execute Things?

After going through the next few chapters installing and running Laravel Homestead a common question is “*Where do I run ...?*” or “*Where does ... run?*” This section provides a brief overview of the major components of Laravel development within Homestead and answers the questions.

The Web Server

The Web Server runs in the Homestead Virtual Machine.

Nginx is the web server used to serve the web pages. The Host OS can access the web pages using the standard HTTP port (80) at the address 192.168.10.10. The Host OS can also access web pages at 127.0.0.1 on port 8000.

Editing Files

Always edit your source code from the Host OS.

The edited files are immediately available in the Homestead VM through shared folders.

MySQL

MySQL runs within the Homestead Virtual Machine.

You can access MySQL from your Host OS with the following information.

Setting Name	Setting Value
Host	127.0.0.1
Port	33060
Username	homestead
Password	secret

Memcached

Memcached is an in-memory key/value cache. It runs within the Homestead Virtual Machine.

Beanstalkd

Beanstalkd is a simple and fast work queue. It runs within the Homestead Virtual Machine.

Git or Subversion

Run from your Host OS.

Although you *can* run these version control systems from either place, it is **strongly** recommended to only run them from your Host OS. Consistently running them in one location avoids potential conflicts.

For example, let's say you install subversion in the Homestead Virtual Machine and it's version 1.8. You check out source code within the Homestead Virtual Machine and then try to check it in from your Host OS. If subversion v1.7 is installed on your Host OS you won't be able to do anything until upgrading subversion on your Host OS.

Bower

Bower is a simple to use package manager for the web. You can run this from either place if bower's installed on your Host OS.

Gulp

Gulp is a simple build system Laravel Elixir uses to concatenate assets, minify assets, combine assets, copy assets, and automate unit tests.

ONLY run this from your Host OS.

When running Gulp from your Host OS, growl-like notifications will appear in your OS when certain tasks are performed (such as LESS files compiled). If you execute Gulp within the Homestead Virtual Machine there will be warning errors when these notification attempts are made.

Composer

Only run Composer from your Host OS.

If you're Host OS is OS X or Linux you can run from either place, but if your Host OS is Windows then Composer creates necessary batch files required to operate correctly.

Artisan

Only run `artisan` from the Homestead Virtual Machine. The main reason for this is that any specific database, queue, and cache drivers are installed within Homestead and may not be available (or installed) on your Host OS. Also, the database setting of `localhost` is from the Homestead VM perspective, not from your Host OS's perspective.



The rule for running commands in the console

The rule is: *Only run artisan in the Homestead VM.* Everything else can or must be executed from your Host OS.

Recap

In this chapter we discussed the various software required to develop applications in Laravel 5.1 and installed VirtualBox and Vagrant.

If your machine is a Windows box, continue to the next chapter, *Setting up a Windows Machine*. Otherwise, skip to the chapter *Setting up an OS X or Linux Machine*.

Chapter 3 - Setting up a Windows Machine

This chapter goes through the steps required to set up and install the supporting software for Laravel Homestead on a Windows machine. It is assumed **VirtualBox** and **Vagrant** were already installed from the previous chapter.

If you're using OS X or Linux, please skip to the next chapter.

Chapter Contents

- [Multiple Ways to Setup Windows](#)
- [Step 1 - Installing PHP Natively](#)
- [Step 2 - Install Node.js](#)
- [Step 3 - Install Composer](#)
- [Step 4 - Install GIT and set up SSH Key](#)
- [Step 5 - Adding the Homestead box](#)
- [Step 6 - Installing Homestead](#)
- [Step 7 - Bring up the Homestead VM](#)
- [Step 8 - Setting up PuTTY](#)
- [Step 9 - Installing Laravel's Installer](#)
- [Recap](#)

Multiple Ways to Setup Windows

With Windows, there's quite a few different paths you can go down to install the required software. I tried multiple methods searching for the combination presented below. This chapter has been tested with Windows 8.1 but should work fine with Windows 7.

Step 1 - Installing PHP Natively

The first step is to get PHP running on Windows.

Step 1.1 - Download / Unzip PHP

Go to windows.php.net/download¹³ and download the latest Zip file for your machine. For my machine I downloaded the **VC11 x64 Thread Safe** version. (php-5.6.10-Win32-VC11-x64.zip at the time of this writing.)

Unzip this file into the C:\Php directory.

Step 1.2 - Update PHP.INI

Open up a command prompt and do the following to create the php.ini file.

Copy php.ini-development to php.ini

```
C:\Users\Chuck> cd \php  
C:\Php> copy php.ini-development php.ini
```

Then edit php.ini in a text editor and change the following lines.

Changes in php.ini

```
// change  
; extension_dir = "ext"  
// to  
extension_dir = "ext"  
  
// change  
;extension=php_openssl.dll  
// to  
extension=php_openssl.dll  
  
// change  
;extension=php_mbstring.dll  
// to  
extension=php_mbstring.dll
```

Now, within the C:\Php directory, you should be able to execute php.

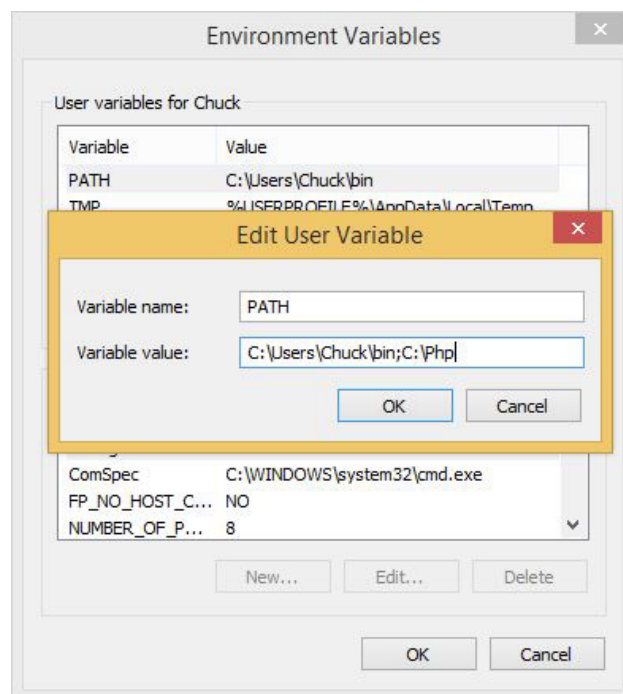
¹³<http://windows.php.net/download>

Checking the PHP version

```
C:\Php> php --version
PHP 5.6.10 (cli) (built: Oct 30 2014 16:05:53)
Copyright (c) 1997-2014 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2014 Zend Technologies
```

Step 1.3 - Add C:\Php to the path

- Open up the *Windows Control Panel*
- Search for *env* in the top right corner
- Click on the **Edit environment variables for your account** link
- If **PATH** is already in your User variables, then **[Edit...]** it, adding `;C:\Php` to the end, otherwise add it.



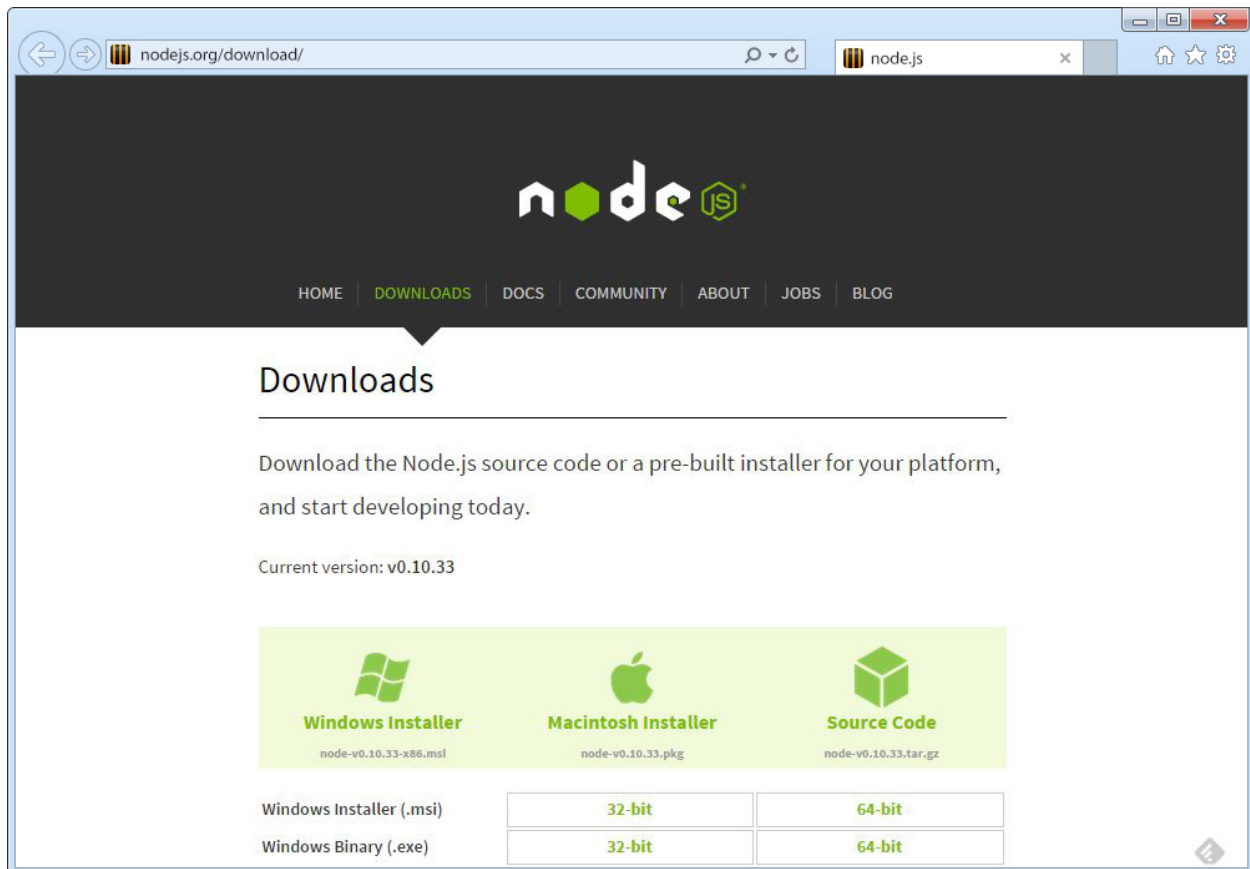
Adding PHP to Windows Path

The next time you open up a command prompt, `php` will be available in your path.

Step 2 - Install Node.js

We'll install Node.js natively in Windows because later it'll make it easy to use Gulp directly from the Windows command prompt.

Go to nodejs.org/download¹⁴ and download the Windows Installer for your version of windows. (Either 32-bit or 64-bit.)



Node.js Download Page

Install using defaults. Once installed, open up a *new* command prompt and check the installation by looking at the versions installed.

Checking node and npm versions

```
C:\Users\Chuck> node --version  
v0.10.33
```

```
C:\Users\Chuck> npm --version  
1.4.28
```

¹⁴<http://nodejs.org/download>

Installing gulp globally

```
C:\Users\Chuck> npm install -g gulp
C:\Users\Chuck\AppData\Roaming\npm\gulp -> C:\Users\Chuck\AppData\Roaming\npm\node_modules\gulp\bin\gulp.js
gulp@3.8.10 C:\Users\Chuck\AppData\Roaming\npm\node_modules\gulp
[snip]
```

Checking the gulp version

```
C:\Users\Chuck> gulp --version
[10:13:44] CLI version 3.8.10
```



Optionally Install Bower

You can optionally install bower globally if you wish to run bower from a Windows prompt. Personally, I usually run bower within the Homestead Virtual Machine, but it's your choice.

Using the Node package manager (NPM), install bower globally.

Installing bower globally

```
C:\Users\Chuck> npm install -g bower
C:\Users\Chuck\AppData\Roaming\npm\bower -> C:\Users\Chuck\AppData\Roaming\npm\node_modules\bower\bin\bower
bower@1.3.12 C:\Users\Chuck\AppData\Roaming\npm\node_modules\bower
[snip]
```

Checking the bower version

```
C:\Users\Chuck> bower --version
1.3.12
```

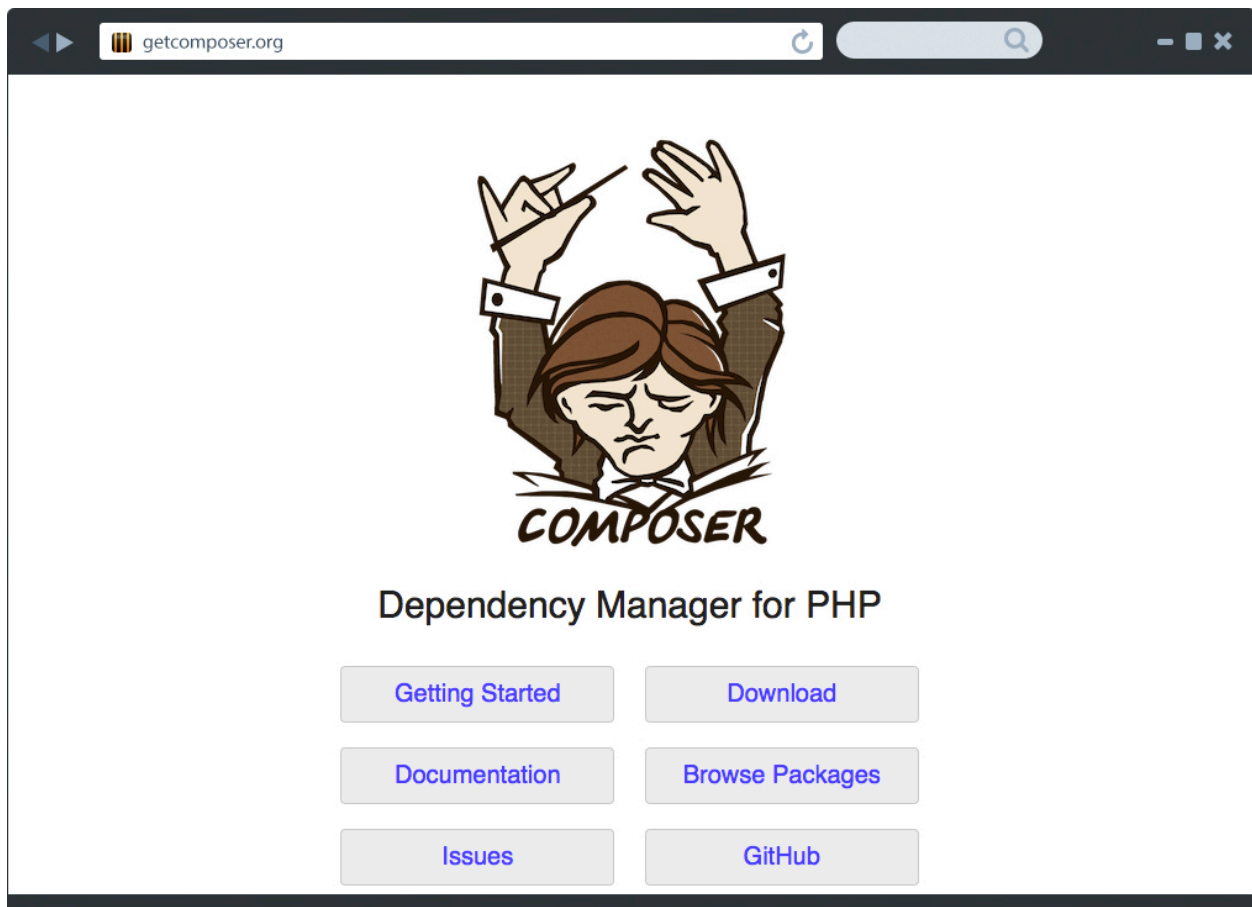


Remember this only installs the programs globally

If you use gulp (or bower) within a particular project you'll still need to install them locally within that project with a `npm install` (omitting the `-g` option). This will be covered later.

Step 3 - Install Composer

Composer is *the* package manager for PHP.



Composer Page

Download and install the Windows setup program, [Composer-Setup.exe](https://getcomposer.org/Composer-Setup.exe)¹⁵. Use the defaults when installing and if it asks you the path to PHP, enter C: \Php\php.exe.

Once Composer installs, close any command prompts and open up a new one. Check the version of composer to see if installed correctly.

Checking the Composer version

```
C:\Users\Chuck> composer --version
Composer version 1.0-dev (b23a3cd36870ff0eefc161a4638d9fcf49d998ba)\
2014-11-21 17:59:11
```

¹⁵<https://getcomposer.org/Composer-Setup.exe>



Installing Composer Updates Your Path

The installation will move C:\Php from your personal path and add it to the system path. It will also add C:\ProgramData\ComposerSetup\bin to your system path.

Step 4 - Install GIT and set up SSH Key

For Windows we'll install the native GIT application and use GIT BASH for setting up the SSH Key. Any other GIT usage in Windows will be through the Windows command prompt.

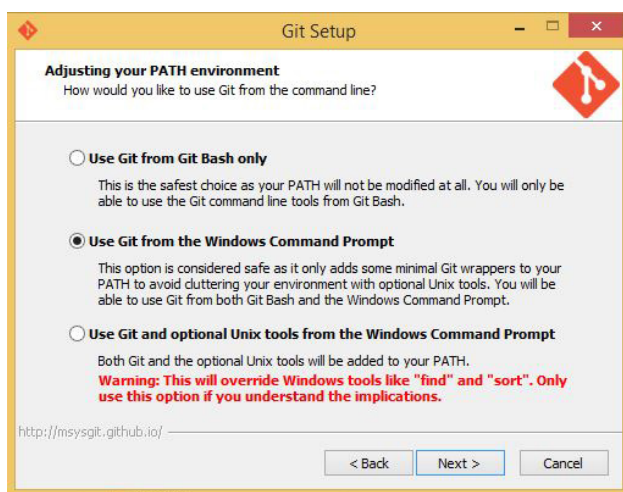
Step 4.1 - Download the git installer

Go to git-scm.com/downloads¹⁶ and click on the [Downloads for Windows] button. This will download the latest version of Git for windows.

(At the time of this writing, the file downloaded is named Git-1.9.4-preview20140920.exe.)

Step 4.2 - Install, choosing the 'Use Git from Command Prompt' option

Run the file just downloaded and choose default options until you get to the following screen.



Git Path Option

Make sure you select the **Use Git from the Windows Command Prompt** option.

Use the defaults for the rest of the installation.

¹⁶<http://git-scm.com/downloads>

Step 4.3 - Checking the Git Version

Close any existing command prompts and open a new command prompt. Make sure Git installed successfully by executing the command below.

Checking the Git Version

```
C:\Users\Chuck> git --version  
git version 1.9.4.msysgit.2
```

Step 4.4 - Setup the SSH Key

Find **Git Bash** in the Windows Start Menu and execute the `ssh-keygen` command below. Press [Enter] all the way through to use the defaults and set up the SSH key with no pass phrase.

Creating SSH Key in Git Bash

```
Chuck@Windows ~  
$ ssh-keygen -t rsa -C "your@email.com"  
Generating public/private rsa key pair.  
Enter file in which to save the key (/c/Users/Chuck/.ssh/id_rsa):  
Created directory '/c/Users/Chuck/.ssh'.  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
$
```

Step 5 - Adding the Homestead box

This step downloads the Laravel Homestead Vagrant box.

Adding the Homestead box in Windows

```
C:\Users\Chuck> vagrant box add laravel/homestead  
==> box: Loading metadata for box 'laravel/homestead'  
    box: URL: https://vagrantcloud.com/laravel/homestead
```

[snip]

It can take a while to download on slow connections.

Step 6. Installing Homestead

Now we'll use composer to install the homestead command. This command line utility makes it easy to control the Homestead VM.

Step 6.1 - Globally requiring Homestead

Globally requiring Homestead 2.0

```
C:\Users\Chuck> composer global require "laravel/homestead=~2.0"
Changed current directory to C:\Users\Chuck\AppData\Roaming\Composer
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing symfony/process (v2.5.7)
  Loading from cache

- Installing symfony/console (v2.5.7)
  Loading from cache

- Installing laravel/homestead (v2.0.7)
  Loading from cache

Writing lock file
Generating autoload files
```

Step 6.2 - Updating Path

Composer just installed Homestead into the vendor directory of your Composer installation. (For example, C:\Users\YOU\AppData\Roaming\Composer).

In order to access homestead from any command prompt, add this path to your User path variable. Follow the same steps to do this as you did back in **Step 1.3 - Adding C:\Php to the path** but this time the path to add will be below (replacing YOU as appropriate).

Paths to add

```
C:\Users\YOU\AppData\Roaming\Composer\vendor\bin;vendor\bin
```



Notice the extra 'vendor\bin' in the path?

We're adding this so any time you're within the base directory of a Laravel project you can easily access any vendor utilities provided in that project. For example, **Phpunit** is installed in the vendor/bin directory of every Laravel application.

Step 6.3 - Verifying Homestead Installed

Close any existing command prompts and open a new command prompt so the latest changes to the path will be in effect. Then check the version of homestead to verify it installed.

Checking the Homestead Version

```
C:\Users\Chuck>homestead --version
Laravel Homestead version 2.0.7
```

Step 6.4 - Initialize Homestead

Once you've installed the homestead command and added the composer bin directory to your path, then you need to initialize Homestead.

Initializing Homestead

```
C:\Users\Chuck> homestead init
Creating Homestead.yaml file...
Homestead.yaml file created at: C:\Users\Chuck\.homestead\Homestead.yaml
```



Remember

You only need to initialize Homestead once on your machine

Step 7 - Bring up the Homestead VM

To bring up Homestead for the first time we'll create a Code directory to store our projects and use the homestead up command.

Bringing up Homestead for the 1st time

```
C:\Users\Chuck> mkdir Code
C:\Users\Chuck> homestead up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'laravel/homestead'...
==> default: Matching MAC address for NAT networking...
==> default Checking if box 'laravel/homestead' is up to date...
```

[snip]

Now the Homestead Virtual Machine is running. If you exit the Windows command prompt, the VM is still running. It'll remain active until you issue a `homestead halt` command from the Windows command prompt.

You can log onto the Homestead Virtual Machine, but on Windows we won't use the `homestead ssh` command, we'll use **PuTTY**.

Step 8 - Setting up PuTTY

Windows does not provide a SSH client so we need to download and install one. For this book we'll use **PuTTY** and set it up to log onto the Homestead VM.

Step 8.1 - Download and install PuTTY

Download [putty-0.63-installer¹⁷](http://the.earth.li/~sgtatham/putty/latest/x86/putty-0.63-installer.exe). Run this file to install PuTTY. You can just use the default installation settings.

Step 8.2 - Convert the SSH Key

Next find PuTTYgen in the Windows Start Menu and run it. Select the **Conversions** menu and then **Import key**. Navigate to the `id_rsa` file created in Step 4.4. Then click the **[Save private key]** button. Yes, you want to save the key without a pass phrase, and save it to the same directory—in my case it's `C:\Users\Chuck\.ssh`—using the filename `id_rsa.ppk`.

Step 8.3 - Setup a Homestead PuTTY Session

Start up PuTTY and set the **Connection | SSH | Auth** private key to the `id_rsa.ppk` just created. Set the Session Hostname to `vagrant@127.0.0.1` and the port to `2222`.

Save the session as the name **homestead**.

The first time you run this session you'll have a confirmation box, but after that you'll log onto the Homestead Virtual Machine without having to type a password.

You may want to create a shortcut on your desktop. The item you want the shortcut to point to is: `"C:\Program Files (x86)\PuTTY\Putty.exe" -load homestead` and name the shortcut **homestead**.



Change PuTTY's Font

The default font PuTTY uses is *Courier New*, which to my eyes is ugly. You can go into **Window | Appearance** in the PuTTY configuration and change the font, size, colors, etc.

¹⁷<http://the.earth.li/~sgtatham/putty/latest/x86/putty-0.63-installer.exe>

Step 8.4 - Connecting to Homestead via PuTTY

Execute the **homestead** session you just created in PuTTY and you should receive a screen similar to the follows.

Homestead's first screen

```
Using username "vagrant".
Authenticating with public key "imported-openssh-key"
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-11-generic x86_64)

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

System information as of Fri Nov 28 04:24:01 UTC 2014

System load:  0.0               Processes:           92
Usage of /:   5.2% of 39.34GB    Users logged in:    0
Memory usage: 33%               IP address for eth0: 10.0.2.15
Swap usage:   0%                IP address for eth1: 192.168.10.10

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

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

Last login: Fri Nov 28 04:24:01 2014 from 10.0.2.2
vagrant@homestead:~$
```

Step 9 - Installing Laravel's Installer

For the last step we'll install the Laravel installer

Globally requiring Laravel Installer

```
C:\Users\Chuck> composer global require "laravel/installer=~1.1"
Changed current directory to C:\Users\Chuck\AppData\Roaming\Composer
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing guzzlehttp/streams (2.1.0)
  Downloading: 100%

- Installing guzzlehttp/guzzle (4.2.3)
  Downloading: 100%

- Installing laravel/installer (v1.1.3)
  Downloading: 100%

Writing lock file
Generating autoload files
```

Since your path was already updated in Step 6.2 to contain composer's bin directory, the `laravel` command should already be accessible from a DOS Prompt. Verify it by checking the version.

Checking the Laravel Version

```
C:\Users\Chuck> laravel --version
Laravel Installer version 1.1
```



Congratulations!

You now have a virtual Ubuntu 64-bit machine, ready for developing your Laravel 5.1 web applications.

Recap

This chapter was basically a laundry list of steps to follow in order to get Laravel Homestead up and running on your Windows machine. The good news is, these steps only have to be performed once. Now, skip to the **Using Homestead** chapter for some information about Laravel Homestead.

Chapter 4 - Setting up an OS X or Linux Machine

This chapter goes through the steps required to set up and install the supporting software for Laravel Homestead on an OS X or Linux machine. It is assumed **VirtualBox** and **Vagrant** were already installed from the **Required Software and Components** chapter.

Chapter Contents

- [Slight Variations with Linux](#)
- [Step 1 - Installing PHP](#)
- [Step 2 - Install Node.js](#)
- [Step 3 - Install Bower and Gulp](#)
- [Step 4 - Install Composer](#)
- [Step 5 - Adding SSH Keys](#)
- [Step 6 - Adding the Homestead box](#)
- [Step 7 - Installing Homestead](#)
- [Step 8 - Bring up the Homestead VM](#)
- [Step 9 - Installing the Laravel Installer](#)
- [Recap](#)

Slight Variations with Linux

There are slight variations between the different Linux distributions. In particular, the package manager. CentOS and Fedora use **yum** as the package manager, Ubuntu uses **apt**. There is no official “package manager” with OS X other than the App Store, but **homebrew** is the unofficial OS X package manager. Regardless of the differences, the essence is pretty much the same across all *nix systems, including OS X.

Step 1 - Installing PHP

Often PHP will be pre-installed on your system. You can check the version from a terminal window.

Checking the PHP version

```
~> php --version
PHP 5.5.9-1ubuntu4.5 (cli) (built: Oct 29 2014 11:59:10)
Copyright (c) 1997-2014 The PHP Group
Zend Engine v2.5.0, Copyright (c) 1998-2014 Zend Technologies
    with Zend OPcache v7.0.3, Copyright (c) 1999-2014, by Zend Technologies
```

Laravel 5.1 requires PHP version 5.5.9 or above. If you don't have PHP installed, or it's not at least version 5.5.9, then you'll need to use your package manager to install PHP.



OS X Yosemite

Yosemite (the version of OS X at the time of this writing) ships with PHP 5.5.14. So no worries there.

Example installing PHP in Ubuntu

```
~> sudo apt-get install php5
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  php5
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
```

[snip]

Step 2 - Install Node.js

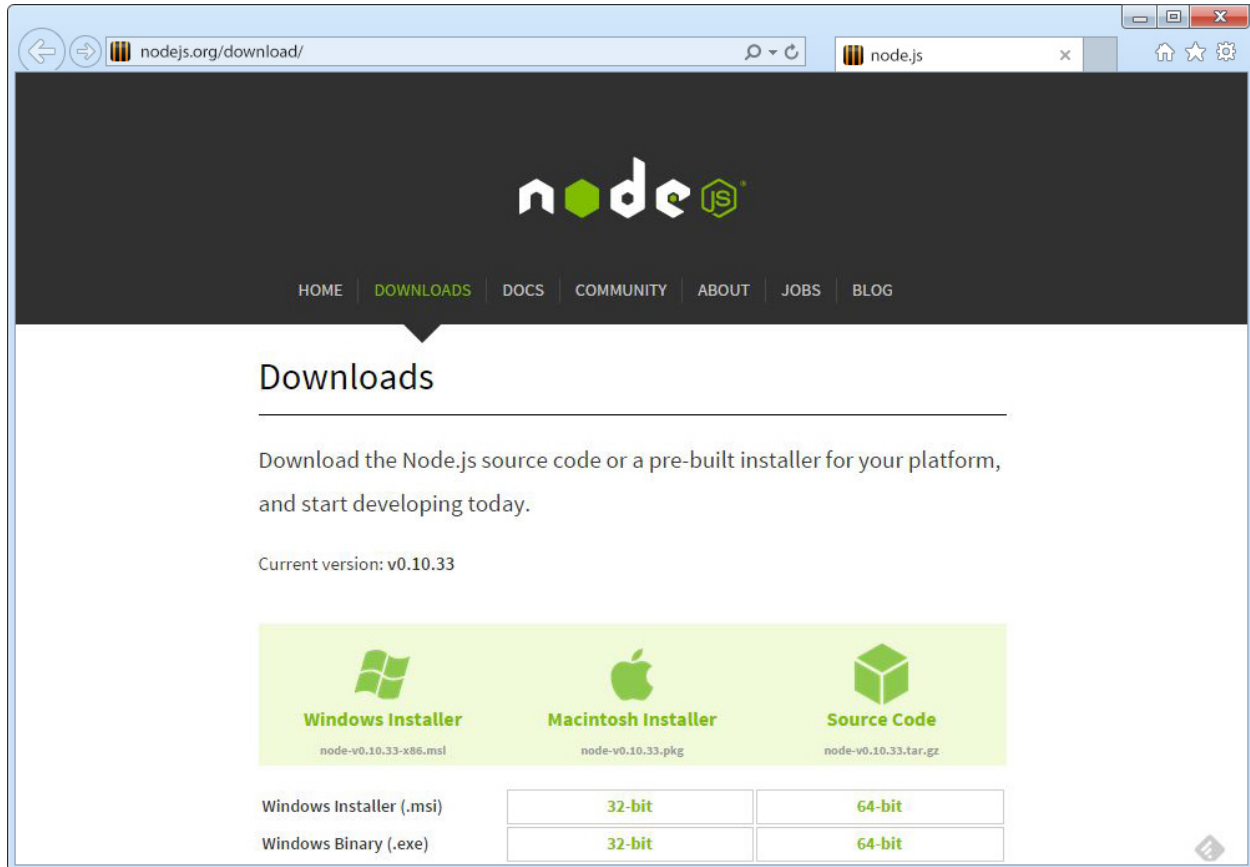
You'll need Node.js installed to later use Gulp.

Often, Node.js is already installed. You can check the version of **npm** to see if Node.js is installed on your system.

Checking the npm version

```
$> npm --version
1.5.0-alpha-4
```

If it's not installed, there are a couple options for installing it. You can use your package manager to install it. With OS X you can install it with Homebrew. Or you can just go to nodejs.org/download¹⁸ and download the correct version for your operating system.



Node.js Download Page

Once installed, be sure to check the version to make sure node and npm are available in your path.

Checking node and npm versions

```
~> node --version  
v0.10.29
```

```
~> npm --version  
1.5.0-alpha-4
```

¹⁸<http://nodejs.org/download>

Step 3 - Install Gulp

Gulp is an integral part of rapid Laravel development. Use the Node package manager (NPM) to install gulp globally.

Installing gulp globally

```
~> npm install -g gulp
/usr/local/bin/gulp -> /usr/local/lib/node_modules/gulp/bin/gulp.js
gulp@3.8.10 /usr/local/lib/node_modules/gulp
[snip]
```

Checking the gulp version

```
~> gulp --version
[10:13:44] CLI version 3.8.10
```



Optionally Install Bower

You can optionally install bower globally if you wish to run bower from your Linux (or OS X) console. Personally, I use bower both from my OS X console and within the Homestead Virtual Machine, whichever I'm currently in.

Use the Node package manager (NPM), install bower globally.

Installing bower globally

```
~> npm install -g bower
/usr/local/bin/bower -> /usr/local/lib/node_modules/bower/bin/bower
bower@1.3.12 /usr/local/lib/node_modules/bower
[snip]
```

Checking the bower version

```
~> bower --version
1.3.12
```



Remember this only installs the programs globally

If you use gulp (or bower) within a particular project you'll still need to install them locally within that project with a `npm install` (omitting the `-g` option). This will be covered later.

Step 4 - Install Composer

Composer is the package manager for PHP. It can easily be installed from a terminal window in *nix systems (including both OS X and Linux). An alternative method of installing Composer in OS X using Homebrew is presented at the bottom of this section.

Installing Composer

```
~> curl -sS https://getcomposer.org/installer | php
#!/usr/bin/env php
All settings correct for using Composer
Downloading...
```

Composer successfully installed to: /Users/chuck/composer.phar
Use it: php composer.phar

Once you have composer.phar downloaded, move it to the global path.

Moving composer.phar

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

And then check the version to make sure it's accessible.

Checking the Composer version

```
~> composer --version
Composer version 1.0-dev (b23a3cd36870ff0eefc161a4638d9fcf49d998ba)\
2014-11-21 17:59:11
```



Install using Homebrew

In OS X, if you are using Homebrew, you can install composer using the instructions below.

Alternative installation in OS X with Homebrew

```
~> brew update
~> brew tap homebrew/dupes
~> brew tap homebrew/php
~> brew install composer
```

Step 5 - Adding SSH Keys

If you haven't already added a SSH key for your machine, you'll need to do it.

Checking for SSH Keys

```
~> ls ~/.ssh
config id_rsa id_rsa.pub
```

If you don't see `id_rsa` and `id_rsa.pub` create them with the following command. (Press [Enter] all the way through to use the defaults and set up the SSH key with no pass phrase.)

Creating SSH Keys

```
~> ssh-keygen -t rsa -C "your@email.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/Chuck/.ssh/id_rsa):
Created directory '/Users/Chuck/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

Step 6 - Adding the Homestead box

This step downloads the Laravel Homestead Vagrant box.

Adding the Homestead box in Linux

```
~> vagrant box add laravel/homestead
==> box: Loading metadata for box 'laravel/homestead'
    box: URL: https://vagrantcloud.com/laravel/homestead
```

[snip]

It can take a while to download on slow connections.

Step 7 - Installing Homestead

Now we'll use composer to install the homestead command. The homestead command line utility makes it easy to control the Homestead Virtual Machine.

Step 7.1 - Globally requiring Homestead

Globally requiring Homestead 2.0

```
~> composer global require "laravel/homestead=~2.0"
Changed current directory to /home/chuck/.composer
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing symfony/process (v2.5.7)
    Loading from cache

  - Installing symfony/console (v2.5.7)
    Loading from cache

  - Installing laravel/homestead (v2.0.7)
    Loading from cache

Writing lock file
Generating autoload files
```

Step 7.2 - Updating Path

Composer just installed Homestead into the vendor directory of your Composer installation. (For example /home/chuck/.composer in Linux or /Users/Chuck/.composer in OS X).

In order to access homestead from any command prompt, add this path to your path variable. This should be added to whatever the startup script is for your operating system. Common startup files are: .bashrc, .bash_profile, .zshrc, etc.

At the bottom of your startup script add the following line:

Updating path in the startup script

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

**Notice the extra 'vendor/bin' in the path?**

We're adding this so any time you're within the base directory of a Laravel project you can easily access any vendor utilities provided in that project such as **phpunit**.

Step 7.3 - Verifying Homestead Installed

Close any existing terminal windows and open a new terminal window so the latest changes to the path will be in effect. Then check the version of homestead to verify it installed.

Checking the Homestead Version

```
~>homestead --version
Laravel Homestead version 2.0.7
```

Step 7.4 - Initialize Homestead

Once you've installed the homestead command and added the composer bin directory to your path, then you need to initialize Homestead.

Initializing Homestead

```
~> homestead init
Creating Homestead.yaml file...
Homestead.yaml file created at: /home/chuck/.homestead/Homestead.yaml
```

**Remember**

You only need to initialize Homestead once on your machine

Step 8 - Bring up the Homestead VM

To bring up Homestead for the first time we'll create a Code directory to store our projects and use the homestead up command.

Bringing up Homestead for the 1st time

```
~> mkdir Code
~> homestead up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'laravel/homestead'...
==> default: Matching MAC address for NAT networking...
==> default Checking if box 'laravel/homestead' is up to date...
```

[snip]

Now the Homestead VM is running. If you exit the terminal window, Homestead is still running. It'll remain active until you issue a `homestead halt` command from a terminal window.

Now you can log onto homestead with the `homestead ssh` command.

Shelling to homestead

```
~> homestead ssh
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-11-generic x86_64)

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

System information as of Fri Nov 28 04:24:01 UTC 2014

System load:  0.0               Processes:            92
Usage of /:   5.2% of 39.34GB    Users logged in:     0
Memory usage: 33%              IP address for eth0: 10.0.2.15
Swap usage:   0%               IP address for eth1: 192.168.10.10

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

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

Last login: Fri Nov 28 04:24:01 2014 from 10.0.2.2
vagrant@homestead:~$
```

Step 9 - Installing the Laravel Installer

For the last step we'll install the Laravel installer. Do the following from your console (not from within the Homestead Virtual Machine).

Globally requiring Laravel Installer

```
~> composer global require "laravel/installer=~1.1"
Changed current directory to /Users/chuck/.composer
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing guzzlehttp/streams (2.1.0)
  Downloading: 100%

- Installing guzzlehttp/guzzle (4.2.3)
  Downloading: 100%

- Installing laravel/installer (v1.1.3)
  Downloading: 100%

Writing lock file
Generating autoload files
```

Since your path was already updated in Step 7.2 to contain composer's bin directory, the `laravel` command should already be accessible from the Console window. Verify it by checking the version.

Checking the Laravel Version

```
~> laravel --version
Laravel Installer version 1.2.1
```



Congratulations!

You now have a virtual Ubuntu 64-bit machine, ready for developing your Laravel 5.1 web applications.

Recap

This chapter contained a series of steps to follow in order to get Laravel Homestead up and running on your OS X or Linux machine. The good news is, these steps only have to be performed once.

The next chapter, **Using Homestead**, contains information about using Homestead.

Chapter 5 - Homestead and Laravel Installer

This chapter explores the two composer tools previously installed: **homestead** and **laravel**. A typical daily workflow is examined, as are the six steps to set up any new Laravel 5.1 project.

Chapter Contents

- The Homestead Tool
- Overview of Common Homestead Commands
- Examining Homestead.yaml
- Adding Software to the Homestead VM
- Daily Workflow
- Six Steps to Starting a New Laravel 5.1 Project
 - Step 1 - Create the app skeleton
 - Step 2 - Configure the web server
 - Step 3 - Add the Host to Your Hosts File
 - Step 4 - NPM Local Installs
 - Step 5 - Create the app's database
 - Step 6 - Testing in the Browser
- Other Homestead Tips
- Recap

The Homestead Tool



Console Defined

Whenever you are prompted to do something from the console, context is important.

The **homestead console** means connecting to the Homestead VM via SSH. For Windows, this means using PuTTY (*explained in the chapter on setting up a Windows machine*). With other operating systems you can execute the `homestead ssh` command from within the terminal. Whenever you see the `$` prompt in this book you are in the homestead console.

The **OS console** means either the Windows command prompt or the terminal application you use. (The `%` prompt in the book is used for your OS specific console.)

From the *console* of your host operating system, you can easily see what the valid homestead commands are by typing the `homestead` command without any arguments.

Homestead Commands

```
% homestead
```

```
Laravel Homestead version 2.0.9
```

Usage:

```
[options] command [arguments]
```

Options:

```
--help           -h Display this help message.
--quiet          -q Do not output any message.
--verbose        -v|vv|vvv Increase the verbosity of messages: 1 for normal \
output, 2 for more verbose output and 3 for debug.
--version        -V Display this application version.
--ansi           Force ANSI output.
--no-ansi        Disable ANSI output.
--no-interaction -n Do not ask any interactive question.
```

Available commands:

```
destroy    Destroy the Homestead machine
edit       Edit the Homestead.yaml file
halt       Halt the Homestead machine
help       Displays help for a command
init       Create a stub Homestead.yaml file
list       Lists commands
provision   Re-provisions the Homestead machine
resume     Resume the suspended Homestead machine
run        Run commands through the Homestead machine via SSH
ssh        Login to the Homestead machine via SSH
status     Get the status of the Homestead machine
suspend    Suspend the Homestead machine
up         Start the Homestead machine
update     Update the Homestead machine image
```

The main command you'll use each day is the `homestead up` command to start the Homestead Virtual Machine.

Overview of Common Homestead Commands

Here's a quick overview of commonly used Homestead commands.

homestead up

Starts the Homestead Virtual Machine. It turns on the power to the VM. If you use the provision option (`homestead up --provision`) then any new sites you've added will be provisioned.

homestead halt

Stops the Homestead Virtual Machine. In other words, powering off.

homestead suspend

Suspends the Homestead Virtual Machine. It's like hibernate.

homestead resume

Resumes the suspended Homestead Virtual Machine.

homestead edit

Edit the Homestead.yaml file. This launches whatever editor is associated with YAML files on your operating system.

homestead status

See the current status of the Homestead Virtual Machine.

Examining Homestead.yaml

The configuration settings for Laravel Homestead are contained in the Homestead.yaml file. This file is located in the `.homestead` directory of your Host OS's home directory.

If you view the contents of this file, you'll see what's below.

Contents of Homestead.yaml

```
---
ip: "192.168.10.10"
memory: 2048
cpus: 1

authorize: ~/.ssh/id_rsa.pub

keys:
  - ~/.ssh/id_rsa

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

sites:
```

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

databases:
- homestead

variables:
- key: APP_ENV
  value: local
```

Here's a definition of each of the settings.

ip The internal IP used to access the machine.

memory

How much memory the VM will use.

cpus The number of CPUs the VM will use.

authorize

This should point to your public SSH key.

keys Your private SSH key.

folders

The shared folders. These are the directories in your Host Operating System and where they will appear within the VM. For Windows the ~/Code equates to something like C:\Users\YOU\Code. In OS X, this is /Users/YOU/Code. Under Linux it's usually something like /home/YOU/Code. Whenever you edit a file in this directory tree on your host machine, it's instantly available to the Homestead Virtual Machine.

sites A list of sites (paths each domain points to) that will be set up on the Homestead Virtual Machine each time you provision.

databases

A list of database Homestead should automatically create.

variables

Variables to make available to the homestead environment.



A configuration note

The only change I usually make to the configuration is to change the list of databases to have one database named `xhomestead` instead of `homestead`. This way if I forget create an app's database when creating a new Laravel application, an error occurs. *(Otherwise, since the default database for a new application is `homestead`, no error will occur and I'll be using the `homestead` db without realizing it.)*

For now, don't change any homestead configuration values except the **databases** setting (and then, only if you want to.)

The Homestead Virtual Machine Details

What	Value
Hostname	homestead
IP Address	192.168.10.10
Username	vagrant
SU Password	vagrant
Database Host	127.0.0.1
Database Port	33060
Database Username	homestead
Database Password	secret

Adding Software to the Homestead VM

When you need to install new software inside the Homestead Virtual Machine, use the Ubuntu utility `apt-get`.

It's an easy two step process.

1. Upgrade Ubuntu
2. Install with `apt-get`

For example, here's how to install **unzip**, a handy utility for dealing with zip archives.

First, Upgrade Ubuntu

Upgrading Latest Ubuntu Software

```
vagrant@homestead:~$ sudo apt-get update
vagrant@homestead:~$ sudo apt-get upgrade
```

You may have to choose "Y" to continue. If prompted during the installation to pick a configuration it's generally best to go with the existing or default.

After the Ubuntu OS within the Homestead VM is updated, install **unzip**.

Next, Install unzip with apt-get

Installing unzip in the Homestead VM

```
vagrant@homestead:~$ sudo apt-get install unzip
```

Daily Workflow

The daily workflow when working with homestead consists of three steps:

Step 1 - homestead up - Start the day by booting your Homestead Virtual Machine.

Step 2 - homestead ssh or PuTTY - SSH to the Homestead VM to access files directly and execute artisan commands.

Step 3 - write beautiful code - In your favorite code editor, on your host operating system, write code.

Optional 4th Step - homestead halt - When you are done for the day, you can optionally power off the Homestead Virtual Machine with the halt command.

Six Steps to Starting a New Laravel 5.1 Project

There are six simple steps to follow whenever starting a new Laravel 5.1 application.

Let's say we want to create a project called **test.app** and use **test** as the project folder.

Step 1 - Create the app skeleton

Using the *Laravel Installer* (the `laravel` command installed in a previous chapter) it's easy to create a new project skeleton.

Creating a new app skeleton

```
~/Code % laravel new test
Crafting application...
Generating optimized class loader
Compiling common classes
Application key [rzUhyDksVxzTXFjzFYiOWToqpunI2m6X] set successfully.
Application ready! Build something amazing.
```

Step 2 - Configure the web server

After there's an application skeleton in place you can set up the Nginx webserver within the homestead environment to serve pages from your app's public directory.

The homestead environment makes this easy with the `serve` command.

Setting up a new virtual host in Homestead

```
~/Code$ serve test.app ~/Code/test/public
dos2unix: converting file /vagrant/scripts/serve.sh to Unix format ...
* Restarting nginx nginx [ OK ]
php5-fpm stop/waiting
php5-fpm start/running, process 2169
```

The `serve` command sets up a new configuration file in `/etc/nginx/sites-available` for the domain we'll be using (**test.app**) and a symbolic link to this file within the `/etc/nginx/sites-enabled` directory.

Even when you reboot the machine, this configuration file will be there.



Why not edit Homestead.yaml

Yes. Another alternative is to set up the parameters for the **test.app** virtual host using the `homestead edit` command and adding a new entry to the `sites` section. But this is easier, and there's no need to continuously re-provision the Homestead VM.

But, if you're setting up an app you always want configured, it's not a bad idea to edit `Homestead.yaml` and set up the configuration there.

Step 3 - Add the Host to Your Hosts File

Since **test.app** does not exist in any DNS, an entry must be added to the Host OS's hosts file. Edit `/etc/hosts` in Linux or OS X. In Windows the file is `C:\Windows\System32\drivers\etc\hosts`. In this hosts file point **test.app** to the IP specified in `Homestead.yaml`.

Add the following line to this file.

Host entry for **test.app**

```
192.168.10.10 test.app
```



Windows requires admin privileges to edit hosts

In Windows you must launch your editor (such as Notepad, Wordpad, or Sublime Text) as an administrator. In Linux or OS X you can use the `sudo` command.

Editing hosts with Linux or OS X

```
sudo nano /etc/hosts
// or
sudo vi /etc/hosts
```

Step 4 - NPM Local Installs

In order to later use **gulp** it's important to make sure all the required npm modules are locally installed.

*You can skip this step if you know you will not use **gulp**.*

Change to your project directory in your Host OS's console and execute the following.

NPM Local Installs

```
~% cd Code/test
~/Code/test% npm install
npm WARN package.json @ No repository field.

> v8flags@1.0.5 install /Users/chuck/Code/test/node_modules/gulp/\
node_modules/v8flags
> node fetch.js

flags for v8 3.14.5.9 cached.

[snip]
```

This will install everything required by **gulp** locally into the `node_modules` directory of your project.

Step 5 - Create the app's database

If your application requires a database, it's easy to create it within the Homestead VM using the **mysql** console.

Creating a Database in the Homestead VM

```
$ mysql --user=homestead --password=secret
mysql> create database test;
mysql> exit;
```

Once the database is created, edit the `.env` file in your project's root directory and change the `DB_NAME` appropriately.

Change DB_NAME in .env

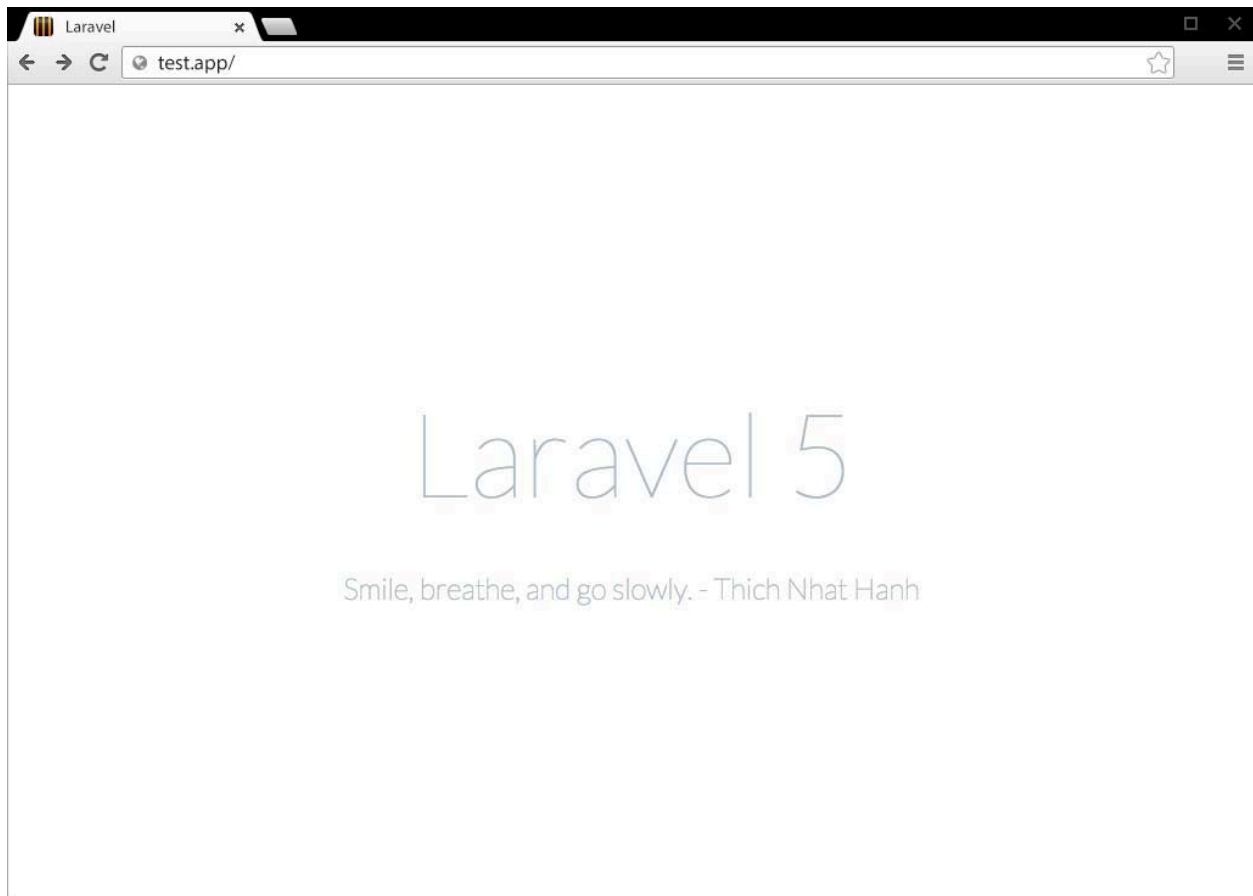
```
// Change the following line
DB_DATABASE=homestead

// To the correct value
DB_DATABASE=test
```

Easy. Now you'll be able to migrate and create tables. This is covered in a later chapter.

Step 6 - Testing in the Browser

Point your browser to `http://test.app` and you should see the page below.



Default Laravel Page

If you see anything else then something didn't work.

Other Homestead Tips

Edit Source Code in your Host Operating System

Although this has been mentioned in an earlier chapter, it bears repeating. Always edit your source code in your Host OS. Through the magic of shared folders, changes you make within the `~/Code` directory are immediately seen within the Homestead Virtual Machine.

Use the `.homestead/aliases` file

Each time you re-provision Homestead with `homestead up --provision` or `homestead provision`, the `.homestead/aliases` file updates the aliases in the Homestead Virtual Machine.

This is a handy place to add aliases, or functions, or even other environment variables.

Keep the Homestead VM up-to-date

As mentioned earlier, two commands will keep the Ubuntu operating system within the Homestead Virtual Machine up to date.

Keeping Ubuntu Updated

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

Recap

This chapter provided details on the `homestead` and `laravel` commands. And the *Six Steps to a New Laravel 5.1 Project* were outlined.

In the next chapter where we'll do a bit of testing.

Chapter 6 - Testing

In this chapter we'll create a project to use throughout the rest of the book and explore various options for testing. We'll create a class to convert Markdown formatted files to HTML. This class will be created using TDD principles.

Chapter Contents

- [Creating the l5beauty Project](#)
- [Running PHPUnit](#)
 - [Laravel 5.1's PHPUnit Configuration](#)
 - [Laravel 5.1 Crawler Methods and Properties](#)
 - [Laravel 5.1 PHPUnit Application methods and properties](#)
 - [Laravel 5.1 PHPUnit Assertions](#)
- [Using Gulp for TDD](#)
- [Creating a Markdown Service](#)
 - [Pulling in Markdown Packages](#)
 - [Creating the Markdown Test Class](#)
 - [Creating the Markdowner Service](#)
 - [A Few More Tests](#)
- [Other Ways Test](#)
 - [phpspec](#)
 - [Unit Testing](#)
 - [Functional / Acceptance Testing](#)
 - [Behavior Driven Development](#)
- [Recap](#)

Creating the l5beauty Project

Follow the [Six Steps to Starting a New Laravel 5.1 Project](#) below to create the *l5beauty* project.

First, from your Host OS, install the app skeleton.

Step 1 - Install the app skeleton

```
~/Code % laravel new l5beauty
Crafting application...
Generating optimized class loader
Compiling common classes
Application key [rzUhyDksVxzTXFjzFYiOWToqpunI2m6X] set successfully.
Application ready! Build something amazing.
```

Next, from within the Homestead VM, set up `l5beauty.app` as the virtual host.

Step 2 - Configure the web server

```
~/Code$ serve l5beauty.app ~/Code/l5beauty/public
dos2unix: converting file /vagrant/scripts/serve.sh to Unix format ...
* Restarting nginx nginx [ OK ]
php5-fpm stop/waiting
php5-fpm start/running, process 2169
```

Back in your Host OS, add the following line to your hosts file.

Step 3 - Add l5beauty.app to Your Hosts File

```
192.168.10.10 l5beauty.app
```

From your Host OS, do the step to install the NPM packages locally.

Step 4 - NPM Local Installs

```
~% cd Code/l5beauty
~/Code/l5beauty% npm install
|
> node-sass@2.0.1 install /Users/chuck/Code/l5beauty/node_modules/laravel-\
    elixir/node_modules/gulp-sass/node_modules/node-sass
> node scripts/install.js

> node-sass@2.0.1 postinstall /Users/chuck/Code/l5beauty/node_modules/\
    laravel-elixir/node_modules/gulp-sass/node_modules/node-sass
> node scripts/build.js

`darwin-x64-node-0.10` exists; testing
Binary is fine; exiting
```

```
gulp@3.8.11 node_modules/gulp
├─ v8flags@2.0.2
├─ pretty-hrtime@0.2.2
```

```
[snip]
```

Go back within the Homestead VM and create the database for this project.

Step 5 - Create the app's database

```
$ mysql --user=homestead --password=secret
mysql> create database l5beauty;
Query OK, 1 row affected (0.00 sec)

mysql> exit;
Bye
```

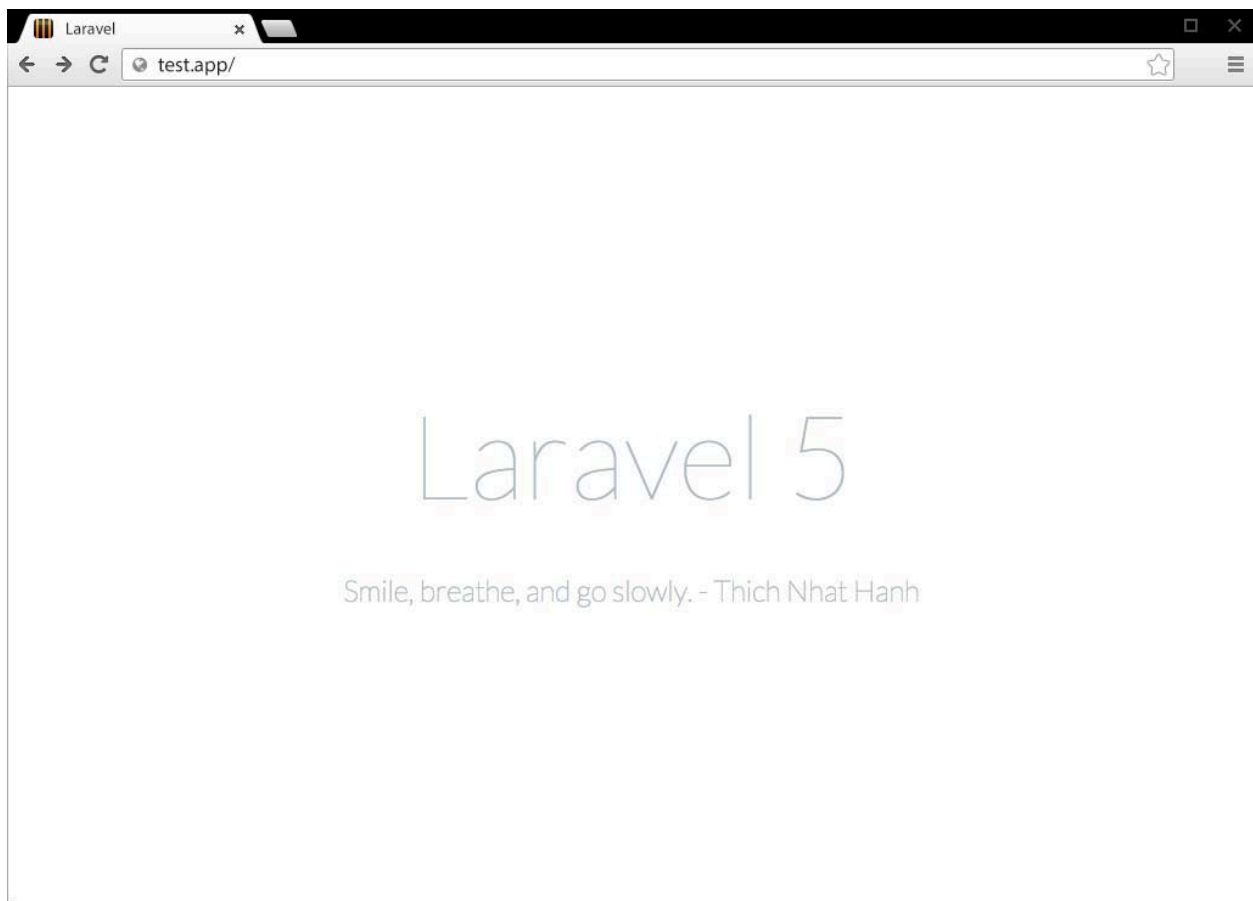
Then edit the `.env` file, changing the database to `l5beauty`.

Changing DB_NAME in configuration

```
// Change the following line
DB_DATABASE=homestead

// To the correct value
DB_DATABASE=l5beauty
```

Finally, bring up `http://l5beauty.app` in your browser to make sure everything is working correctly.



Default Laravel Page

Running PHPUnit

Laravel 5.1 comes out of the box ready for testing. There's even a very simple unit test supplied to make sure a web request to the application returns the expected 200 HTTP response.

To run PHPUnit, execute the `phpunit` command from the project's root directory.

Running PHPUnit

```
~% cd Code/15beauty
~/Code/15beauty% phpunit
PHPUnit 4.7.4 by Sebastian Bergmann and contributors.

.

Time: 544 ms, Memory: 10.25Mb

OK (1 test, 2 assertions)
```



Did you get an error?

If you receive a **command not found** or a **permissions denied** error when attempting to run the `phpunit` command it could be because of an installation bug. The `phpunit` command should be found in the `vendor/bin` directory—and this directory was added to the path in your Host OS back in Chapter 3 or 4. The problem is that the Laravel Installer has a bug that doesn't necessarily set the permissions correctly on `phpunit` and several other utilities.

To fix this bug, follow the two steps below.

Step 1 - Delete the `vendor` directory. Just wipe it out using whatever command is appropriate for your Host OS.

Step 2 - Recreate the `vendor` directory using the `composer update` command from your project's root directory. Do this from your Host Operating System.

That's it. Then try executing the `phpunit` command again.

Laravel 5.1's PHPUnit Configuration

In the root of each Laravel 5.1 project is the file `phpunit.xml`. This contains the configuration PHPUnit uses when `phpunit` is executed from the project's root directory.

Examination of the `phpunit.xml` will show the tests reside within the `tests` directory. There are two files located there.

1. `ExampleTest.php` - Contains one test `testBasicExample()`. The `ExampleTest` class is derived from the `TestCase` parent provided in the other file.
2. `TestCase.php` - The base class from which to derive Laravel tests.

Take a look at the `testBasicExample()` method in `ExampleTest.php`.

The `testBasicExample()` method

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

This test says “Visit the home page and we should see the words ‘Laravel 5’.” Can tests get any simpler than this?

The `TestCase` class provides additional Laravel 5.1 specific application methods and properties to your unit tests. `TestCase` also provides a long list of additional assertion methods and *crawler* type tests.

Laravel 5.1 Crawler Methods and Properties

The *Crawler* tests allow you to test pages in your web application. The nice thing is that many of these tests are fluent and return `$this`, allowing you to build the `->visit()->see()` type test in the above example.

Here are some of the available properties and methods.

`$response`

The last response returned by the web application.

`$currentUri`

The current URL being viewed.

`visit($uri)`

(Fluent) Visit the given URI with a GET request.

`get($uri, array $headers = [])`

(Fluent) Fetch the given URI with a GET request, optionally passing headers.

`post($uri, array $data = [], array $headers = [])`

(Fluent) Make a POST request to the specified URI.

`put($uri, array $data = [], array $headers = [])`

(Fluent) Make a PUT request to the specified URI.

`patch($uri, array $data = [], array $headers = [])`

(Fluent) Make a PATCH request to the specified URI.

`delete($uri, array $data = [], array $headers = [])`

(Fluent) Make a DELETE request to the specified URI.

`followRedirects()`

(Fluent) Follow any redirects from latest response.

`see($text, $negate = false)`

(Fluent) Assert the given text appears (or doesn't appear) on the page.

seeJson(array \$data = null)

(Fluent) Assert the response contains JSON. If `$data` passed, also asserts the JSON value exactly matches.

seeStatusCode(\$status)

(Fluent) Assert the response has the expected status code.

seePageIs(\$uri)

(Fluent) Assert current page matches given URI.

seeOnPage(\$uri) and landOn(\$uri)

(Fluent) Aliases to `seePageIs()`

click(\$name)

(Fluent) Click on a link with the given body, name or id.

type(\$text, \$element)

(Fluent) Fill an input field with the given text.

check(\$element)

(Fluent) Check a checkbox on the page.

select(\$option, \$element)

(Fluent) Select an option from a dropdown.

attach(\$absolutePath, \$element)

(Fluent) Attach a file to a form field.

press(\$buttonText)

(Fluent) Submit a form using the button with the given text.

withoutMiddleware()

(Fluent) Disable middleware for the test.

dump()

Dump the content of the latest response.

Laravel 5.1 PHPUnit Application methods and properties

Here's a brief rundown of some of the additional application methods and properties Laravel 5.1 provides to PHPUnit.

\$app The instance of the Laravel 5.1 application.

\$code

The latest code returned by artisan

refreshApplication()

Refreshes the application. Automatically called by the TestCase's setup() method.

call(\$method, \$uri, \$parameters = [], \$cookies = [], \$files = [], \$server = [], \$content = null)

Calls the given URI and returns the response.

callSecure(\$method, \$uri, \$parameters = [], \$cookies = [], \$files = [], \$server = [], \$content = null)

Calls the given HTTPS URI and returns the response.

action(\$method, \$action, \$wildcards = [], \$parameters = [], \$cookies = [], \$files = [], \$server = [], \$content = null)

Calls a controller action and returns the response.

route(\$method, \$name, \$routeParameters = [], \$parameters = [], \$cookies = [], \$files = [], \$server = [], \$content = null)

Calls a named route and returns the response.

instance(\$abstract, \$object)

Register an instance of an object in the container.

expectsEvents(\$events)

Specify a list of events that should be fired for the given operation.

withoutEvents()

Mock the event dispatcher so all events are silenced.

expectsJobs(\$jobs)

Specify a list of jobs that should be dispatched for the given operation.

withSession(array \$data)

Set the session to the given array.

session(array \$data)

Starts session and sets the session values from the array.

flushSession()

Flushes the contents of the current session.

startSession()

Starts the application's session.

actingAs(\$user)

(Fluent) Sets the currently logged in user for the application.

be(\$user)

Sets the currently logged in user for the application.

seeInDatabase(\$table, array \$data, \$connection = null)

(Fluent) Asserts a given where condition exists in the database.

notSeeInDatabase(\$table, \$array \$data, \$connection = null)

(Fluent) Asserts a given where condition does not exist in the database.

missingFromDatabase(\$table, array \$data, \$connection = null)

(Fluent) Alias to `notSeeInDatabase()`.

seed()

Seeds the database.

artisan(\$command, \$parameters = [])

Executes the artisan command and returns the code.

Any of these methods or properties can be accessed within your test classes. The provided `ExampleTest.php` file contains a line using `$this->call(...)` inside the `testBasicExample()` method.

Laravel 5.1 PHPUnit Assertions

In addition to the standard PHPUnit assertions (such as `assertEquals()`, `assertContains()`, `assertInstanceOf()`, ...), Laravel 5.1 provides many additional assertions to help write tests dealing with the web application.

assertPageLoaded(\$uri, \$message = null)

Assert the latest page loaded; throw exception with `$uri/$message` if not.

assertResponseOk()

Assert that the client response has an OK status code.

assertReponseStatus(\$code)

Assert that the client response has a given code.

assertViewHas(\$key, \$value = null)

Assert that the response view has a given piece of bound data.

assertViewHasAll(\$bindings)

Assert that the view has a given list of bound data.

assertViewMissing(\$key)

Assert that the response view is missing a piece of bound data.

`assertRedirectedTo($uri, $with = [])`

Assert whether the client was redirected to a given URI.

`assertRedirectedToRoute($name, $parameters = [], $with = [])`

Assert whether the client was redirected to a given route.

`assertRedirectedToAction($name, $parameters = [], $with = [])`

Assert whether the client was redirected to a given action.

`assertSessionHas($key, $value = null)`

Assert that the session has given key(s)/value(s).

`assertSessionHasAll($bindings)`

Assert that the session has a given list of values.

`assertSessionHasErrors($bindings = [])`

Assert that the session has errors bound.

`assertHasOldInput()`

Assert that the session has old input.

Using Gulp for TDD

Gulp¹⁹ is a build and automation system written in JavaScript. It allows common tasks such as minification of source files to be automated. Gulp can even watch your source code for changes and automatically run tasks when this occurs.

Laravel 5.1 includes **Laravel Elixir**²⁰ which allows Gulp tasks to be built in easy ways. Elixir adds an elegant syntax to gulp. Think of it this way ... what Laravel is to PHP, Elixir is to Gulp.

One of the most common uses of Gulp is to automate unit tests. We'll follow the TDD (Test Driven Development) process here and let Gulp automatically run our tests.

First, edit the `gulpfile.js` file in the 15beauty project's root directory to match what's below.

Configuring Gulp to run PHPUnit Tests

```
var elixir = require('laravel-elixir');
```

```
elixir(function(mix) {  
    mix.phpUnit();  
});
```

¹⁹<http://gulpjs.com/>

²⁰<http://laravel.com/docs/5.1/elixir>

Here we call the `elixir()` function, passing a function. The `mix` object this function receives is a stream on which multiple things can occur. You might want to build LESS files into CSS files here, then concatenate those CSS files together, and then provide versioning on the resulting concatenated files. All of those things can be specified by using a fluent interface on the `mix` object.

But for now, we're only running PHPUnit tests.

Next, from the project root on your Host OS, run `gulp` to see what happens.

Running Gulp

```
~% cd Code/15beauty
~/Code/15beauty% gulp
[15:26:23] Using gulpfile ~/Code/15beauty/gulpfile.js
[15:26:23] Starting 'default'...
[15:26:23] Starting 'phpunit'...
[15:26:25] Finished 'default' after 2.15 s
[15:26:25]

    *** Debug Cmd: ./vendor/bin/phpunit --colors --debug ***

[15:26:28] PHPUnit 4.7.4 by Sebastian Bergmann and contributors.

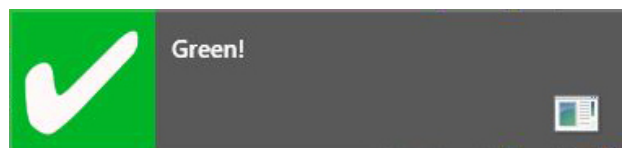
Configuration read from /Users/chuck/Code/15beauty/phpunit.xml

Starting test 'ExampleTest::testBasicExample'.
.

Time: 2.07 seconds, Memory: 10.25Mb

OK (1 test, 2 assertions)
[15:26:28] gulp-notify: [Green!]
[15:26:28] Finished 'phpunit' after 4.96 s
```

You should have received a notification, a popup alert of some sort, on your Host OS. The notification should be green which indicates everything tested successfully.



PHPUnit Success

To have `gulp` go into automatic mode for unit tests, use the `gulp tdd` command in your Host OS.

Running Gulp

```
~% cd Code/15beauty
~/Code/15beauty% gulp tdd
[15:29:49] Using gulpfile ~/Code/15beauty/gulpfile.js
[15:29:49] Starting 'tdd'...
[15:29:49] Finished 'tdd' after 21 ms
```

The command will just *hang* there, watching for source file changes and running unit tests when needed.

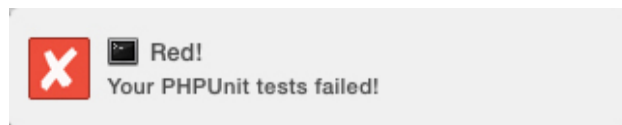
To see how this works, let's break the existing unit test.

Change the `see()` line in `tests/ExampleTest.php` to what's below.

Breaking ExampleTest.php

```
1 ->see('Laravel 5x');
```

When you save this file, gulp will notice and run PHPUnit again. The will fail and you will see a notice on your computer similar to the one below.



PHPUnit Failure

Change the line back to what it was before, save it, and again gulp will run PHPUnit. This time you should receive a notice indicating you are "*back to green*".



To exit Gulp's *tdd* mode

Press `Ctrl+C`

Creating a Markdown Service

The blogging application we'll be building will allow editing posts in Markdown format. Markdown is an easy-to-read *and* easy-to-write format that transforms easily to HTML.

To illustrate testing, we'll build a service to convert markdown text to HTML text using TDD.

Pulling in Markdown Packages

There are many PHP packages out there for converting Markdown to HTML. If you go to <http://packagist.org> and search for *markdown*, there are twenty pages of packages.

We'll use the package created by Michel Fortin because there's another package called *SmartyPants* by the same author that converts quotation marks to the nice looking curly quotes.

From your Host OS's console do the following to pull in the packages.

Adding Markdown and SmartyPants

```
~/Code/15beauty% composer require michelf/php-markdown
Using version ^1.5 for michelf/php-markdown
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing michelf/php-markdown (1.5.0)
    Downloading: 100%

Writing lock file
Generating autoload files
Generating optimized class loader

~/Code/15beauty% composer require "michelf/php-smartypants=1.6.0-beta1"
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing michelf/php-smartypants (1.6.0-beta1)
    Loading from cache

Writing lock file
Generating autoload files
Generating optimized class loader
```

Did you notice that the specific version of the package was specified when requiring *SmartyPants*? This is because at the time of this writing there isn't a stable package that can be pulled in automatically.

Creating the Markdown Test Class

The first thing to do when starting a TDD session is to fire up Gulp in TDD mode.

Starting Gulp in TDD mode

```
~/Code/15beauty% gulp tdd
[19:41:38] Using gulpfile ~/Code/15beauty/gulpfile.js
[19:41:38] Starting 'tdd'...
[19:41:38] Finished 'tdd' after 23 ms
```

Now that Gulp is watching for changes and ready to run PHPUnit as soon as it detects any, let's create the test class.

In the tests directory, create a new folder named `Services` and a file called `MarkdownerTest.php`.

Initial tests/Services/MarkdownerTest.php

```
1 <?php
2
3 class MarkdownerTest extends TestCase
4 {
5
6     protected $markdown;
7
8     public function setup()
9     {
10         $this->markdown = new \App\Services\Markdowner();
11     }
12
13     public function testSimpleParagraph()
14     {
15         $this->assertEquals(
16             "<p>test</p>\n",
17             $this->markdown->toHTML('test')
18         );
19     }
20 }
```

Line 6

Store an instance of the markdown object

Line 8

Have the `setup()` method create a new instance of the `Markdowner` class. (Yes, this doesn't exist yet.)

Line 13

A simple test we know should work.

You should have received a failure notice. *(If you didn't Ctrl+C out of Gulp and restart it.)*

Even though a notice appeared saying the test failed, sometimes it's useful to look at the console to determine what the failure was. In this case, it's pretty obvious. The App\Services\Markdowner class doesn't exist.

Creating the Markdowner Service

What we'll do here is create a simple service that wraps the php-markdown and php-smartypants packages we imported earlier.

In the app\Services directory create a Markdowner.php file with the following contents.

Contents of app/Services/Markdowner.

```
1 <?php
2
3 namespace App\Services;
4
5 use Michelf\MarkdownExtra;
6 use Michelf\SmartyPants;
7
8 class Markdowner
9 {
10
11     public function toHTML($text)
12     {
13         $text = $this->preTransformText($text);
14         $text = MarkdownExtra::defaultTransform($text);
15         $text = SmartyPants::defaultTransform($text);
16         $text = $this->postTransformText($text);
17         return $text;
18     }
19
20     protected function preTransformText($text)
21     {
22         return $text;
23     }
24
25     protected function postTransformText($text)
26     {
27         return $text;
28     }
29 }
```

Line 3

Don't forget the namespace.

Lines 5 and 6

The classes we'll be using.

Line 11

The toHTML() method which runs the text through the transformations.

Line 14

Notice we're using the Markdown Extra version of the library.

Line 20

In case we want to later do our own transformations before anything else.

Line 25

Like `preTransformText()`, but this time if we later want to add our own final transformations.

When you save this file, Gulp should notice and you will receive a "GREEN" alert telling you everything worked as expected.

If you don't receive the green alert, go back and check for typos in both the `App\Services\Markdowner` and `MarkdownerTest` classes.

A Few More Tests

Admittedly, this isn't a great example of TDD because it's simple a test and a complete class created to fix the test. In actual practice TDD would have many more iterations, resulting in a flow like the one below:

- Create `MarkdownerTest` w/ `testSimpleParagraph()`
- Tests Fail
- Create `Markdowner` class, hard-coding `toHTML()` to pass the test
- Tests Succeed
- Update `Markdowner` class to use `MarkdownExtra`
- Tests Succeed
- Add a `testQuotes()` to `MarkdownerTest` class
- Tests Fail
- Update `Markdowner` class to use `SmartyPants`
- Tests Succeed

And so forth. Even the structure of our `Markdowner` class is flawed when it comes to testing. To do *pure* unit testing on this class it should be structured such that instances of both the `MarkdownExtra` and `SmartyPants` classes are injected into the constructor. This way our unit test could inject mock objects and only verify the behavior of `MarkdownExtra` and not the subordinate classes it calls.

But this isn't a book on testing. In fact, this is the only chapter where testing occurs.

For now, we'll leave the structure as is but add a few more tests.

Update `MarkdownerTest` to match what's below.

Final Contents of `app/Services/Markdowner`.

```

1  <?php
2
3  class MarkdownerTest extends TestCase
4  {
5
6      protected $markdown;
7
8      public function setup()
9      {
10         $this->markdown = new \App\Services\Markdowner();
11     }
12
13     /**
14      * @dataProvider conversionsProvider
15      */
16     public function testConversions($value, $expected)
17     {
18         $this->assertEquals($expected, $this->markdown->toHTML($value));
19     }
20
21     public function conversionsProvider()
22     {
23         return [
24             ["test", "<p>test</p>\n"],
25             ["# title", "<h1>title</h1>\n"],
26             ["Here's Johnny!", "<p>Here&#8217;s Johnny!</p>\n"],
27         ];
28     }
29 }
```

Here we changed the test class to test multiple conversions at once and added three tests in `conversionsProvider()`. Your tests should be green before moving forward.

Once the tests are green hit `Ctrl+C` in your Host OS console to stop Gulp.

Other Ways to Test

It's not the intent here to provide a definitive list of all the ways to test with Laravel 5.1 because there's really no single way to do testing in PHP. Therefore, there's no single way to test in Laravel 5.

But, we'll explore some alternatives.

phpspec

Besides PHPUnit, Laravel 5.1 also provides [phpspec](#)²¹ out of the box. This is another popular PHP test suit with more of a focus on Behavior Driven Development.

Here's a few notes on phpspec.

- The binary is in `vendor/bin`, thus you can call `phpspec` from your project's root directory.
- The configuration file is in the project root. It's named `phpspec.yml`.
- To run phpspec from Gulp, Laravel Elixir provides the `phpSpec()` function you can call on the `mix` object.
- If you change your application's namespace from `App` to something else, be sure to update `phpspec.yml` accordingly.

Unit Testing

Although PHPUnit is the standard when it comes to PHP unit testing, there are other packages you can use.

- [Enhance PHP](#)²² - A unit testing framework with support for mocks and stubs.
- [SimpleTest](#)²³ - Another unit testing framework with mock objects.

Integration and Acceptance Testing

These tests actually use your application instead of just verifying that units of code within your application work as expected. When using the fluent test methods Laravel 5.1 provides you can do some integration tests using PHPUnit. `ExampleTest.php` shows a simple example. But there are other testing frameworks that focus on integration and acceptance testing.

²¹<http://phpspec.net>

²²<https://github.com/Enhance-PHP/Enhance-PHP>

²³<http://simpletest.org>

- [Codeception](#)²⁴ - Problem the most popular framework for acceptance testing.
- [Selenium](#)²⁵ - Browser automation.
- [Mink](#)²⁶ - Brower automation.

Behavior Driven Development

BDD comes in two flavors: SpecBDD and StoryBDD.

SpecDD focuses on the technical aspects of your code. Laravel 5.1 includes *phpspec* which is the standard for SpecDD.

StoryBDD emphasizes business or feature testing. Behat is the most popular StoryBDD framework. Although, Codeception can also be used for StoryBDD.

Recap

The first thing we did in this chapter was creating a project named **l5beauty**. Then we explored unit testing using PHPUnit within this project. Finally, we created a Markdowner service class for the dual purposes of having something to test and to use later to convert markdown text to HTML.

This was a pretty long chapter because testing is a large topic and a single chapter cannot give it justice. But, as I've mentioned, testing is not the focus of this book. There will be no more testing in subsequent chapters.

How about something quicker? In the next chapter we'll create a blog in 10 minutes.

²⁴<http://codeception.com>

²⁵<http://seleniumhq.org>

²⁶<http://mink.behat.org>

Chapter 7 - The 10 Minute Blog

In this chapter we'll turn the **l5beauty** project into a blog, complete with test data. Harnessing the power of Laravel 5.1 a blog can be created in less than 10 minutes. This time is from start to finish, without spending time reviewing the detailed discussions below. There's not many bells or whistles, and no administration of the blog, but what do you expect for less than 10 minutes of development time.

Chapter Contents

- [Pre-work before the 10 Minute Blog](#)
- 0:00 to 2:30 - Creating the Posts table
- 2:30 to 5:00 - Seeding Posts with test data
- 5:00 to 5:30 - Creating configuration
- 5:30 to 7:30 - Creating the routes and controller
- 7:30 to 10:00 - Creating the views
- [Recap](#)

Pre-work before the 10 Minute Blog

We've already completed most of the pre-work with the **l5beauty** project created in the last chapter.

All that is needed to create a blog in 10 minutes is a freshly created Laravel 5.1 project with the database created. No need for migrations to have been run, although it won't hurt anything if they have been.

We'll use the 10 Minute Blog as the starting point for a project which will grow into a full-featured blogging application. This application won't use the password reset feature of Laravel 5.1 so delete the password resets migration as illustrated below.

Deleting the password resets migration

```
~/Code/l5beauty$ rm database/migrations/2014_10_12_100000_create_\  
password_resets_table.php
```



Don't Forget the Line Continuation

The above code is actually typed in as one line. That backslash at the end of the first line is the line continuation character.

0:00 to 2:30 - Creating the Posts table

First, create a Post model within the Homestead VM.

Creating Post model

```
~/Code/15beauty$ php artisan make:model --migration Post
Model created successfully.
Created Migration: 2015_03_22_210207_create_posts_table
```

This does two things:

1. It creates the model class `App\Post` class in the `app` directory of the project.
2. It creates a migration for the posts table. This migration will end up in the `database/migrations` folder.



What is a Migration?

Think of migrations as version control for your database structure. Check out the documentation at laravel.com. Migrations and seeding are explained well there.

Edit the migration file just created in the `database/migrations` directory to match what's below.

The `create_posts_table` migration

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreatePostsTable extends Migration
{
    /**
     * Run the migrations.
     */
    public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->increments('id');
            $table->string('slug')->unique();
            $table->string('title');
```

```

        $table->text('content');
        $table->timestamps();
        $table->timestamp('published_at')->index();
    });
}

/**
 * Reverse the migrations.
 */
public function down()
{
    Schema::drop('posts');
}
}

```

We've added four additional columns here from the default ones provided when the migration was created.

slug We'll convert the title of each post to a slug value and use that as part of the URI to the post. This helps make the blog search engine friendly.

title Every post needs a title.

content
Every post needs some content.

published_at
We want to control when the post is to be published.

Now that the migration is edited, run migrations to create the table.

Running the migrations

```

~/Code/15beauty$ php artisan migrate
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2015_03_22_210207_create_posts_table

```

Finally, edit `app/Post.php` to match what's below.

First version of the Post model

```
1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Post extends Model
8 {
9     protected $dates = ['published_at'];
10
11     public function setTitleAttribute($value)
12     {
13         $this->attributes['title'] = $value;
14
15         if (! $this->exists) {
16             $this->attributes['slug'] = str_slug($value);
17         }
18     }
19 }
```

Line 9

Tell Laravel 5.1 that the `published_at` column should be treated as a date.

Line 11

Whenever a value is assigned to the `title` property of this object, the `setTitleAttribute()` method will be called to do it. We assign the property as normal and if the row has not yet been saved to the database then we convert the title to a slug value and assign the slug.

2:30 to 5:00 - Seeding Posts with test data

Now that there's a place to store posts for our blog, let's create some random data. To do this we'll make use of Laravel 5.1's **Model Factories** which is built on the excellent *Faker* library by Francois Zaninotto.

Add the following lines to the `ModelFactory.php` file located in the `database/factories` directory.

The Post Model Factory

```

1 $factory->define(App\Post::class, function ($faker) {
2     return [
3         'title' => $faker->sentence(mt_rand(3, 10)),
4         'content' => join("\n\n", $faker->paragraphs(mt_rand(3, 6))),
5         'published_at' => $faker->dateTimeBetween('-1 month', '+3 days'),
6     ];
7 });

```

Line 1

We're defining the factory for the AppPost class. The function gets an instance of a Faker object.

Line 2

The factory function should return an array of column keys and values.

Line 3

For the title, we'll use a random sentence between 3 and 10 words long.

Line 4

For the content, we'll use between 3 and 6 random paragraphs.

Line 5

And for published_at, we'll use a random time between a month ago and 3 days from now.

Next edit the DatabaseSeeder.php file located in the database/seeds folder to match what's below.

Contents of DatabaseSeeder.php

```

1 <?php
2
3 use Illuminate\Database\Seeder;
4 use Illuminate\Database\Eloquent\Model;
5
6 class DatabaseSeeder extends Seeder
7 {
8     /**
9      * Run the database seeds.
10     */
11     public function run()
12     {
13         Model::unguard();
14

```

```
15     $this->call('PostTableSeeder');
16 }
17
18 }
19
20 class PostTableSeeder extends Seeder
21 {
22     public function run()
23     {
24         App\Post::truncate();
25
26         factory(App\Post::class, 20)->create();
27     }
28 }
```

Line 15

Call the seeder for the posts table

Line 20

Yeah, stuffing another class into this file. Sometimes rules are made to be broken. Although, this will probably change in the future if we need to add more seeds.

Line 24

Truncate any existing records in the posts table.

Line 26

Call the Post model factory, 20 times, creating the resulting rows in the database.

Finally, to get the random data into the database, use artisan to seed it.

Seeding the database

```
~/Code/15beauty$ php artisan db:seed
Seeded: PostTableSeeder
```

Once `artisan db:seed` is executed there will be 20 rows of data in the posts database. You can use your favorite database client to view it, but I'm not going to here because we're on a tight schedule.

5:00 to 5:30 - Creating configuration

We may want to have some configuration options with our blog. Things like the title and number of posts per page. So let's quickly set that up.

Create a new file in the `config` directory called `blog.php` with the contents below.

Contents of config/blog.php

```
1 <?php
2 return [
3     'title' => 'My Blog',
4     'posts_per_page' => 5
5 ];
```

Within our Laravel 5.1 application it'll be easy to refer to these settings using the `config()` helper function. For instance, `config('blog.title')` will return the title. You could also return these values using the Config facade, for example `Config::get('blog.title')`.



The Timezone

This is a good time to update the `config/app.php` file if you wish to change your app's timezone from UTC to your local timezone.

5:30 to 7:30 - Creating the routes and controller

Next edit the `app/Http/routes.php` file, to match what's below.

The Initial Routes

```
1 <?php
2
3 get('/', function () {
4     return redirect('/blog');
5 });
6
7 get('blog', 'BlogController@index');
8 get('blog/{slug}', 'BlogController@showPost');
```

Line 3

When a GET request to `http://15beauty.app/` is made, we return a redirect to the user, redirecting them to `http://15beauty.app/blog`.

Line 7

When a GET request to `http://15beauty.app/blog` is made, the `index()` method of the `BlogController` class will be called and the result returned to the user. (*The full class name of the controller is `App\Http\Controllers\BlogController`.*)

Line 8

When a GET request to `http://15beauty.app/blog/ANYTHING-HERE` is made, then the `showPost()` method of `BlogController` is called. The value **ANYTHING-HERE** will be passed to the `showPost()` method as an argument.

Next we'll create `BlogController`.

First, use `artisan` to create an empty controller.

Creating an empty BlogController

```
~/Code/15beauty$ php artisan make:controller BlogController --plain
Controller created successfully.
```

(The `--plain` option is used to create an empty class without the standard RESTful methods.)

A new file named `BlogController.php` will be created in the `app/Http/Controllers` directory. Edit it to match what's below.

Initial version of BlogController.php

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use App\Post;
6 use Carbon\Carbon;
7
8 class BlogController extends Controller
9 {
10     public function index()
11     {
12         $posts = Post::where('published_at', '<=', Carbon::now())
13             ->orderBy('published_at', 'desc')
14             ->paginate(config('blog.posts_per_page'));
15
16         return view('blog.index', compact('posts'));
17     }
18
19     public function showPost($slug)
20     {
21         $post = Post::whereSlug($slug)->firstOrFail();
22
23         return view('blog.post')->withPost($post);
```



```

24     }
25 }

```

Line 12

Start an Eloquent query where we want all posts not scheduled in the future.

Line 13

We'll order the posts with the most recently published first.

Line 14

And we'll tap into Eloquent's powerful pagination feature to only return the maximum number of posts per page we've set up in the configuration.

Line 16

Return the `blog\index.blade.php` view (yet to be created), passing it the `$posts` variable.

Line 21

Fetch a single post from the slug and throw an exception if it's not found.

Line 23

The `->withPost()` method is another way to pass variables to the view. In this case we're passing the `$post` variable. It'll be available to the view as `$post`.

You can check the application's routing table with the following artisan command.

Showing the routes

```
~/Code/15beauty$ php artisan route:list
```

Method	URI	Action
GET HEAD	/	Closure
GET HEAD	blog	App\Http\Controllers\BlogController@index
GET HEAD	blog/{slug}	App\Http\Controllers\BlogController@showPost

(Empty columns omitted in above table.)

7:30 to 10:00 - Creating the views

All that's left to do is to create two views, one to show the index and one to show a post.

Create a new directory named `blog` in the `resources/views` directory.

Then create a new file, `index.blade.php`. Since this file ends with `.blade.php` it is a Blade template, which is Laravel 5's built in template engine. Make `index.blade.php` match the content below.

Content of blog.index View

```

1 <html>
2 <head>
3   <title>{{ config('blog.title') }}</title>
4   <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css"
5       rel="stylesheet">
6 </head>
7 <body>
8   <div class="container">
9     <h1>{{ config('blog.title') }}</h1>
10    <h5>Page {{ $posts->currentPage() }} of {{ $posts->lastPage() }}</h5>
11    <hr>
12    <ul>
13      @foreach ($posts as $post)
14        <li>
15          <a href="/blog/{{ $post->slug }}">{{ $post->title }}</a>
16          <em>({{ $post->published_at->format('M jS Y g:ia') }})</em>
17          <p>
18            {{ str_limit($post->content) }}
19          </p>
20        </li>
21      @endforeach
22    </ul>
23    <hr>
24    {!! $posts->render() !!}
25  </div>
26 </body>
27 </html>

```

Line 3

Here we use two curly braces to surround the output of whatever value is assigned in the blog config file. Note that two curly braces will escape the output, making it safe for HTML.

Line 4

We'll use bootstrap to provide just a touch of styling

Line 10

The `$posts` variable is actually a pagination object which basically means it's a collection but also has methods for determining the page metrics.

Line 13

Here's the blade syntax for a foreach loop

Line 16

Since we specified `published_at` is a date column, we can use the `format()` method here to show the date just how we want it.

Line 24

Two interesting things about this line. First, notice how we use the a single curly brace with double exclamation marks before and after a call to the `render()` method. Unlike the double curly brace syntax, this does not escape the output. The second thing is the `render()` method itself. We use this to display next and previous links.

The last step in the 10 minute blog is to create the view to display posts. Create a new file named `post.blade.php` in the `resources/views/blog` directory and match its contents to what's below.

Content of `blog.post View`

```

1 <html>
2 <head>
3   <title>{{ $post->title }}</title>
4   <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css"
5         rel="stylesheet">
6 </head>
7 <body>
8   <div class="container">
9     <h1>{{ $post->title }}</h1>
10    <h5>{{ $post->published_at->format('M jS Y g:ia') }}</h5>
11    <hr>
12    {!! nl2br(e($post->content)) !!}
13    <hr>
14    <button class="btn btn-primary" onclick="history.go(-1)">
15      &laquo; Back
16    </button>
17  </div>
18 </body>
19 </html>

```

Nothing to explain here because everything should be clear.

**Congratulations**

Point your browser to `http://15beauty.app` and browse around a bit.

You have created a simple and clean blog in just a few minutes. Isn't the power of Laravel 5.1 amazing?

Does this really only take 10 minutes?

Yes. I timed myself, working slowly and methodically, copying and pasting much of the code. It took a total of 7 minutes and 8 seconds.

Recap

In this chapter we created a working blog with test data in less than 10 minutes.

‘Nuf said.

Chapter 8 - Starting the Admin Area

In this chapter we'll continue building on the `l5beauty` project and start developing the administration area.

Chapter Contents

- Establishing the Routes
 - Middleware
 - Resource Controllers
- Creating the Admin Controllers
- Creating the Views
 - Creating an Admin Layout
 - Creating the Navbar Partial
 - Creating the Login Form
 - Creating the Errors Partial
 - Creating the Post Index View
- Testing logging in and out
 - Creating the admin user
 - Fixing log out location
 - Fix default login location
 - Logging in and out
- Cleaning up a couple unneeded files
- Recap

Establishing the Routes

In the last chapter, when creating the blog in 10 minutes, the `app/Http/routes.php` file was set up for the blog. Now, we'll add some additional routes for administration.



Why Routes?

Laravel 5.1 needs a way to tie web requests to the code that handles the web request. This is called routing. All routing within this book is defined in the `app/Http/routes.php` file.

Whenever a web request is made to a file that doesn't exist within the `public` directory, Laravel 5.1 looks at the routes file to determine what to return. For instance, a web request to `/css/app.css` will be handled by the web server (assuming `public/css/app.css` exists), but a request to `/blog/my-welcome-page` doesn't exist in the `public` directory and Laravel 5.1 tries to find a matching route to execute.

Update the app/Http/routes.php to match what's below.

Adding Admin Routes

```
1 <?php
2
3 // Blog pages
4 get('/', function () {
5     return redirect('/blog');
6 });
7 get('blog', 'BlogController@index');
8 get('blog/{slug}', 'BlogController@showPost');
9
10 // Admin area
11 get('admin', function () {
12     return redirect('/admin/post');
13 });
14 $router->group([
15     'namespace' => 'Admin',
16     'middleware' => 'auth',
17 ], function () {
18     resource('admin/post', 'PostController');
19     resource('admin/tag', 'TagController');
20     get('admin/upload', 'UploadController@index');
21 });
22
23 // Logging in and out
24 get('/auth/login', 'Auth\AuthController@login');
25 post('/auth/login', 'Auth\AuthController@postLogin');
26 get('/auth/logout', 'Auth\AuthController@logout');
```

Line 11 - 13

Redirect requests to /admin to the /admin/post page

Line 14 - 16

Start a routing group using the namespace Admin (which expands out to the App\Http\Controllers\Admin namespace) and force the auth middleware to be active. (See the section below on Middleware).

Line 18 - 19

Within the route group, add two **Resource Controllers** (see below).

Line 20

Add a route so whenever a GET request is made to /admin/upload the index() method of App\Http\Controllers\Admin\UploadController will be called.

Line 24 - 26

Here we add routes for logging in and logging out.

Once you save the `routes.php` file, the next step will be to create any missing controllers. But before we get to that, let's explore a couple concepts: *Middleware* and *Resource Controllers*.

**Tip - Read the Docs**

Check out the Laravel 5.1 documentation at laravel.com for more information about routing.

Middleware

If you used Laravel 4 you may recall the concept of filters. The middleware in Laravel 5.1 provides the functionality that filters did in Laravel 4, but they are more aptly named.

Abstract Flow of Request to Response

```
HTTP request received
-> Check for maintenance mode*
    -> Start session*
        -> Get response from controller action
    -> Encrypt Cookies*
-> Add cookies to response*
Return response to user
```

In this simplified flow from a request to the response, the items with the asterisk(*) at the end are considered middleware.

The file `app/Http/Kernel.php` contains a list of the middleware for your application. When viewing this file note the `$middleware` property contains the global middleware (that is, middleware always executed) and the `$routeMiddleware` property contains a list of middleware that can be applied at the route level

Auth Middleware

Let's say a route has the **auth** middleware active, the flow would look similar to what's below.

Request to Response with auth middleware

HTTP request received

-> Global "before" middleware

-> (auth) If not logged on then return redirect to logon form

-> Get response from controller action

-> Global "after" middleware

Return response to user

Thus, if the **auth** middleware detects the user is not logged on, the controller action is never executed. Instead, the user is redirected to the logon page.



More About Middleware

The best place to learn more about Laravel 5.1 Middleware is the official Laravel 5.1 documentation.

Resource Controllers

Earlier, in our routing file, we specified a resourceful route using the `resource()` function. This single declaration creates multiple routes, names those routes, and points them to a series of expected action methods on the controller.

For example, the `resource('admin/post', 'PostController')` statement sets up all the routes in the table below.

HTTP Verb	Path	Action Method	Route Name
GET	admin/post	index()	admin.post.index
GET	admin/post/create	create()	admin.post.create
POST	admin/post	store()	admin.post.store
GET	admin/post/{post}	show()	admin.post.show
GET	admin/post/{post}/edit	edit()	admin.post.edit
PUT/PATCH	admin/post/{post}	update()	admin.post.update
DELETE	admin/post/{post}	destroy()	admin.post.destroy

Creating the Admin Controllers

Now that the routes are set up for the administration area, use artisan to create the controllers.

Creating the admin controllers with artisan

```
~/Code/15beauty$ php artisan make:controller Admin\\PostController
Controller created successfully.
```

```
~/Code/15beauty$ php artisan make:controller Admin\\TagController
Controller created successfully.
```

```
~/Code/15beauty$ php artisan make:controller Admin\\UploadController --plain
Controller created successfully.
```

Upon completion of the above three artisan commands, there will be three new controller files in the `app/Http/Controllers/Admin` directory.

NOTE: The `--plain` option was only used on the upload controller. The `PostController.php` and `TagController.php` files will be created with all the resource methods stubbed out.



Try the artisan `route:list` command

If you use the `artisan route:list` command now from the Homestead VM you'll see all the routes and all the actions and controllers they map to.

Update the `index()` method within the `PostController` class to match what's below.

PostController's `index()` method

```
1  /**
2   * Display a listing of the posts.
3   *
4   * @return Response
5   */
6  public function index()
7  {
8      return view('admin.post.index');
9  }
```

The `index()` method simply returns the view. We'll build it shortly.

Creating the Views

There's a few views we need to create. Let's just run through them one-by-one.

Creating an Admin Layout

The Blade template engine is one of the most powerful features of Laravel. We'll set up a layout to use for our blog administration which will give the administration area a consistent look.

Create a directory named `resources/views/admin` and within this directory create a `layout.blade.php` file with the following content.

Content of `admin.layout` view

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1">
7
8      <title>{{ config('blog.title') }} Admin</title>
9
10     <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css"
11         rel="stylesheet">
12     @yield('styles')
13
14     <!--[if lt IE 9]>
15         <script src="//oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
16         <script src="//oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
17     <![endif]-->
18 </head>
19 <body>
20
21 {{-- Navigation Bar --}}
22 <nav class="navbar navbar-default">
23     <div class="container-fluid">
24         <div class="navbar-header">
25             <button type="button" class="navbar-toggle collapsed"
26                 data-toggle="collapse" data-target="#navbar-menu">
27                 <span class="sr-only">Toggle Navigation</span>
28                 <span class="icon-bar"></span>
29                 <span class="icon-bar"></span>
30                 <span class="icon-bar"></span>
31             </button>
32             <a class="navbar-brand" href="#">{{ config('blog.title') }} Admin</a>
33         </div>
34         <div class="collapse navbar-collapse" id="navbar-menu">
```

```

35     @include('admin.partials.navbar')
36     </div>
37 </div>
38 </nav>
39
40 @yield('content')
41
42 <script
43 src="//ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
44 <script
45 src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.min.js"></script>
46
47 @yield('scripts')
48
49 </body>
50 </html>

```

This snippet of code may look familiar. It's the basic template for the Bootstrap CSS Framework. There're just a few additional hooks in it.

```
<title>{{ config('blog.title') }} Admin</title>
```

Here we set the title of the page to the blog's title with the word "Admin" added on.

```
@yield('styles')
```

This Blade directive will output the section (if there is one) named **styles**. It allows us to put some extra CSS at the top of the template on a page-by-page basis.

```
@include('admin.partials.navbar')
```

Here we're including another blade template (which does not yet exist).

```
@yield('content')
```

This will output the main content of the page.

```
@yield('scripts')
```

Here's where additional JavaScript can be output.



Escaped or Unescaped?

Blade templates provide two ways to output PHP expressions. You can wrap the PHP code in double curly braces `{{ 'like this' }}` and the value of the PHP expression will be output at that point in the template, but the output will be escaped—meaning HTML entities will be encoded. If you want the value to not be escaped, then wrap the expression in a curly brace and double exclamation `{!! 'like this' !!}`.

Creating the Navbar Partial

This view is the one the layout includes.

Create the new directory `resources/admin/partials` and within that directory create a `navbar.blade.php` file with the following content.

Initial Content of `admin.partials.navbar` View

```

1 <ul class="nav navbar-nav">
2   <li><a href="/">Blog Home</a></li>
3   @if (Auth::check())
4     <li @if (Request::is('admin/post*')) class="active" @endif>
5       <a href="/admin/post">Posts</a>
6     </li>
7     <li @if (Request::is('admin/tag*')) class="active" @endif>
8       <a href="/admin/tag">Tags</a>
9     </li>
10    <li @if (Request::is('admin/upload*')) class="active" @endif>
11      <a href="/admin/upload">Uploads</a>
12    </li>
13  @endif
14 </ul>
15
16 <ul class="nav navbar-nav navbar-right">
17   @if (Auth::guest())
18     <li><a href="/auth/login">Login</a></li>
19   @else
20     <li class="dropdown">
21       <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button"
22         aria-expanded="false">{{ Auth::user()->name }}
23       <span class="caret"></span>
24     </a>
25     <ul class="dropdown-menu" role="menu">
26       <li><a href="/auth/logout">Logout</a></li>
27     </ul>
28   </li>
29   @endif
30 </ul>

```

If a user is logged in, then this template displays a menu for **Posts**, **Tags**, and **Uploads** on the left and a **Logout** on the right.

If there is no user logged in then only a **Login** link is displayed on the right.


```
36         <input type="checkbox" name="remember"> Remember Me
37     </label>
38 </div>
39 </div>
40 </div>
41
42     <div class="form-group">
43         <div class="col-md-6 col-md-offset-4">
44             <button type="submit" class="btn btn-primary">Login</button>
45         </div>
46     </div>
47 </form>
48 </div>
49 </div>
50 </div>
51 </div>
52 </div>
53 @endsection
```

Just a couple things to note with this form.

1. We included a not-yet-created `admin.partials.errors`. This will be created next.
2. The `old()` function used to output the value of the email field will contain the value entered if this form generates an error and is displayed again.

Creating the Errors Partial

Checking for input errors and displaying those errors is such a common task when dealing with forms that we're breaking it out into its own tiny Blade template.

Create a new `errors.blade.php` file in the `resources/views/admin/partials` directory with the following content.

Content of admin.partials.errors View

```

1 @if (count($errors) > 0)
2     <div class="alert alert-danger">
3         <strong>Whoops!</strong>
4         There were some problems with your input.<br><br>
5         <ul>
6             @foreach ($errors->all() as $error)
7                 <li>{{ $error }}</li>
8             @endforeach
9         </ul>
10    </div>
11 @endif

```

The `$errors` variable is available to every view. It will contain a collection of errors, if there are any. So here we just check if there's any errors and output them.

Creating the Post Index View

Create a new directory named `resources/views/admin/post` and within that directory a new file named `index.blade.php` with the following content.

Initial Content of admin.post.index View

```

1 @extends('admin.layout')
2
3 @section('content')
4     <div class="container-fluid">
5         <div class="row">
6             <div class="col-md-8 col-md-offset-2">
7                 <div class="panel panel-default">
8                     <div class="panel-heading">
9                         <h3 class="panel-title">Posts</h3>
10                    </div>
11                    <div class="panel-body">
12
13                        TODO
14
15                    </div>
16                </div>
17            </div>
18        </div>

```

```

19     </div>
20 @stop

```

This is just a temporary view at this point. In a future chapter we'll finish it.

Testing logging in and out

Point your browser to `http://l5beauty.app/admin` and see what happens. You should see the logon screen.

Here's actually what happened to get to the logon screen.

1. The admin URI was matched (in `app/Http/routes.php`) and the closure was executed which redirected your browser to `/admin/post`.
2. The route named **admin.post.index** (see `artisan route:list`) was matched from the `/admin/post` URI, but the **auth** middleware determined no user was logged in and did another redirect, this time to `/auth/login`.
3. The `/auth/login` URI was matched and the `getLogin()` method of the `Auth\AuthController` was executed. (*This method actually resides in the `AuthenticateUsers` trait.*)
4. The `getLogin()` method returned the contents of the **auth.login** view, which is the login screen you should have seen.

Now, you could try logging in, but at this point it won't do much good since we haven't added a user to the system yet.

Creating the admin user

The `artisan tinker` command is a handy way to interact with your application. Follow the steps below to create an administration user for the l5beauty blog.

Creating a user using tinker

```

~/Code/l5beauty$ php artisan tinker
Psy Shell v0.4.3 (PHP 5.6.7-1+deb.sury.org~utopic+1 â€” cli) by Justin Hileman
>>> $user = new App\User;
=> <App\User #000000007543b78f0000000009f4a1ca> {}
>>> $user->name = 'Your Name';
=> "Your Name"
>>> $user->email = 'YOUR@email.com';
=> "YOUR@email.com"
>>> $user->password = bcrypt('YOUR PASSWORD');

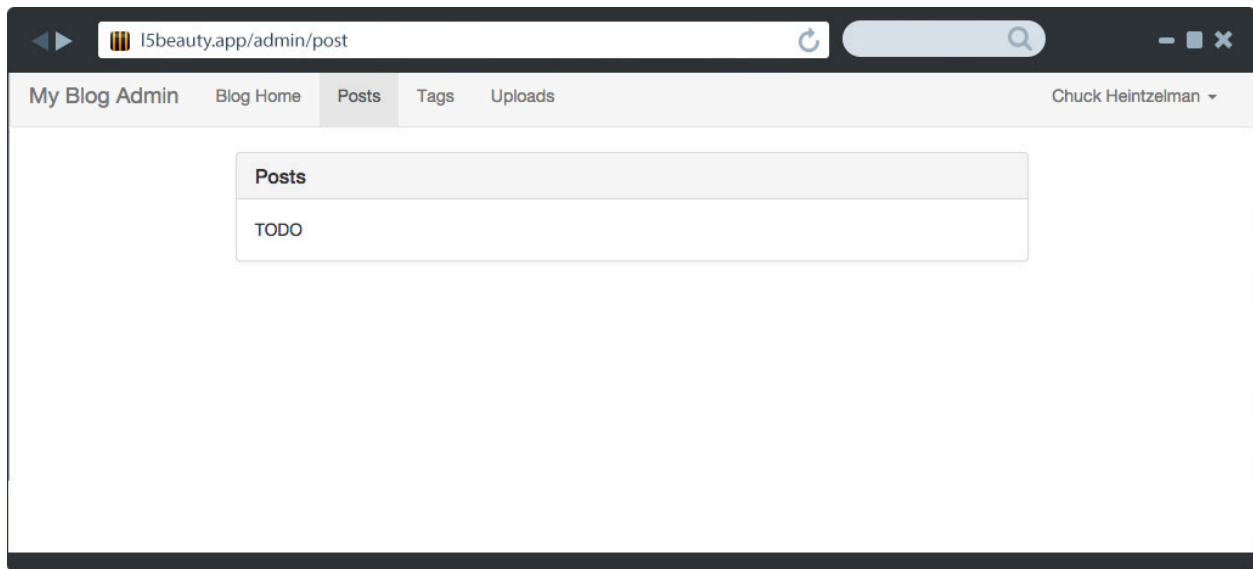
```



```
=> "$2y$10$gBF9EIr9IrIbMX7dwQsCT06IsBC0/c0P6qzZ01zwPhoW61MMw0VgC"  
>>> $user->save();  
=> true  
>>> exit;
```

Now you'll be able to log in with this user you just created. Go back to the login page on the browser and give it a try.

A successful login should present you with the page below.



List of Posts

Fixing log out location

If you click on the pull down at the top right of the screen and choose **Logout** to log out, you'll notice that instead of the login page, now you're back at the blog page.

Why is this?

If you look at the `AuthController` class (in the `app/Http/Controllers/Auth` directory) for the `getLogout()` method, you'll discover there isn't any method by that name. And there's no `getLogout()` in the parent class, nor the grandparent. But `AuthController` does use the `AuthenticatesAndRegistersUsers` trait, and that uses the `AuthenticatesUsers` trait. It's in the `AuthenticateUsers` trait where the `getLogout()` method is hiding.

So if you look way down the directory path in the `vendor/laravel/framework/src` and go further down into the `Illuminate/Foundation/Auth` directory you'll find this trait in the `AuthenticateUsers.php` file. And in that file, you can see the `getLogout()` method redirects to the root path ('/') when done.

To fix this, edit the `AuthController` class (it's in the `app/Http/Controllers/Auth` directory) so it looks like what's below.

Content of `AuthController` class

```
1 <?php
2
3 namespace App\Http\Controllers\Auth;
4
5 use App\User;
6 use Validator;
7 use App\Http\Controllers\Controller;
8 use Illuminate\Foundation\Auth\AuthenticatesUsers;
9
10 class AuthController extends Controller
11 {
12     use AuthenticatesUsers;
13
14     protected $redirectAfterLogout = '/auth/login';
15     protected $redirectTo = '/admin/post';
16
17     /**
18      * Create a new authentication controller instance.
19      */
20     public function __construct()
21     {
22         $this->middleware('guest', ['except' => 'getLogout']);
23     }
24
25     /**
26      * Get a validator for an incoming registration request.
27      *
28      * @param array $data
29      * @return \Illuminate\Contracts\Validation\Validator
30      */
31     protected function validator(array $data)
32     {
33         return Validator::make($data, [
34             'name' => 'required|max:255',
35             'email' => 'required|email|max:255|unique:users',
36             'password' => 'required|confirmed|min:6',
37         ]);
38     }
39 }
```

We removed the `AuthenticateAndRegistersUsers` trait because our application isn't going to allow users to be registered, instead we're just using the `AuthenticatesUsers` trait. Then we added the `$redirectAfterLogout` and `$redirectTo` properties to specify where to redirect after logins and logouts.

The reset of the class is the same as it was before.

Fix default login location

The default login location is currently set to `/home`, but we'll change it.

Edit the `RedirectIfAuthenticated.php` file located in the `app/Http/Middleware` directory. Change line #38 to what is below.

Change default logged in location

```
// Line #38 should be
return new RedirectResponse('/home');

// Change it to
return new RedirectResponse('/admin/post');
```

This path will redirect when the **guest** middleware is used on a route.

Logging in and out

Since the `getLogout()` method is now fixed to return back to the administration area instead of the blog, you should be able to log in and out successfully.

Try it.

- Point your browser back to `http://15beauty.app/admin`. This will redirect you to the log in page.
- Enter the credentials for the user you created. This will take you to the **List of Posts** page.
- Use the dropdown in the navbar to logout. This will take you back to the log in page.

Cleaning up a couple unneeded files

There's a couple files we don't need. Let's remove them.

- `app/Http/Controllers/Auth/PasswordController.php` - We're not going to implement password resets so this isn't needed.
- `resources/views/welcome.blade.php` - Not using this view.

Recap

Quite a bit was accomplished in this chapter. The routes were established for most of the administration area and there was a brief discussion about Middleware and Resource Controllers in Laravel 5. The logging in and out process was customized specifically for our administration area.

All in all, we now have a strong base to build the admin area upon. In the next chapter we'll start making the admin area useful and add *Tags* to our blog system.

Chapter 9 - Using Bower

In this chapter we'll work on some of the supporting software the administration area will be built on. Namely, how the assets are pulled in and which assets are used. The build system will use bower and gulp to automatically download and combine jQuery, Bootstrap, Font Awesome, and DataTables from the Internet.

Chapter Contents

- [Stealing Code](#)
- [Installing Bower](#)
- [Pulling in Bootstrap](#)
- [Creating admin.less](#)
- [Gulping Bootstrap](#)
- [Running gulp](#)
- [Updating the admin layout](#)
- [Adding FontAwesome and DataTables](#)
- [Recap](#)

Stealing Code

One of the fastest ways to develop web applications is to leverage the work of others. In other words, stealing their code.

No, you're not really stealing.

For instance, look at Twitter Bootstrap's license. It states anyone is allowed to use, free of charge, the Bootstrap framework.

You can't steal something that's free.

Always look at the license of any library you use.

Point is, today's web sites consist of many things: frameworks, utilities, libraries, assets, and so forth. It'd take forever to create a decent application if every component had to be created from scratch.

We'll use Bower to manage fetching and installing many of the packages we'll *steal*.

Installing Bower



Where Should Bower Run?

Bower's one of those utilities you can run either from your Host OS or from the Homestead VM. In this book I'll use the Host OS, but if you have any issues just use the Homestead VM.

Decide where you'll run bower and consistently *only run it from there*.

Since bower runs with NodeJS, you need to install it globally first. If you haven't already installed bower globally, follow the instructions below.

Installing Bower Globally

```
~% npm install -g bower
/usr/local/bin/bower -> /usr/local/lib/node_modules/bower/bin/bower
bower@1.4.1 /usr/local/lib/node_modules/bower
└─ is-root@1.0.0
[snip]
```

(Note you may need to use `sudo` or run the above command from a Windows Command Prompt with Administration privileges.)

Next create a `.bowerrc` file in the root of the **l5beauty** project. This is optional. What we're doing here is telling bower to stash anything it downloads into the `vendor` directory. If you skip this step then bower will create a directory named `bower_components` in your root directory and store items there.

Contents of `.bowerrc`

```
{
  "directory": "vendor/bower_d1"
}
```

Then install bower locally within the **l5project**

Installing Bower Locally

```
~% cd Code/l5beauty
~/Code/l5beauty% npm install bower
bower@1.4.1 node_modules/bower
├─ is-root@1.0.0
├─ junk@1.0.1
[snip]
```

Finally, create the `bower.json` file in the project root. This is will be where bower keeps track of packages to maintain. It's like `composer.json`, but for bower.

Contents `bower.json` file

```
{
  "name": "l5beauty",
  "description": "My awesome blog",
  "ignore": [
    "**/*.*",
    "node_modules",
    "vendor/bower_dl",
    "test",
    "tests"
  ]
}
```

Pulling in Bootstrap

Now that bower's set up to use, let's use it to pull some assets off from the web which we want to be part of the administration area of **l5beauty**.

Since we're currently using Bootstrap, we'll start with that (and jQuery).

Installing JQuery and Bootstrap

```
~/Code/l5beauty% bower install jquery bootstrap --save
bower jquery#*          not-cached git://github.com/jquery/jquery.git#*
bower jquery#*          resolve git://github.com/jquery/jquery.git#*
bower bootstrap#*       not-cached git://github.com/twbs/bootstrap.git#*
bower bootstrap#*       resolve git://github.com/twbs/bootstrap.git#*
bower jquery#*          download https://github.com/.../2.1.4.tar.gz
bower bootstrap#*       download https://github.com/.../v3.3.5.tar.gz
bower jquery#*          extract archive.tar.gz
bower bootstrap#*       extract archive.tar.gz
bower jquery#*          resolved git://github.com/jquery/jquery.git#2.1.4
bower bootstrap#*       resolved git://github.com/...git#3.3.5
bower jquery#>= 1.9.1    install jquery#2.1.4
bower bootstrap#~3.3.5  install bootstrap#3.3.5
```

jquery#2.1.4 vendor/bower_dl/jquery

bootstrap#3.3.5 vendor/bower_dl/bootstrap

└─ jquery#2.1.4

(Your output may vary slightly.)

Now if you look at `bower.json` you'll notice two dependencies were added. One for jquery and one for bootstrap.

New dependencies in bower.json

```
{
  ...
  "dependencies": {
    "jquery": "~2.1.4",
    "bootstrap": "~3.3.5"
  }
}
```

These two packages were downloaded into the `vendor/bower_dl` directory.



The bower update command

To get the latest versions of any of your bower dependencies, simply run the `bower update` command from the root directory of the **l5beauty** project.

Creating admin.less

We'll use gulp to compile Bootstrap's less file for the administration pages. Create `admin.less` in the `resources/assets/less` directory with the following content.

Content of admin.less

```
@import "bootstrap/bootstrap";
@import "//fonts.googleapis.com/css?family=Roboto:400,300";

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

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

First we import the `bootstrap.less` file (*which doesn't exist yet, we'll copy it to the correct location with gulp shortly.*) Then we import the font we'll be using and add a few small tweaks to the CSS.

Gulping Bootstrap

Now that bower is pulling in the latest jQuery and Bootstrap, Gulp can be used to merge it into your project.

Update `gulpfile.js` to match what's below.

New gulpfile.js

```
1 var gulp = require('gulp');
2 var elixir = require('laravel-elixir');
3
4 /**
5  * Copy any needed files.
6  *
7  * Do a 'gulp copyfiles' after bower updates
8  */
9 gulp.task("copyfiles", function() {
10
11     gulp.src("vendor/bower_dl/jquery/dist/jquery.js")
12         .pipe(gulp.dest("resources/assets/js/"));
13
14 }
```

```
14  gulp.src("vendor/bower_dl/bootstrap/less/**")
15    .pipe(gulp.dest("resources/assets/less/bootstrap"));
16
17  gulp.src("vendor/bower_dl/bootstrap/dist/js/bootstrap.js")
18    .pipe(gulp.dest("resources/assets/js/"));
19
20  gulp.src("vendor/bower_dl/bootstrap/dist/fonts/**")
21    .pipe(gulp.dest("public/assets/fonts"));
22
23  });
24
25  /**
26   * Default gulp is to run this elixir stuff
27   */
28  elixir(function(mix) {
29
30    // Combine scripts
31    mix.scripts([
32      'js/jquery.js',
33      'js/bootstrap.js'
34    ],
35      'public/assets/js/admin.js',
36      'resources/assets'
37    );
38
39    // Compile Less
40    mix.less('admin.less', 'public/assets/css/admin.css');
41  });
```

This file changed quite a bit.

First of all we added a **copyfiles** gulp task. The reason we're doing this is twofold:

1. Gulp runs asynchronously. When files are copied they may still be being copied when they're combined with `mix.less()` or `mix.scripts()`.
2. Copying the files only needs to occur after a `bower update`.

The `scripts()` function will combine `jquery.js` and `bootstrap.js` into one file, which we're naming `public/assets/js/admin.js`. And the `less()` function will process the `admin.less` file we just created, pulling in `bootstrap`.

`phpUnit()` was removed from `gulpfile.js` because we're not worried about testing in this book.

Running gulp

Since bower has been updated, run `gulp copyfiles` followed by `gulp` without any argument.

Running gulp twice

```
~/Code/15beauty% gulp copyfiles
[11:54:39] Using gulpfile ~/Projects/15beauty/gulpfile.js
[11:54:39] Starting 'copyfiles'...
[11:54:39] Finished 'copyfiles' after 19 ms

~/Code/15beauty% gulp
[11:55:33] Using gulpfile ~/Projects/15beauty/gulpfile.js
[11:55:33] Starting 'default'...
[11:55:33] Starting 'scripts'...
[11:55:33] Merging: resources/assets/js/jquery.js,
resources/assets/js/bootstrap.js
[11:55:34] Finished 'default' after 106 ms
[11:55:34] Finished 'scripts' after 228 ms
[11:55:34] Starting 'less'...
[11:55:34] Running Less: resources/assets/less/admin.less
[11:55:35] gulp-notify: [Laravel Elixir] Less Compiled!
[11:55:35] Finished 'less' after 829 ms
```

Everything should copy, combine, and compile as expected. Examine the `public/assets` directory to double-check what's expected there exists. You should have the following:

- The directory `public/assets/fonts`
- The file `public/assets/css/admin.css`
- The file `public/assets/js/admin.js`

Updating the admin layout

Since we're loading jQuery and Bootstrap locally now from the combined files, the administration layout can be updated. Update `layout.blade.php` in the `resources/views/admin` directory to match what's below.

New admin layout

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1">
7
8      <title>{{ config('blog.title') }} Admin</title>
9
10     <link href="/assets/css/admin.css" rel="stylesheet">
11     @yield('styles')
12
13     <!--[if lt IE 9]>
14         <script src="//oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
15         <script src="//oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
16     <![endif]-->
17 </head>
18 <body>
19
20 <nav class="navbar navbar-default">
21     <div class="container-fluid">
22         <div class="navbar-header">
23             <button type="button" class="navbar-toggle collapsed"
24                 data-toggle="collapse" data-target="#navbar-menu">
25                 <span class="sr-only">Toggle Navigation</span>
26                 <span class="icon-bar"></span>
27                 <span class="icon-bar"></span>
28                 <span class="icon-bar"></span>
29             </button>
30             <a class="navbar-brand" href="#">{{ config('blog.title') }} Admin</a>
31         </div>
32         <div class="collapse navbar-collapse" id="navbar-menu">
33             @include('admin.partials.navbar')
34         </div>
35     </div>
36 </nav>
37
38 @yield('content')
39
40 <script src="/assets/js/admin.js"></script>
41

```

```

42 @yield('scripts')
43
44 </body>
45 </html>

```

The only changes were the CSS link in the header and replacing the two script lines at the bottom with the new, combined `admin.js`.

Go to `http://15beauty.app/admin` in your browser. The administration area (and logon form) should look exactly the same as it did before.

Adding FontAwesome and DataTables

Now that bower and gulp are set up to handle our assets correctly, let's add two more packages: [Font Awesome](#)²⁷ and [DataTables](#)²⁸.

Follow the instructions below to add these packages as bower dependencies.

Adding FontAwesome and DataTables to bower

```

~/Code/15beauty% bower install fontawesome --save
~/Code/15beauty% bower install datatables --save
~/Code/15beauty% bower install datatables-plugins --save

```

(We're adding datatables-plugins in order to use bootstrap specific styles in DataTables.)

Next edit `gulpfile.js` to copy the needed assets into our project.

Updated gulpfile.js

```

1  var gulp = require('gulp');
2  var rename = require('gulp-rename');
3  var elixir = require('laravel-elixir');
4
5  /**
6   * Copy any needed files.
7   *
8   * Do a 'gulp copyfiles' after bower updates
9   */
10 gulp.task("copyfiles", function() {
11

```

²⁷<http://fontawesome.github.io/Font-Awesome/>

²⁸<https://www.datatables.net/>

```
12  // Copy jQuery, Bootstrap, and FontAwesome
13  gulp.src("vendor/bower_dl/jquery/dist/jquery.js")
14    .pipe(gulp.dest("resources/assets/js/"));
15
16  gulp.src("vendor/bower_dl/bootstrap/less/**")
17    .pipe(gulp.dest("resources/assets/less/bootstrap"));
18
19  gulp.src("vendor/bower_dl/bootstrap/dist/js/bootstrap.js")
20    .pipe(gulp.dest("resources/assets/js/"));
21
22  gulp.src("vendor/bower_dl/bootstrap/dist/fonts/**")
23    .pipe(gulp.dest("public/assets/fonts"));
24
25  gulp.src("vendor/bower_dl/fontawesome/less/**")
26    .pipe(gulp.dest("resources/assets/less/fontawesome"));
27
28  gulp.src("vendor/bower_dl/fontawesome/fonts/**")
29    .pipe(gulp.dest("public/assets/fonts"));
30
31  // Copy datatables
32  var dtDir = 'vendor/bower_dl/datatables-plugins/integration/';
33
34  gulp.src("vendor/bower_dl/datatables/media/js/jquery.dataTables.js")
35    .pipe(gulp.dest('resources/assets/js/'));
36
37  gulp.src(dtDir + 'bootstrap/3/dataTables.bootstrap.css')
38    .pipe(rename('dataTables.bootstrap.less'))
39    .pipe(gulp.dest('resources/assets/less/others/'));
40
41  gulp.src(dtDir + 'bootstrap/3/dataTables.bootstrap.js')
42    .pipe(gulp.dest('resources/assets/js/'));
43
44  });
45
46  /**
47   * Default gulp is to run this elixir stuff
48   */
49  elixir(function(mix) {
50
51    // Combine scripts
52    mix.scripts([
53      'js/jquery.js',
```

```

54     'js/bootstrap.js',
55     'js/jquery.dataTables.js',
56     'js/dataTables.bootstrap.js'
57 ],
58 'public/assets/js/admin.js',
59 'resources/assets'
60 );
61
62 // Compile Less
63 mix.less('admin.less', 'public/assets/css/admin.css');
64 });

```

Now edit the `admin.less` file in `resources/assets/less` to match what's below.

Content of `admin.less`

```

1  @import "bootstrap/bootstrap";
2  @import "//fonts.googleapis.com/css?family=Roboto:400,300";
3
4  @btn-font-weight: 300;
5  @font-family-sans-serif: "Roboto", Helvetica, Arial, sans-serif;
6
7  body, label, .checkbox label {
8    font-weight: 300;
9  }
10
11 @import "fontawesome/font-awesome";
12 @import "others/dataTables.bootstrap.less";

```

Since we're renaming a file using gulp instead of elixir (for that *copyfiles* task), we need to pull in the `gulp-rename` module.

Pulling in `gulp-rename`

```

%/Code/15beauty% npm install gulp-rename --save
gulp-rename@1.2.2 node_modules/gulp-rename

```

Now run `gulp` twice. Once to copy the files (`gulp copyfiles`) because we added bower assets and once to process and combine the assets.

Running gulp twice

```
~/Code/l5beauty% gulp copyfiles
[12:52:02] Using gulpfile ~/Projects/l5beauty/gulpfile.js
[12:52:02] Starting 'copyfiles'...
[12:52:02] Finished 'copyfiles' after 31 ms

~/Code/l5beauty% gulp
[12:52:26] Using gulpfile ~/Projects/l5beauty/gulpfile.js
[12:52:26] Starting 'default'...
[12:52:26] Starting 'scripts'...
[12:52:26] Merging: resources/assets/js/jquery.js,
resources/assets/js/bootstrap.js, resources/assets/js/jquery.dataTables.js,
resources/assets/js/dataTables.bootstrap.js
[12:52:26] Finished 'default' after 107 ms
[12:52:26] Finished 'scripts' after 401 ms
[12:52:26] Starting 'less'...
[12:52:26] Running Less: resources/assets/less/admin.less
[12:52:30] gulp-notify: [Laravel Elixir] Less Compiled!
[12:52:30] Finished 'less' after 4.19 s
```

Recap

In this chapter we used bower to download jQuery, Bootstrap, Font Awesome, and DataTables from the Internet. Then gulp was used to combine everything into a single CSS file (admin.css) and a single JavaScript file (admin.js).

These pieces will come together in the next chapter when we implement a tagging system for the **l5beauty** project.

Chapter 10 - Blog Tags

The basic blog built in the *Chapter 7 - The 10 Minute Blog* chapter wasn't very fancy. Most blogging platforms allow blog posts to be categorized or "tagged" in different ways. In this chapter we'll develop a tagging system for the **l5beauty** project.

Chapter Contents

- Creating the Model and Migrations
 - Editing tags migration
 - Editing post tag pivot migration
 - Running Migrations
- Implementing admin.tag.index
 - Implementing TagController index
 - Adding the admin.tag.index view
 - The success and errors partial
 - The empty list
- Implementing admin.tag.create
 - Implementing TagController create
 - Adding the admin.tag.create view
 - Adding the admin.tag._form partial
 - The screen
- Implementing admin.tag.store
 - Adding TagCreateRequest
 - Implementing TagController store
- Implementing admin.tag.edit
 - Implementing TagController edit
 - Adding admin.tag.edit view
 - The screen
- Implementing admin.tag.update
 - Adding TagUpdateRequest
 - Implementing TagController update
- Finishing the Tag System
 - Implementing TagController destroy
 - Removing admin.tag.show
- Recap

Creating the Model and Migrations

First step is to create the Tag model. Do this from within the Homestead VM.

Creating Tag Model

```
~/Code/15beauty$ php artisan make:model --migration Tag
Model created successfully.
Created Migration: 2015_04_07_044634_create_tags_table
```

This creates the model in the file `app/Tag.php`. Because we used the `--migration` option the `make:model` command will automatically create a migration for the database.

There will be a many-to-many relationship between **Tags** and **Posts**. Follow the command below to create a migration for the pivot table to store this relationship.

Creating the Post and Tag Pivot Migration

```
~/Code/15beauty$ php artisan make:migration --create=post_tag_pivot \
    create_post_tag_pivot
Created Migration: 2015_04_07_044734_create_post_tag_pivot
```

Editing tags migration

Edit the newly created tags migration file in the `database/migrations` directory. Make it match what's below.

The tags table create migration

```
1 <?php
2
3 use Illuminate\Database\Schema\Blueprint;
4 use Illuminate\Database\Migrations\Migration;
5
6 class CreateTagsTable extends Migration
7 {
8     /**
9      * Run the migrations.
10     */
11     public function up()
12     {
13         Schema::create('tags', function (Blueprint $table) {
14             $table->increments('id');
```

```
15     $table->string('tag')->unique();
16     $table->string('title');
17     $table->string('subtitle');
18     $table->string('page_image');
19     $table->string('meta_description');
20     $table->string('layout')->default('blog.layouts.index');
21     $table->boolean('reverse_direction');
22     $table->timestamps();
23 });
24 }
25
26 /**
27  * Reverse the migrations.
28  */
29 public function down()
30 {
31     Schema::drop('tags');
32 }
33 }
```

Here's the description of a few of the columns.

page_image

The look and feel of the finished blog will allow a large image at the top of each page. If we're viewing a page of posts for a particular tag it'll be nice to show a different image. This column allows the image to be set.

meta_description

A description to embed in a meta tag for search engines.

layout

The blog will eventually use layouts

reverse_directions

Normally blogs list posts in the order of publication with the most recent on top. This flag will flip that order.

Editing post tag pivot migration

Edit the post/tag pivot migration created earlier to match what's below.

The post/tag pivot create migration

```

1  <?php
2
3  use Illuminate\Database\Schema\Blueprint;
4  use Illuminate\Database\Migrations\Migration;
5
6  class CreatePostTagPivot extends Migration
7  {
8      /**
9       * Run the migrations.
10      */
11     public function up()
12     {
13         Schema::create('post_tag_pivot', function (Blueprint $table) {
14             $table->increments('id');
15             $table->integer('post_id')->unsigned()->index();
16             $table->integer('tag_id')->unsigned()->index();
17         });
18     }
19
20     /**
21      * Reverse the migrations.
22      */
23     public function down()
24     {
25         Schema::drop('post_tag_pivot');
26     }
27 }

```

Running Migrations

Run the migrations to create the two new tables.

Running the migrations

```

~/Code/15beauty$ php artisan migrate
Migrated: 2015_04_07_044634_create_tags_table
Migrated: 2015_04_07_044734_create_post_tag_pivot

```

Implementing admin.tag.index

Back in *Chapter 8 - Starting the Admin Area* we created the TagController and routes to it. Implementing the admin.tag.index route will be fairly trivial. We'll update the index() method in the controller and provide a view.

Implementing TagController index

Update TagController.php in the app/Http/Controllers/Admin directory, making the changes specified below.

Updates to TagController for index

```
1 // Add the following use statement at the top
2 use App\Tag;
3
4 // Replace the index() method with what's below
5 /**
6  * Display a listing of the tags.
7  */
8 public function index()
9 {
10     $tags = Tag::all();
11
12     return view('admin.tag.index')
13         ->withTags($tags);
14 }
```

Easy, huh? All we're doing here is returning a view and passing it a variable that will be called \$tags containing all the tags from the database. The Tag is a model so we can use the all() method to return a collection of all tags on file.

Adding the admin.tag.index View

Create a new tag directory in the resources/views/admin folder and then create the index.blade.php file with the following contents.

Content of admin.tag.index View

```

1  @extends('admin.layout')
2
3  @section('content')
4      <div class="container-fluid">
5          <div class="row page-title-row">
6              <div class="col-md-6">
7                  <h3>Tags <small>&raquo; Listing</small></h3>
8              </div>
9              <div class="col-md-6 text-right">
10                 <a href="/admin/tag/create" class="btn btn-success btn-md">
11                     <i class="fa fa-plus-circle"></i> New Tag
12                 </a>
13             </div>
14         </div>
15
16         <div class="row">
17             <div class="col-sm-12">
18
19                 @include('admin.partials.errors')
20                 @include('admin.partials.success')
21
22                 <table id="tags-table" class="table table-striped table-bordered">
23                     <thead>
24                         <tr>
25                             <th>Tag</th>
26                             <th>Title</th>
27                             <th class="hidden-sm">Subtitle</th>
28                             <th class="hidden-md">Page Image</th>
29                             <th class="hidden-md">Meta Description</th>
30                             <th class="hidden-md">Layout</th>
31                             <th class="hidden-sm">Direction</th>
32                             <th data-sortable="false">Actions</th>
33                         </tr>
34                     </thead>
35                     <tbody>
36                         @foreach ($tags as $tag)
37                             <tr>
38                                 <td>{{ $tag->tag }}</td>
39                                 <td>{{ $tag->title }}</td>
40                                 <td class="hidden-sm">{{ $tag->subtitle }}</td>
41                                 <td class="hidden-md">{{ $tag->page_image }}</td>

```

```

42         <td class="hidden-md">{{ $tag->meta_description }}</td>
43         <td class="hidden-md">{{ $tag->layout }}</td>
44         <td class="hidden-sm">
45             @if ($tag->reverse_direction)
46                 Reverse
47             @else
48                 Normal
49             @endif
50         </td>
51         <td>
52             <a href="/admin/tag/{{ $tag->id }}/edit"
53                 class="btn btn-xs btn-info">
54                 <i class="fa fa-edit"></i> Edit
55             </a>
56         </td>
57     </tr>
58 @endforeach
59 </tbody>
60 </table>
61 </div>
62 </div>
63 </div>
64 @stop
65
66 @section('scripts')
67     <script>
68         $(function() {
69             $("#tags-table").DataTable({
70             });
71         });
72     </script>
73 @stop

```

This template is easy to follow. We're using the admin layout. At the top is the page title and a button to create new tags. Then a table outputting all the tags we have on file (that \$tags variable we passed to the view). And finally, there's a bit of JavaScript at the bottom to convert the table to DataTables.

There are two views included within the admin.tag.index view: admin.partials.errors and admin.partials.success. The admin.partials.errors view was created back in Chapter 8, but we haven't created the admin.partials.success view yet.

The success partial

Create the `success.blade.php` file in the `resources/views/admin/partials` directory with the following content.

The `admin.partials.success` View

```
1 @if (Session::has('success'))
2     <div class="alert alert-success">
3         <button type="button" class="close" data-dismiss="alert">&times;</button>
4         <strong>
5             <i class="fa fa-check-circle fa-lg fa-fw"></i> Success. &nbsp;
6         </strong>
7         {{ Session::get('success') }}
8     </div>
9 @endif
```

If there's a success value in the session, we output the alert.

We'll use **flash data** to store success whenever we want this type of feedback to the user.



What is Flash Data?

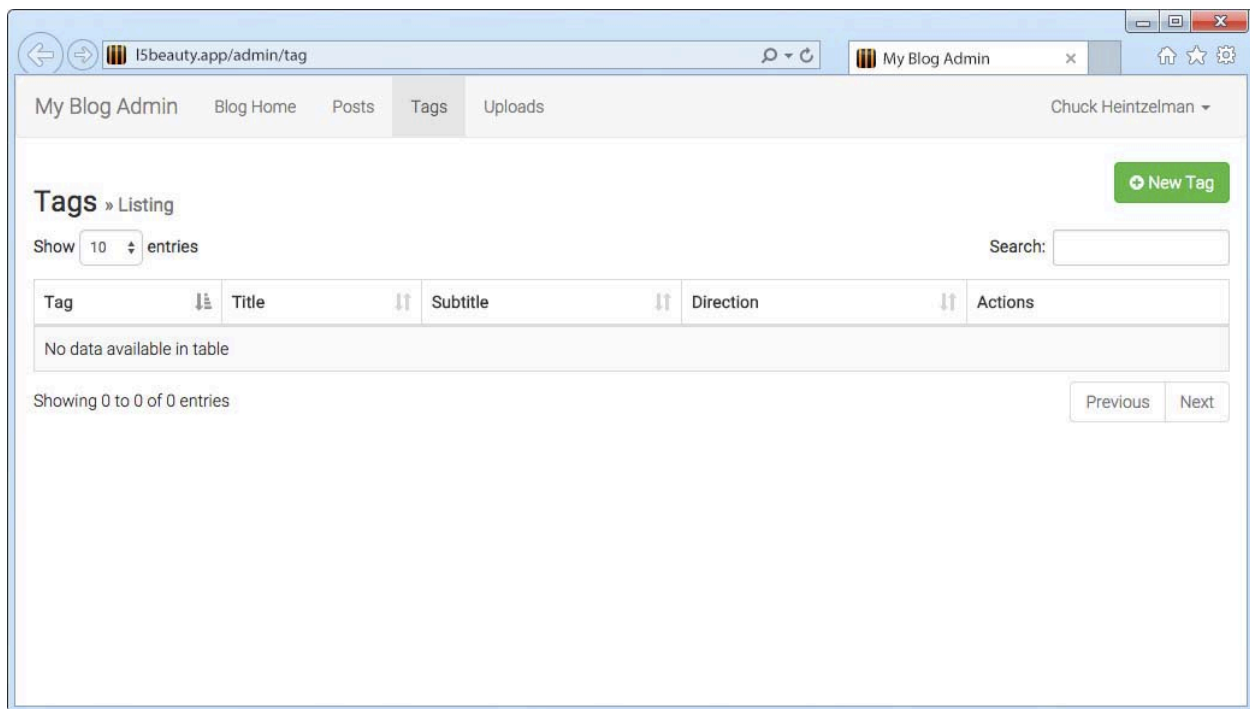
Often in web applications there's a need to store data in the session for only the next request. This is called **flash data**. Laravel 5.1 makes flashing data to the session easy. Use the `Session::flash('name', 'value')` method to do it.

Another way to set flash data is to return a **redirect** response and pass the value using a `->withName('value')` to the redirect before returning it.

The empty list

Bring up the `l5beauty` project in your browser, navigate to the administration page and click the **Tags** link at the top.

You should see a screen like the one below.



List of Tags

No data yet. But it sure does look professional.

Implementing admin.tag.create

Next we'll implement the screen that allows a new tag to be created. This way when you click on the [New Tag] button you'll have a form to fill out.

Implementing TagController create

Update the TagController class as instructed below. This file is in the app/Http/Controllers/Admin directory.

Updates to TagController for create

```

1 // Add the $fields property at the top of the class
2 class TagController extends Controller
3 {
4     protected $fields = [
5         'tag' => '',
6         'title' => '',
7         'subtitle' => '',
8         'meta_description' => '',

```

```
9     'page_image' => '',
10     'layout' => 'blog.layouts.index',
11     'reverse_direction' => 0,
12 ];
13
14 // Replace the create() method with this
15 /**
16  * Show form for creating new tag
17  */
18 public function create()
19 {
20     $data = [];
21     foreach ($this->fields as $field => $default) {
22         $data[$field] = old($field, $default);
23     }
24
25     return view('admin.tag.create', $data);
26 }
```

This method will be executed in two contexts:

1. When the [New Tag] button is hit
2. Or when the form is filled out but there's an input error.

If the first case we want the form to be populated with default values. In the second case we want to pass any data that was previously input back to the form. This is accomplished with the `old()` function which returns the previous input or a default.

The view is returned with `$data`. Thus each column will be a variable in the view. For example the `$layout` variable will exist in the view (and it'll have a default value of 'blog.layouts.index').

Adding the admin.tag.create View

Create the `create.blade.php` file in the `resources/views/admin/tag` directory with the content below.

Content of the admin.tag.create View

```

1  @extends('admin.layout')
2
3  @section('content')
4      <div class="container-fluid">
5          <div class="row page-title-row">
6              <div class="col-md-12">
7                  <h3>Tags <small>&raquo; Create New Tag</small></h3>
8              </div>
9          </div>
10
11         <div class="row">
12             <div class="col-md-8 col-md-offset-2">
13                 <div class="panel panel-default">
14                     <div class="panel-heading">
15                         <h3 class="panel-title">New Tag Form</h3>
16                     </div>
17                     <div class="panel-body">
18
19                         @include('admin.partials.errors')
20
21                         <form class="form-horizontal" role="form" method="POST"
22                             action="/admin/tag">
23                             <input type="hidden" name="_token" value="{{ csrf_token() }}">
24
25                             <div class="form-group">
26                                 <label for="tag" class="col-md-3 control-label">Tag</label>
27                                 <div class="col-md-3">
28                                     <input type="text" class="form-control" name="tag" id="tag"
29                                         value="{{ $tag }}" autofocus>
30                                 </div>
31                             </div>
32
33                             @include('admin.tag._form')
34
35                             <div class="form-group">
36                                 <div class="col-md-7 col-md-offset-3">
37                                     <button type="submit" class="btn btn-primary btn-md">
38                                         <i class="fa fa-plus-circle"></i>
39                                         &nbsp; Add New Tag
40                                     </button>
41                             </div>

```

```

42         </div>
43
44     </form>
45
46 </div>
47 </div>
48 </div>
49 </div>
50 </div>
51
52 @stop

```

This is a simple to follow Blade template. There's no navigation back to the **Tags Listing** page, but it's easy enough to use the **Tags** link in the navigation bar whenever the user needs to get back to the **Tags Listing** page.

The fields for the form itself will be in the `admin.tag._form` view because we can use the same form for both creation and editing.

Adding the admin.tag._form partial

The reason I named this view `_form`, beginning with an underscore is that I'm following a common Ruby convention of prefacing partials that aren't in their own directory with an underscore.

Create the `_form.blade.php` file in the `resources/views/admin/tag` directory with the content below.

Content of admin.tag._form View

```

1 <div class="form-group">
2   <label for="title" class="col-md-3 control-label">
3     Title
4   </label>
5   <div class="col-md-8">
6     <input type="text" class="form-control" name="title"
7       id="title" value="{{ $title }}">
8   </div>
9 </div>
10
11 <div class="form-group">
12   <label for="subtitle" class="col-md-3 control-label">
13     Subtitle
14   </label>

```

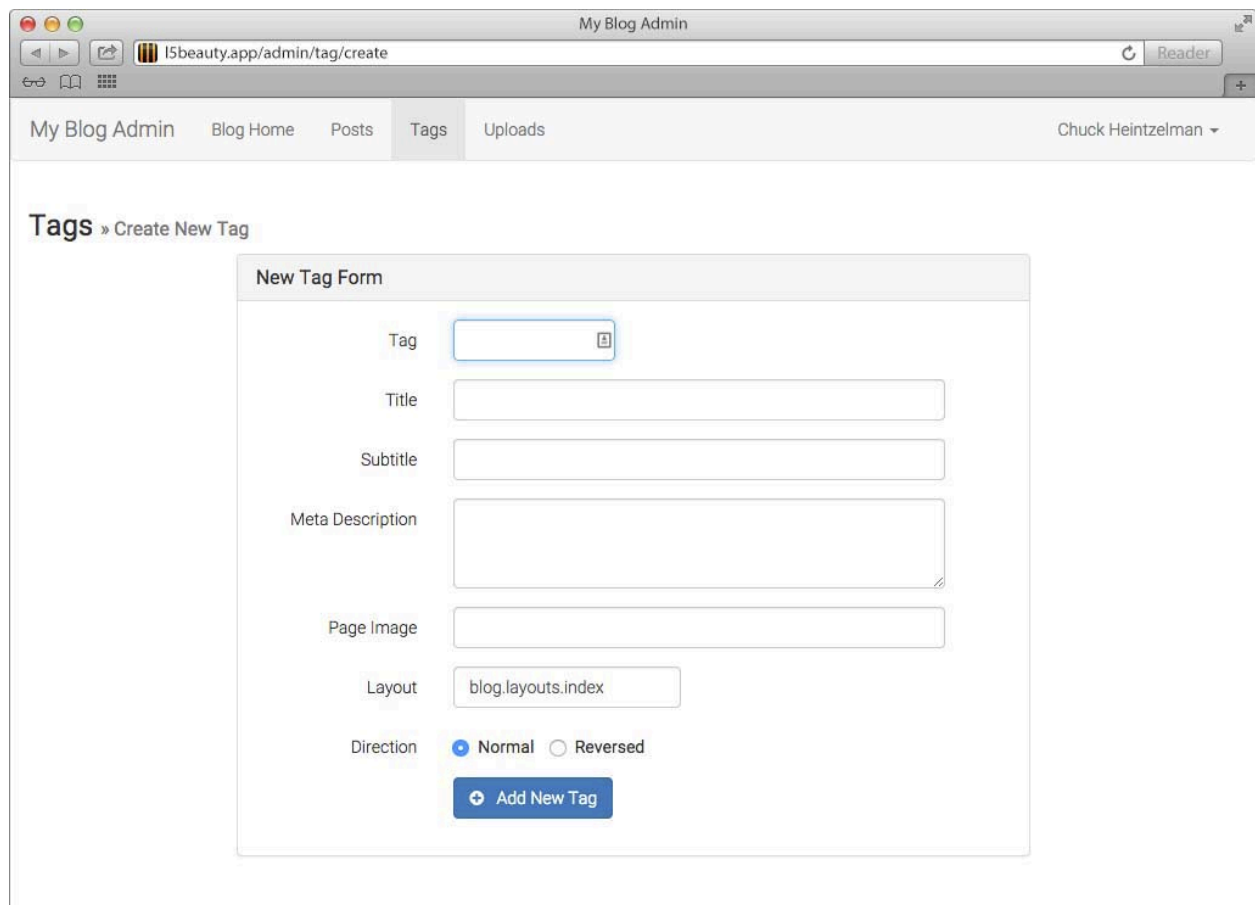
```
15     <div class="col-md-8">
16         <input type="text" class="form-control" name="subtitle"
17             id="subtitle" value="{{ $subtitle }}">
18     </div>
19 </div>
20
21 <div class="form-group">
22     <label for="meta_description" class="col-md-3 control-label">
23         Meta Description
24     </label>
25     <div class="col-md-8">
26         <textarea class="form-control" id="meta_description"
27             name="meta_description" rows="3">{{
28             $meta_description
29         }}</textarea>
30     </div>
31 </div>
32
33 <div class="form-group">
34     <label for="page_image" class="col-md-3 control-label">
35         Page Image
36     </label>
37     <div class="col-md-8">
38         <input type="text" class="form-control" name="page_image"
39             id="page_image" value="{{ $page_image }}">
40     </div>
41 </div>
42
43 <div class="form-group">
44     <label for="layout" class="col-md-3 control-label">
45         Layout
46     </label>
47     <div class="col-md-4">
48         <input type="text" class="form-control" name="layout" id="layout"
49             value="{{ $layout }}">
50     </div>
51 </div>
52
53 <div class="form-group">
54     <label for="reverse_direction" class="col-md-3 control-label">
55         Direction
56     </label>
```

```
57 <div class="col-md-7">
58   <label class="radio-inline">
59     <input type="radio" name="reverse_direction"
60       id="reverse_direction"
61       @if (! $reverse_direction)
62         checked="checked"
63       @endif
64       value="0"> Normal
65   </label>
66   <label class="radio-inline">
67     <input type="radio" name="reverse_direction"
68       @if ($reverse_direction)
69         checked="checked"
70       @endif
71       value="1"> Reversed
72   </label>
73 </div>
74 </div>
```

The screen

In your browser navigate to the screen we just created by clicking the [New Tag] button on the Tags Listing screen.

You should see a screen like the one below.

The image is a screenshot of a web browser window titled 'My Blog Admin'. The address bar shows '15beauty.app/admin/tag/create'. The browser's navigation bar includes 'My Blog Admin', 'Blog Home', 'Posts', 'Tags' (which is the active tab), and 'Uploads'. A user name 'Chuck Heintzelman' is visible in the top right. The main content area is titled 'Tags » Create New Tag'. Below this is a 'New Tag Form' box. The form contains several input fields: 'Tag' (a small text box with a help icon), 'Title' (a standard text box), 'Subtitle' (a standard text box), 'Meta Description' (a larger text box with a resize handle), 'Page Image' (a standard text box), 'Layout' (a dropdown menu showing 'blog.layouts.index'), and 'Direction' (radio buttons for 'Normal' and 'Reversed', with 'Normal' selected). At the bottom of the form is a blue button with a plus icon and the text 'Add New Tag'.

New Tag Form

Again, we have a beautiful looking form, created with a minimum of effort.

Implementing `admin.tag.store`

Now that we have a form for creating the tag, we need to build the code that will be executed when this form is filled out and submitted.

Adding `TagCreateRequest`

One of the great features of Laravel 5.1 are *Form Requests*. These are classes which contain validation logic for a particular form.

Create the new `TagCreateRequest` class with the artisan command below.

Creating the TagCreateRequest class

```
vagrant@homestead:~/Code/15beauty$ php artisan make:request TagCreateRequest
Request created successfully.
```

This creates a skeleton class in the app/Http/Requests directory. Edit the TagCreateRequest.php file created there to match what's below.

Content of TagCreateRequest.php

```
1 <?php
2 namespace App\Http\Requests;
3
4 class TagCreateRequest extends Request
5 {
6
7     /**
8      * Determine if the user is authorized to make this request.
9      *
10     * @return bool
11     */
12     public function authorize()
13     {
14         return true;
15     }
16
17     /**
18      * Get the validation rules that apply to the request.
19      *
20     * @return array
21     */
22     public function rules()
23     {
24         return [
25             'tag' => 'required|unique:tags,tag',
26             'title' => 'required',
27             'subtitle' => 'required',
28             'layout' => 'required',
29         ];
30     }
31 }
```

Up the hierarchy of this class is the `Illuminate\Foundation\Http\FormRequest`, which will perform the validation when the class is instantiated.

The two methods we must provide are:

1. `authorize()` - This method should return whether or not this request is authorized. Since we don't have any additional authorization (beyond the middleware protecting our route), we simply return `true`.
2. `rules()` - Return an array of validation rules. We want the **tag**, **title**, **subtitle**, and **layout** all to be required values. Additionally, the **tag** must be unique in the tags database.

There's many validation rules available and you can even create your own. See the [Laravel Documentation](#)²⁹ for details.



The Magic of Form Requests

The “magic” of Form Requests is that they validate input during construction and if validation fails, control returns back to the form the user was just on with the appropriate error messages.

This means if a Form Request is an argument of a controller method, the form will be validated before the first line of that method is executed.

Implementing TagController store

Update the `TagController.php` file to match what's below.

Updates to TagController for store

```

1 <?php
2 // Edit the "use" statements at the top so there's only the following
3 use App\Http\Controllers\Controller;
4 use App\Http\Requests\TagCreateRequest;
5 use App\Tag;
6
7 // Update the store() method to match what's below
8 /**
9  * Store the newly created tag in the database.
10  *
11  * @param TagCreateRequest $request
12  * @return Response
13  */

```

²⁹<http://laravel.com/docs/5.1/validation>

```

14 public function store(TagCreateRequest $request)
15 {
16     $tag = new Tag();
17     foreach (array_keys($this->fields) as $field) {
18         $tag->$field = $request->get($field);
19     }
20     $tag->save();
21
22     return redirect('/admin/tag')
23         ->withSuccess("The tag '$tag->tag' was created.");
24 }

```

Now, through the magic of [Dependency Injection](#)³⁰, the TagCreateRequest is constructed, the form is validated, and only if it passes validation will the object be passed to the store() method.

The store() method will then create and save the new Tag. Finally, it redirects back to the **Tags Listing** page, flashing a “success” message as it does.

Try adding a couple tags. You’ll be able to delete these tags later. Just get some data into the system to make sure it’s working.

Implementing admin.tag.edit

Next will edit the route named admin.tag.edit to present the form for editing a tag.

Implementing TagController edit

Update the TagController class, editing the edit() method to match what’s below.

Updates to TagController for edit

```

1  /**
2   * Show the form for editing a tag
3   *
4   * @param int $id
5   * @return Response
6   */
7  public function edit($id)
8  {
9      $tag = Tag::findOrFail($id);
10     $data = ['id' => $id];

```

³⁰<http://laravel.com/docs/5.1/controllers#dependency-injection-and-controllers>

```

11     foreach (array_keys($this->fields) as $field) {
12         $data[$field] = old($field, $tag->$field);
13     }
14
15     return view('admin.tag.edit', $data);
16 }

```

Nothing magical is happening here. This function loads a Tag object based on the id, builds the associative array `$data` with either values from this object or the *old* input values, and returns a view passing the values to it.

Adding admin.tag.edit View

In the `resources/views/admin/tag` folder, create the `edit.blade.php` view with the following content.

Content of admin.tag.edit View

```

1  @extends('admin.layout')
2
3  @section('content')
4      <div class="container-fluid">
5          <div class="row page-title-row">
6              <div class="col-md-12">
7                  <h3>Tags <small>&raquo; Edit Tag</small></h3>
8              </div>
9          </div>
10
11         <div class="row">
12             <div class="col-md-8 col-md-offset-2">
13                 <div class="panel panel-default">
14                     <div class="panel-heading">
15                         <h3 class="panel-title">Tag Edit Form</h3>
16                     </div>
17                     <div class="panel-body">
18
19                         @include('admin.partials.errors')
20                         @include('admin.partials.success')
21
22                         <form class="form-horizontal" role="form" method="POST"
23                             action="/admin/tag/{{ $id }}">
24                             <input type="hidden" name="_token" value="{{ csrf_token() }}">

```

```

25         <input type="hidden" name="_method" value="PUT">
26         <input type="hidden" name="id" value="{{ $id }}">
27
28         <div class="form-group">
29             <label for="tag" class="col-md-3 control-label">Tag</label>
30             <div class="col-md-3">
31                 <p class="form-control-static">{{ $tag }}</p>
32             </div>
33         </div>
34
35         @include('admin.tag._form')
36
37         <div class="form-group">
38             <div class="col-md-7 col-md-offset-3">
39                 <button type="submit" class="btn btn-primary btn-md">
40                     <i class="fa fa-save"></i>
41                     &nbsp; Save Changes
42                 </button>
43                 <button type="button" class="btn btn-danger btn-md"
44                     data-toggle="modal" data-target="#modal-delete">
45                     <i class="fa fa-times-circle"></i>
46                     Delete
47                 </button>
48             </div>
49         </div>
50
51     </form>
52
53 </div>
54 </div>
55 </div>
56 </div>
57 </div>
58
59 {{-- Confirm Delete --}}
60 <div class="modal fade" id="modal-delete" tabIndex="-1">
61     <div class="modal-dialog">
62         <div class="modal-content">
63             <div class="modal-header">
64                 <button type="button" class="close" data-dismiss="modal">
65                     &times;
66                 </button>

```

```

67         <h4 class="modal-title">Please Confirm</h4>
68     </div>
69     <div class="modal-body">
70         <p class="lead">
71             <i class="fa fa-question-circle fa-lg"></i> &nbsp;
72             Are you sure you want to delete this tag?
73         </p>
74     </div>
75     <div class="modal-footer">
76         <form method="POST" action="/admin/tag/{{ $id }}">
77             <input type="hidden" name="_token" value="{{ csrf_token() }}">
78             <input type="hidden" name="_method" value="DELETE">
79             <button type="button" class="btn btn-default"
80                 data-dismiss="modal">Close</button>
81             <button type="submit" class="btn btn-danger">
82                 <i class="fa fa-times-circle"></i> Yes
83             </button>
84         </form>
85     </div>
86 </div>
87 </div>
88 </div>
89
90 @stop

```

There's a lot going on in this template, but it should be easy to follow.

Line 1

As always, we're using the admin layout

Lines 19 and 20

And we use the partials to provide error or success feedback to the user.

Lines 22 - 25

Starting a form to save any edits. The `$id` is used so we're updating the correct tag. Hidden values are used for the CSRF Token and to fake a **PUT** method.

Line 35

Using the same basic form as we used for the create.

Lines 43 - 47

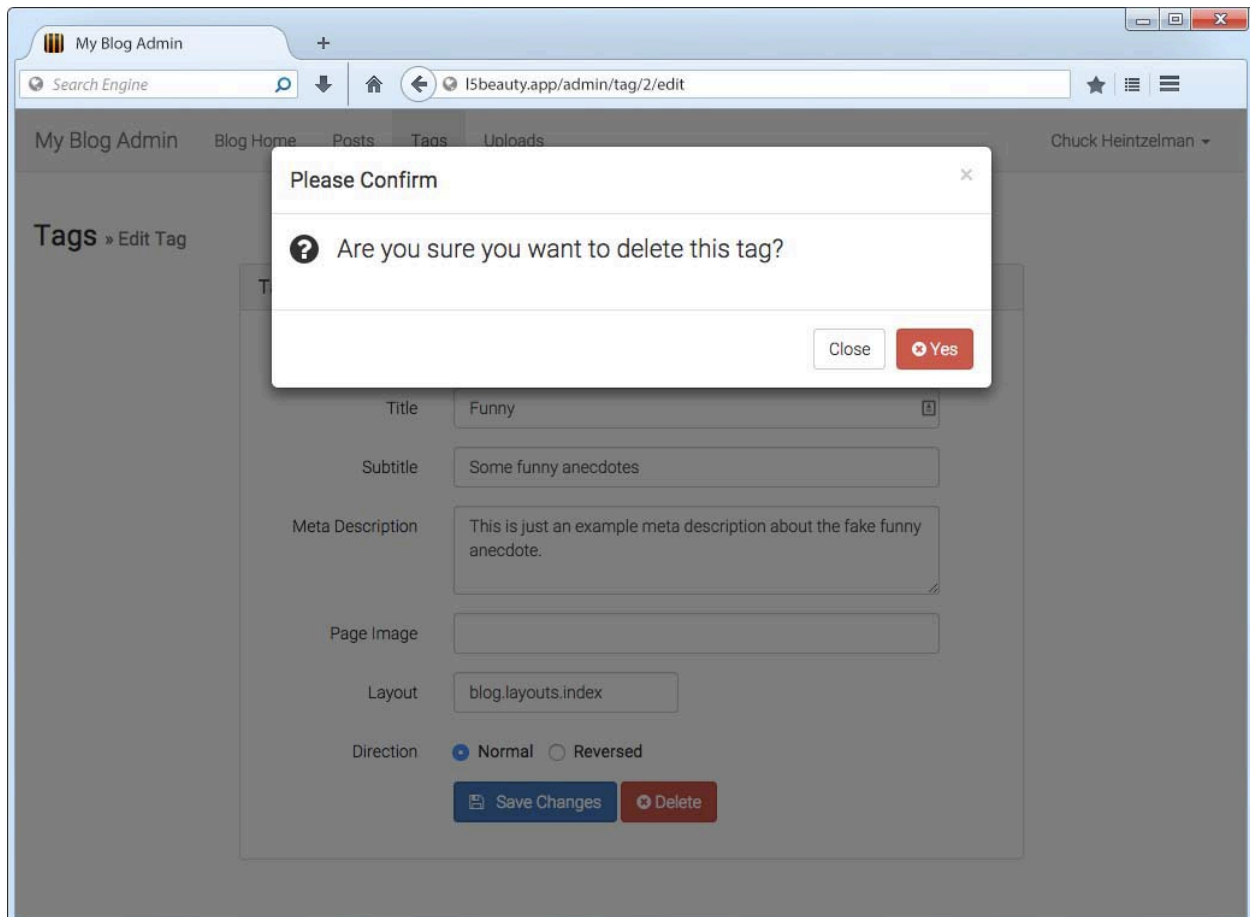
In addition to the standard "Save" button, here's a "Delete" button which will pop up a Modal dialog.

Lines 60 - 88

Here's the Modal dialog we're popping up. If the user chooses "Yes" then a POST is made which fakes a **DELETE** method.

The screen

Now if you click the **[edit]** button in your browser one of the tags added to the system (from the **Tags Listing**) screen, you'll see the edit form. If you click the **[Delete]** button from the edit form you'll see a screen similar to the one below.



Tag Edit Form

Of course, the tag cannot yet be deleted or updated, we'll do that next.

Implementing admin.tag.update

In the same way the `admin.tag.store` route processes the result of the `admin.tag.create` form, the `admin.tag.update` route will process the result of the `admin.tag.edit` form.

Adding TagUpdateRequest

To create the skeleton of the *Form Request* to handle the update, use the artisan command as illustrated below.

Creating the TagUpdateRequest class

```
~/Code/15beauty$ php artisan make:request TagUpdateRequest
Request created successfully.
```

Now edit the TagUpdateRequest.php file to match what's below. (*This file is in the app/Http/Requests directory.*)

Content of TagUpdateRequest.php

```
1 <?php
2 namespace App\Http\Requests;
3
4 class TagUpdateRequest extends Request
5 {
6
7     /**
8      * Determine if the user is authorized to make this request.
9      *
10     * @return bool
11     */
12     public function authorize()
13     {
14         return true;
15     }
16
17     /**
18      * Get the validation rules that apply to the request.
19      *
20     * @return array
21     */
22     public function rules()
23     {
24         return [
25             'title' => 'required',
26             'subtitle' => 'required',
27             'layout' => 'required',
28         ];
```

```

29     }
30 }

```

This request class is almost exactly the same as the `TagCreateRequest` we created earlier in this chapter.

Implementing TagController update

Update the `TagController.php` file to match what's below.

Updates to TagController for update

```

1  // Add the new use statement near the top
2  use App\Http\Requests\TagUpdateRequest;
3
4  // Replace the update() method with the following
5  /**
6   * Update the tag in storage
7   *
8   * @param TagUpdateRequest $request
9   * @param int $id
10  * @return Response
11  */
12  public function update(TagUpdateRequest $request, $id)
13  {
14      $tag = Tag::findOrFail($id);
15
16      foreach (array_keys(array_except($this->fields, ['tag'])) as $field) {
17          $tag->$field = $request->get($field);
18      }
19      $tag->save();
20
21      return redirect("/admin/tag/$id/edit")
22          ->withSuccess("Changes saved.");
23  }

```

Now you should be able to edit a tag and click **[Save Changes]** button successfully.

Finishing the Tag System

The administration side of our tagging system is almost finished. *(We'll tie the actual tags to posts in a later chapter when we implement the Post create and edit functionality.)*

The two things left are implementing the `destroy()` method and a bit of cleanup.

Implementing TagController destroy

To add the ability to delete tags from the edit screen, update the `destroy()` method in `TagController.php` as illustrated below.

Updates to `TagController` for `destroy()`

```
1  /**
2   * Delete the tag
3   *
4   * @param int $id
5   * @return Response
6   */
7  public function destroy($id)
8  {
9      $tag = Tag::findOrFail($id);
10     $tag->delete();
11
12     return redirect('/admin/tag')
13         ->withSuccess("The '$tag->tag' tag has been deleted.");
14 }
```

No magic here. Loads the tag, deletes it, and returns to the **Tags Listing** page with a success message.

Removing admin.tag.show

You may have noticed there's one route that hasn't been used. The `admin.tag.show` route was set up to show a tag. Often a *show* is used when a user doesn't have editing privileges, but does have viewing privileges.

The `admin.tag.show` route isn't needed in our system. So let's remove it.

First adjust the route in `app/Http/routes.php` as instructed below.

Updating routes to remove `admin.tag.show`

```
// Change the following line
    resource('admin/tag', 'TagController');
// to this
    resource('admin/tag', 'TagController', ['except' => 'show']);
```

This addition tells the router to set up all the resource routes except for the `show` route.

Then edit `TagController.php` and remove the `show()` method.



Congratulations

Now the **l5beauty** blog administration has all the functionality in place to create, update, and delete tags from your system.

Recap

This was a fairly long chapter, but in it we fully implemented the administration side of our tagging system. This included everything from setting up the Tag model, creating database migration, finishing the TagController class, using *Form Requests*, and creating the needed Blade template views.

And, don't forget, the resulting interface was simple and clean.

In the next chapter we'll add the "Upload" feature to the administration.

Chapter 11 - Upload Manager

In this chapter we'll create an *Upload Manager* for the blog administration. First, the local file system will be used to store any uploaded files. Then, we'll change the configuration to allow files to be stored on Amazon's S3 cloud storage.

Chapter Contents

- [Configuring the File System](#)
- [Adding a Helpers File](#)
- [Creating an Upload Manager Service](#)
 - [Detecting Mime Types](#)
 - [Creating the UploadsManager class](#)
- [Implementing UploadController index](#)
 - [Creating the index method](#)
 - [Creating the index view](#)
 - [The Upload Manager Screen](#)
- [Finishing the Upload Manager](#)
 - [Updating the routes](#)
 - [Adding all the Modal Dialogs](#)
 - [Adding the Request classes](#)
 - [Finishing UploadController](#)
 - [Finishing the UploadsManager Service class](#)
- [Setting Up Your S3 Account](#)
 - [1. Log in \(or sign up\) with Amazon](#)
 - [2. Get a Web Services Account](#)
 - [3. Create a S3 Bucket](#)
 - [4. Create an Access Key](#)
- [Configuring L5Beauty to Use S3](#)
- [Installing an Additional Package](#)
- [Test The Upload Manager](#)
- [Fixing Bucket Permissions](#)
- [Recap](#)

Configuring the File System

Let's start with the configuration. Specifically, where will any files be stored? Create an upload directory within the project's public directory. This way any files uploaded will be publicly accessible to the web.

Creating Upload Public Directory

```
vagrant@homestead:~/Code/15beauty$ mkdir public/uploads
```

Easy. Now edit `config/blog.php`, to match what's below.

Updated blog configuration

```
1 <?php
2 return [
3     'title' => 'My Blog',
4     'posts_per_page' => 5,
5     'uploads' => [
6         'storage' => 'local',
7         'webpath' => '/uploads',
8     ],
9 ];
```

With the above changes you'll be able to access the file system used with `config('blog.uploads.storage')`. The `config('blog.uploads.webpath')` will be the *root* of our storage on the web.

Finally, edit the `config/filesystems.php`, changing the section as specified below.

Changes to config/filesystems.php

```
1 // Change the following lines
2 'disks' => [
3     'local' => [
4         'driver' => 'local',
5         'root'   => storage_path('app'),
6     ],
7
8     ...
9
10    's3' => [
11        'driver' => 's3',
12        'key'    => 'your-key',
13        'secret' => 'your-secret',
14        'region' => 'your-region',
15        'bucket' => 'your-bucket',
16    ],
17
18    // to the following
```

```

19     'disks' => [
20         'local' => [
21             'driver' => 'local',
22             'root'   => public_path('uploads'),
23         ],
24
25         ...
26
27         's3' => [
28             'driver' => 's3',
29             'key'     => env('AWS_KEY'),
30             'secret'  => env('AWS_SECRET'),
31             'region'  => env('AWS_REGION'),
32             'bucket'  => env('AWS_BUCKET'),
33         ],

```

The first thing changed was the root of the local storage. This changed to the `public/uploads` directory just created a moment ago. Then each of the configuration settings for the Amazon S3 storage driver changed to pull the values from the environment. This is so we can change them later in our `.env` file.

Adding a Helpers file

With projects in Laravel 5.1 it's often handy to have a place to stash those little functions which don't warrant a class on their own. A common practice is to place these functions in a helpers file.

Create a new file in the app directory named `helpers.php`. Populate this file with the code below.

Content of `helpers.php`

```

1  <?php
2
3  /**
4   * Return sizes readable by humans
5   */
6  function human_filesize($bytes, $decimals = 2)
7  {
8      $size = ['B', 'kB', 'MB', 'GB', 'TB', 'PB'];
9      $factor = floor((strlen($bytes) - 1) / 3);
10
11     return sprintf("%.{$decimals}f", $bytes / pow(1024, $factor)) .
12         @$size[$factor];

```

```
13 }
14
15 /**
16  * Is the mime type an image
17  */
18 function is_image($mimeType)
19 {
20     return starts_with($mimeType, 'image/');
21 }
```

The `human_filesize()` function returns a file size which is easier to read than just a count of bytes. The `is_image()` function returns true if the mime type is an image.

To make your application aware of the `helpers.php` file, update `composer.json` as instructed below.

Updates to `composer.json` for `helpers.php`

```
{
    ...
    "autoload": {
        "classmap": [
            "database"
        ],
        "psr-4": {
            "App\\": "app/"
        },
        "files": [
            "app/helpers.php"
        ]
    },
    ...
}
```

The `files` array within the `autoload` section specifies any files to always load. Do a `composer dumpauto` to set up the autoloader correctly.

Composer `dumpauto`

```
vagrant@homestead:~/Code/15beauty$ composer dumpauto
Generating autoload files
```

Now any functions in `helpers.php` will always be loaded and available to the **15beauty** application.

Creating an Upload Manager Service

Now that the basic configuration is finished, let's create a *Service* class to manage our uploaded files.

Detecting Mime Types

Depending on the files being uploaded, we may want to have different actions occur. So it'd be nice to easily detect the Mime type of files.

PHP has the function `mime_content_type()`, but it's deprecated. The PHP Fileinfo functions can detect the mime type, but in Windows a DLL must be copied to make fileinfo functions work. Let's use a different solution.

Searching packagist.org for "mime" shows a package by **dflydev** which looks good.

Add the package to composer with the instructions below.

Adding dflydev mime package to composer

```
vagrant@homestead:~/Code/15beauty$ composer require "dflydev/apache-mime-types"
Using version ^1.0 for dflydev/apache-mime-types
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing dflydev/apache-mime-types (v1.0.1)
  Downloading: 100%

Writing lock file
Generating autoload files
```

We'll use this package in the `UploadsManager` class to detect a file's mime type.

Creating the UploadsManager class

Create the file `UploadsManager` in the `app/Services` directory with the following content.

Initial version of UploadsManager

```

1  <?php
2  namespace App\Services;
3
4  use Carbon\Carbon;
5  use Dflydev\ApacheMimeTypes\PhpRepository;
6  use Illuminate\Support\Facades\Storage;
7
8  class UploadsManager
9  {
10     protected $disk;
11     protected $mimeDetect;
12
13     public function __construct(PHPRepository $mimeDetect)
14     {
15         $this->disk = Storage::disk(config('blog.uploads.storage'));
16         $this->mimeDetect = $mimeDetect;
17     }
18
19     /**
20      * Return files and directories within a folder
21      *
22      * @param string $folder
23      * @return array of [
24      *     'folder' => 'path to current folder',
25      *     'folderName' => 'name of just current folder',
26      *     'breadCrumbs' => breadcrumb array of [ $path => $foldername ]
27      *     'folders' => array of [ $path => $foldername ] of each subfolder
28      *     'files' => array of file details on each file in folder
29      * ]
30      */
31     public function folderInfo($folder)
32     {
33         $folder = $this->cleanFolder($folder);
34
35         $breadcrumbs = $this->breadcrumbs($folder);
36         $slice = array_slice($breadcrumbs, -1);
37         $folderName = current($slice);
38         $breadcrumbs = array_slice($breadcrumbs, 0, -1);
39
40         $subfolders = [];
41         foreach (array_unique($this->disk->directories($folder)) as $subfolder) {

```



```
42     $subfolders["/$subfolder"] = basename($subfolder);
43 }
44
45 $files = [];
46 foreach ($this->disk->files($folder) as $path) {
47     $files[] = $this->fileDetails($path);
48 }
49
50 return compact(
51     'folder',
52     'folderName',
53     'breadcrumbs',
54     'subfolders',
55     'files'
56 );
57 }
58
59 /**
60  * Sanitize the folder name
61  */
62 protected function cleanFolder($folder)
63 {
64     return '/' . trim(str_replace('..', '', $folder), '/');
65 }
66
67 /**
68  * Return breadcrumbs to current folder
69  */
70 protected function breadcrumbs($folder)
71 {
72     $folder = trim($folder, '/');
73     $crumbs = ['/ => 'root'];
74
75     if (empty($folder)) {
76         return $crumbs;
77     }
78
79     $folders = explode('/', $folder);
80     $build = '';
81     foreach ($folders as $folder) {
82         $build .= '/' . $folder;
83         $crumbs[$build] = $folder;
```

```
84     }
85
86     return $crumbs;
87 }
88
89 /**
90  * Return an array of file details for a file
91  */
92 protected function fileDetails($path)
93 {
94     $path = '/' . ltrim($path, '/');
95
96     return [
97         'name' => basename($path),
98         'fullPath' => $path,
99         'webPath' => $this->fileWebpath($path),
100         'mimeType' => $this->fileMimeType($path),
101         'size' => $this->fileSize($path),
102         'modified' => $this->fileModified($path),
103     ];
104 }
105
106 /**
107  * Return the full web path to a file
108  */
109 public function fileWebpath($path)
110 {
111     $path = rtrim(config('blog.uploads.webpath'), '/') . '/' .
112         ltrim($path, '/');
113     return url($path);
114 }
115
116 /**
117  * Return the mime type
118  */
119 public function fileMimeType($path)
120 {
121     return $this->mimeType->findType(
122         pathinfo($path, PATHINFO_EXTENSION)
123     );
124 }
125
```

```

126  /**
127   * Return the file size
128   */
129  public function fileSize($path)
130  {
131      return $this->disk->size($path);
132  }
133
134  /**
135   * Return the last modified time
136   */
137  public function fileModified($path)
138  {
139      return Carbon::createFromTimestamp(
140          $this->disk->lastModified($path)
141      );
142  }
143  }

```

__construct()

Inject dependencies. Use the *disk* defined in the blog configuration and the class from the *dflydev/apache-mime-types* package required earlier.

folderInfo()

Here's the main method that returns everything needed about the contents of a folder.

Comments within the code explain the rest of the methods well enough you'll be able to follow what's going on.

Implementing UploadController index

Now that the UploadsManager service class has been created to do the bulk of our work, implementing the index method is almost trivial.

Creating the index method

Update the UploadController.php file located in the app/Http/Controllers/Admin directory to match what's below.

Updating UploadController for index

```
1  <?php
2  namespace App\Http\Controllers\Admin;
3
4  use App\Http\Controllers\Controller;
5  use App\Services\UploadsManager;
6  use Illuminate\Http\Request;
7
8  class UploadController extends Controller
9  {
10     protected $manager;
11
12     public function __construct(UploadsManager $manager)
13     {
14         $this->manager = $manager;
15     }
16
17     /**
18      * Show page of files / subfolders
19      */
20     public function index(Request $request)
21     {
22         $folder = $request->get('folder');
23         $data = $this->manager->folderInfo($folder);
24
25         return view('admin.upload.index', $data);
26     }
27 }
```

The constructor injects the UploadsManager dependency and the index() method simply returns the view with data returned from the folderInfo() method.

You probably noticed the \$folder is taken from the request. Yes, we'll just use a query argument to handle changing folders.

Creating the index view

Create the resources/views/admin/upload directory and within it create the index.blade.php file with the following content.

Content of admin.upload.index View

```

1 @extends('admin.layout')
2 
3 @section('content')
4     <div class="container-fluid">
5 
6         {{-- Top Bar --}}
7         <div class="row page-title-row">
8             <div class="col-md-6">
9                 <h3 class="pull-left">Uploads&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~</h3>
10                <div class="pull-left">
11                    <ul class="breadcrumb">
12                        @foreach ($breadcrumbs as $path => $disp)
13                            <li><a href="/admin/upload?folder={{ $path }}">{{ $disp }}</a></li>
14                        @endforeach
15                        <li class="active">{{ $folderName }}</li>
16                    </ul>
17                </div>
18            </div>
19            <div class="col-md-6 text-right">
20                <button type="button" class="btn btn-success btn-md"
21                    data-toggle="modal" data-target="#modal-folder-create">
22                    <i class="fa fa-plus-circle"></i> New Folder
23                </button>
24                <button type="button" class="btn btn-primary btn-md"
25                    data-toggle="modal" data-target="#modal-file-upload">
26                    <i class="fa fa-upload"></i> Upload
27                </button>
28            </div>
29        </div>
30 
31    <div class="row">
32        <div class="col-sm-12">
33 
34            @include('admin.partials.errors')
35            @include('admin.partials.success')
36 
37            <table id="uploads-table" class="table table-striped table-bordered">
38                <thead>
39                    <tr>
40                        <th>Name</th>
41                        <th>Type</th>

```

```

42         <th>Date</th>
43         <th>Size</th>
44         <th data-sortable="false">Actions</th>
45     </tr>
46 </thead>
47 <tbody>
48
49 {{-- The Subfolders --}}
50 @foreach ($subfolders as $path => $name)
51     <tr>
52         <td>
53             <a href="/admin/upload?folder={{ $path }}">
54                 <i class="fa fa-folder fa-lg fa-fw"></i>
55                 {{ $name }}
56             </a>
57         </td>
58         <td>Folder</td>
59         <td>-</td>
60         <td>-</td>
61         <td>
62             <button type="button" class="btn btn-xs btn-danger"
63                 onclick="delete_folder('{{ $name }}')">
64                 <i class="fa fa-times-circle fa-lg"></i>
65                 Delete
66             </button>
67         </td>
68     </tr>
69 @endforeach
70
71 {{-- The Files --}}
72 @foreach ($files as $file)
73     <tr>
74         <td>
75             <a href="{{ $file['webPath'] }}">
76                 @if (is_image($file['mimeType']))
77                     <i class="fa fa-file-image-o fa-lg fa-fw"></i>
78                 @else
79                     <i class="fa fa-file-o fa-lg fa-fw"></i>
80                 @endif
81                 {{ $file['name'] }}
82             </a>
83         </td>

```

```

84     <td>{{ $file['mimeType'] or 'Unknown' }}</td>
85     <td>{{ $file['modified']->format('j-M-y g:ia') }}</td>
86     <td>{{ human_filesize($file['size']) }}</td>
87     <td>
88         <button type="button" class="btn btn-xs btn-danger"
89             onclick="delete_file('{{ $file['name'] }}')">
90             <i class="fa fa-times-circle fa-lg"></i>
91             Delete
92         </button>
93         @if (is_image($file['mimeType']))
94             <button type="button" class="btn btn-xs btn-success"
95                 onclick="preview_image('{{ $file['webPath'] }}')">
96                 <i class="fa fa-eye fa-lg"></i>
97                 Preview
98             </button>
99         @endif
100     </td>
101 </tr>
102 @endforeach
103
104     </tbody>
105 </table>
106
107 </div>
108 </div>
109 </div>
110
111 @include('admin.upload._modals')
112
113 @stop
114
115 @section('scripts')
116     <script>
117
118         // Confirm file delete
119         function delete_file(name) {
120             $("#delete-file-name1").html(name);
121             $("#delete-file-name2").val(name);
122             $("#modal-file-delete").modal("show");
123         }
124
125         // Confirm folder delete

```

```
126     function delete_folder(name) {
127         $("#delete-folder-name1").html(name);
128         $("#delete-folder-name2").val(name);
129         $("#modal-folder-delete").modal("show");
130     }
131
132     // Preview image
133     function preview_image(path) {
134         $("#preview-image").attr("src", path);
135         $("#modal-image-view").modal("show");
136     }
137
138     // Startup code
139     $(function() {
140         $("#uploads-table").DataTable();
141     });
142 </script>
143 @stop
```

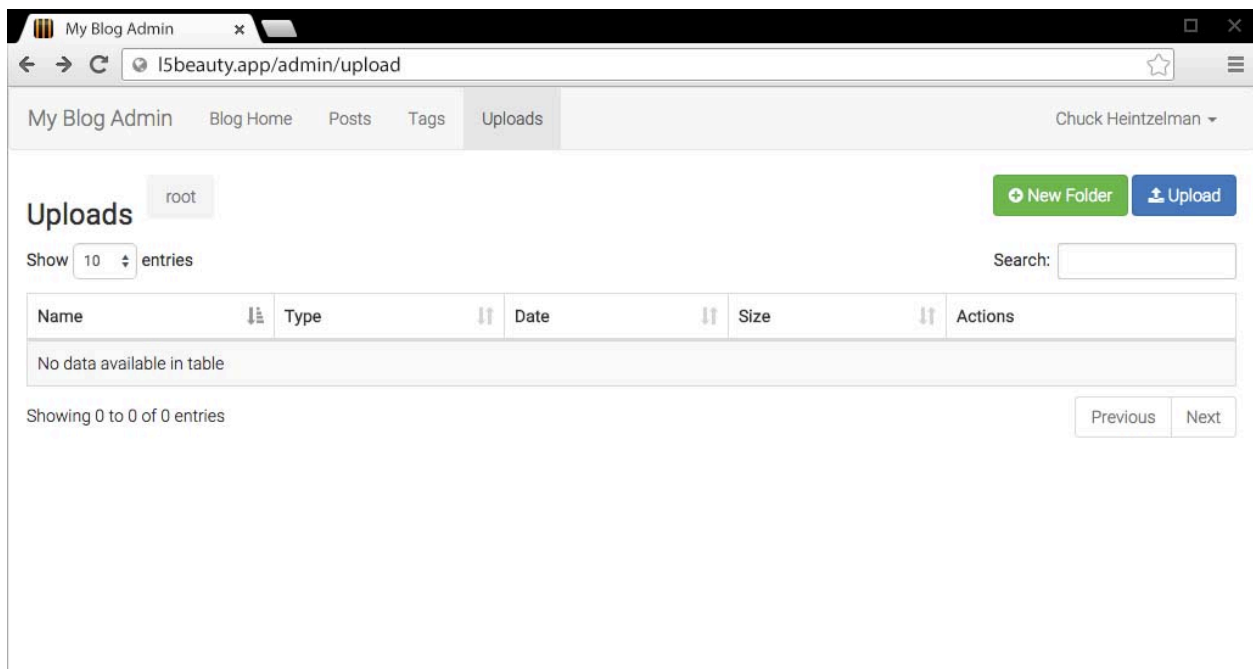
Although this template is long, it is easy to follow. All of the management of file uploading and downloading will be handled here.

Did you notice the inclusion of the `admin.upload._modals` view toward the end? I broke out the modal dialogs into a separate template.

For now, create an empty `_modals.blade.php` in the `resources/views/admin/upload` directory.

The Upload Manager Screen

Load up your browser, logging into the administration area and click on the **Uploads** link in the navigation bar. The screen should look like the one below.



Upload Manager Screen

Nice and clean. Now let's implement all the modal dialogs and the functionality behind them.

Finishing the Upload Manager

There's actually not a lot left to the upload manager. Let's take it one area at a time and complete all the functionality.

Updating the routes

We need to finish setting up all needed routes for the Upload Manager. Edit `app/Http/routes.php` making the changes below.

Changes to routes.php

```

1 // After the line that reads
2 get('admin/upload', 'UploadController@index');
3
4 // Add the following routes
5 post('admin/upload/file', 'UploadController@uploadFile');
6 delete('admin/upload/file', 'UploadController@deleteFile');
7 post('admin/upload/folder', 'UploadController@createFolder');
8 delete('admin/upload/folder', 'UploadController@deleteFolder');
```

Adding all the Modal Dialogs

Edit that empty `_modals.blade.php` file we created a few minutes ago. (*It's in the `resources/views/admin/upload` directory.*) Make the content match what's below.

Content of `admin.upload._modals View`

```

1  {{-- Create Folder Modal --}}
2  <div class="modal fade" id="modal-folder-create">
3    <div class="modal-dialog">
4      <div class="modal-content">
5        <form method="POST" action="/admin/upload/folder"
6          class="form-horizontal">
7          <input type="hidden" name="_token" value="{{ csrf_token() }}">
8          <input type="hidden" name="folder" value="{{ $folder }}">
9          <div class="modal-header">
10             <button type="button" class="close" data-dismiss="modal">
11               &times;
12             </button>
13             <h4 class="modal-title">Create New Folder</h4>
14           </div>
15           <div class="modal-body">
16             <div class="form-group">
17               <label for="new_folder_name" class="col-sm-3 control-label">
18                 Folder Name
19               </label>
20               <div class="col-sm-8">
21                 <input type="text" id="new_folder_name" name="new_folder"
22                   class="form-control">
23               </div>
24             </div>
25           </div>
26           <div class="modal-footer">
27             <button type="button" class="btn btn-default" data-dismiss="modal">
28               Cancel
29             </button>
30             <button type="submit" class="btn btn-primary">
31               Create Folder
32             </button>
33           </div>
34         </form>
35       </div>
36     </div>

```

```

37 </div>
38
39 {{-- Delete File Modal --}}
40 <div class="modal fade" id="modal-file-delete">
41   <div class="modal-dialog">
42     <div class="modal-content">
43       <div class="modal-header">
44         <button type="button" class="close" data-dismiss="modal">
45           &times;
46         </button>
47         <h4 class="modal-title">Please Confirm</h4>
48       </div>
49       <div class="modal-body">
50         <p class="lead">
51           <i class="fa fa-question-circle fa-lg"></i> &nbsp;
52           Are you sure you want to delete the
53           <kbd><span id="delete-file-name1">file</span></kbd>
54           file?
55         </p>
56       </div>
57       <div class="modal-footer">
58         <form method="POST" action="/admin/upload/file">
59           <input type="hidden" name="_token" value="{{ csrf_token() }}">
60           <input type="hidden" name="_method" value="DELETE">
61           <input type="hidden" name="folder" value="{{ $folder }}">
62           <input type="hidden" name="del_file" id="delete-file-name2">
63           <button type="button" class="btn btn-default" data-dismiss="modal">
64             Cancel
65           </button>
66           <button type="submit" class="btn btn-danger">
67             Delete File
68           </button>
69         </form>
70       </div>
71     </div>
72   </div>
73 </div>
74
75 {{-- Delete Folder Modal --}}
76 <div class="modal fade" id="modal-folder-delete">
77   <div class="modal-dialog">
78     <div class="modal-content">

```

```

79     <div class="modal-header">
80         <button type="button" class="close" data-dismiss="modal">
81             &times;
82         </button>
83         <h4 class="modal-title">Please Confirm</h4>
84     </div>
85     <div class="modal-body">
86         <p class="lead">
87             <i class="fa fa-question-circle fa-lg"></i> &nbsp;
88             Are you sure you want to delete the
89             <kbd><span id="delete-folder-name1">folder</span></kbd>
90             folder?
91         </p>
92     </div>
93     <div class="modal-footer">
94         <form method="POST" action="/admin/upload/folder">
95             <input type="hidden" name="_token" value="{ { csrf_token() } }">
96             <input type="hidden" name="_method" value="DELETE">
97             <input type="hidden" name="folder" value="{ { $folder } }">
98             <input type="hidden" name="del_folder" id="delete-folder-name2">
99             <button type="button" class="btn btn-default" data-dismiss="modal">
100                 Cancel
101             </button>
102             <button type="submit" class="btn btn-danger">
103                 Delete Folder
104             </button>
105         </form>
106     </div>
107 </div>
108 </div>
109 </div>
110
111 {{-- Upload File Modal --}}
112 <div class="modal fade" id="modal-file-upload">
113     <div class="modal-dialog">
114         <div class="modal-content">
115             <form method="POST" action="/admin/upload/file"
116                 class="form-horizontal" enctype="multipart/form-data">
117                 <input type="hidden" name="_token" value="{ { csrf_token() } }">
118                 <input type="hidden" name="folder" value="{ { $folder } }">
119                 <div class="modal-header">
120                     <button type="button" class="close" data-dismiss="modal">

```

```

121         &times;
122     </button>
123     <h4 class="modal-title">Upload New File</h4>
124 </div>
125 <div class="modal-body">
126     <div class="form-group">
127         <label for="file" class="col-sm-3 control-label">
128             File
129         </label>
130         <div class="col-sm-8">
131             <input type="file" id="file" name="file">
132         </div>
133     </div>
134     <div class="form-group">
135         <label for="file_name" class="col-sm-3 control-label">
136             Optional Filename
137         </label>
138         <div class="col-sm-4">
139             <input type="text" id="file_name" name="file_name"
140                 class="form-control">
141         </div>
142     </div>
143 </div>
144 <div class="modal-footer">
145     <button type="button" class="btn btn-default" data-dismiss="modal">
146         Cancel
147     </button>
148     <button type="submit" class="btn btn-primary">
149         Upload File
150     </button>
151 </div>
152 </form>
153 </div>
154 </div>
155 </div>
156
157 {{-- View Image Modal --}}
158 <div class="modal fade" id="modal-image-view">
159     <div class="modal-dialog">
160         <div class="modal-content">
161             <div class="modal-header">
162                 <button type="button" class="close" data-dismiss="modal">

```

```

163         &times;
164     </button>
165     <h4 class="modal-title">Image Preview</h4>
166 </div>
167 <div class="modal-body">
168     
169 </div>
170 <div class="modal-footer">
171     <button type="button" class="btn btn-default" data-dismiss="modal">
172         Cancel
173     </button>
174 </div>
175 </div>
176 </div>
177 </div>

```

There's five different modal popup boxes in that file. Each one will do the appropriate POST or DELETE to the routes just set up.

Adding the Request classes

Create the `UploadFileRequest` and `UploadNewFolderRequest` with the content below.

You can manually create these files or use the artisan `make:request ClassName` command to create a blank file in the `app/Http/Requests` directory.

First the Form Request class to handle uploaded files.

Content of `UploadFileRequest.php`

```

1 <?php
2 namespace App\Http\Requests;
3
4 class UploadFileRequest extends Request
5 {
6     /**
7      * Determine if the user is authorized to make this request.
8      *
9      * @return bool
10     */
11     public function authorize()
12     {
13         return true;

```

```
14     }
15
16     /**
17      * Get the validation rules that apply to the request.
18      *
19      * @return array
20      */
21     public function rules()
22     {
23         return [
24             'file' => 'required',
25             'folder' => 'required',
26         ];
27     }
28 }
```

Then the Form Request class to handle validating requests to create new folders.

Content of UploadNewFolderRequest.php

```
1 <?php
2 namespace App\Http\Requests;
3
4 class UploadNewFolderRequest extends Request
5 {
6     /**
7      * Determine if the user is authorized to make this request.
8      *
9      * @return bool
10     */
11     public function authorize()
12     {
13         return true;
14     }
15
16     /**
17      * Get the validation rules that apply to the request.
18      *
19      * @return array
20     */
21     public function rules()
22     {
23         return [
```

```

24         'folder' => 'required',
25         'new_folder' => 'required',
26     ];
27 }
28 }

```

Again, these are very simple classes that validate the input on construction.

Finishing UploadController

Update the `UploadController.php` file as instructed below.

Updates to UploadController

```

1  <?php
2  // Add the following 3 lines at the top, with the use statements
3  use App\Http\Requests\UploadFileRequest;
4  use App\Http\Requests\UploadNewFolderRequest;
5  use Illuminate\Support\Facades\File;
6
7  // Add the following 4 methods to the UploadControllerClass
8  /**
9   * Create a new folder
10  */
11  public function createFolder(UploadNewFolderRequest $request)
12  {
13      $new_folder = $request->get('new_folder');
14      $folder = $request->get('folder').'/'. $new_folder;
15
16      $result = $this->manager->createDirectory($folder);
17
18      if ($result === true) {
19          return redirect()
20              ->back()
21              ->withSuccess("Folder '$new_folder' created.");
22      }
23
24      $error = $result ? : "An error occurred creating directory.";
25      return redirect()
26          ->back()
27          ->withErrors([$error]);
28  }

```



```
29
30  /**
31   * Delete a file
32   */
33  public function deleteFile(Request $request)
34  {
35      $del_file = $request->get('del_file');
36      $path = $request->get('folder').'/'. $del_file;
37
38      $result = $this->manager->deleteFile($path);
39
40      if ($result === true) {
41          return redirect()
42              ->back()
43              ->withSuccess("File '$del_file' deleted.");
44      }
45
46      $error = $result ? : "An error occurred deleting file.";
47      return redirect()
48          ->back()
49          ->withErrors([$error]);
50  }
51
52  /**
53   * Delete a folder
54   */
55  public function deleteFolder(Request $request)
56  {
57      $del_folder = $request->get('del_folder');
58      $folder = $request->get('folder').'/'. $del_folder;
59
60      $result = $this->manager->deleteDirectory($folder);
61
62      if ($result === true) {
63          return redirect()
64              ->back()
65              ->withSuccess("Folder '$del_folder' deleted.");
66      }
67
68      $error = $result ? : "An error occurred deleting directory.";
69      return redirect()
70          ->back()
```

```
71         ->withErrors([$error]);
72     }
73
74     /**
75      * Upload new file
76      */
77     public function uploadFile(UploadFileRequest $request)
78     {
79         $file = $_FILES['file'];
80         $fileName = $request->get('file_name');
81         $fileName = $fileName ?: $file['name'];
82         $path = str_finish($request->get('folder'), '/') . $fileName;
83         $content = File::get($file['tmp_name']);
84
85         $result = $this->manager->saveFile($path, $content);
86
87         if ($result === true) {
88             return redirect()
89                 ->back()
90                 ->withSuccess("File '$fileName' uploaded.");
91         }
92
93         $error = $result ? : "An error occurred uploading file.";
94         return redirect()
95             ->back()
96             ->withErrors([$error]);
97     }
```

I'm not going to comment much on those methods because this chapter's already getting long. Basically, we're calling methods on the UploadsManager class to do the actual file work.

Finishing the UploadsManager Service class

Update the UploadsManager class in the app/Services directory, making the changes below.

Updates to UploadsManager

```
1 <?php
2 // Add the 4 methods below to the class
3 /**
4  * Create a new directory
5  */
6 public function createDirectory($folder)
7 {
8     $folder = $this->cleanFolder($folder);
9
10    if ($this->disk->exists($folder)) {
11        return "Folder '$folder' already exists.";
12    }
13
14    return $this->disk->makeDirectory($folder);
15 }
16
17 /**
18  * Delete a directory
19  */
20 public function deleteDirectory($folder)
21 {
22     $folder = $this->cleanFolder($folder);
23
24     $filesFolders = array_merge(
25         $this->disk->directories($folder),
26         $this->disk->files($folder)
27     );
28     if (! empty($filesFolders)) {
29         return "Directory must be empty to delete it.";
30     }
31
32     return $this->disk->deleteDirectory($folder);
33 }
34
35 /**
36  * Delete a file
37  */
38 public function deleteFile($path)
39 {
40     $path = $this->cleanFolder($path);
41 }
```

```
42     if (! $this->disk->exists($path)) {
43         return "File does not exist.";
44     }
45
46     return $this->disk->delete($path);
47 }
48
49 /**
50  * Save a file
51  */
52 public function saveFile($path, $content)
53 {
54     $path = $this->cleanFolder($path);
55
56     if ($this->disk->exists($path)) {
57         return "File already exists.";
58     }
59
60     return $this->disk->put($path, $content);
61 }
```

Four new methods here, which implement the new functionality of the Upload Manager.



Congratulations

You now have completed the Upload Manager for your blog's administration area. Everything should work. Give it a try. Upload a few files, create a directory. Try uploading an image and you'll see an option to preview it.

Setting Up Your S3 Account

Using Amazon Simple Storage Service, also known as Amazon S3, is a cheap, fast and easy solution for storing files in *The Cloud*. With Laravel 5.1 it's easy to configure the Upload Manager created in this chapter to save and retrieve files using Amazon S3.

Converting to Amazon S3 is optional. If you don't want to do this, skip to the next chapter.

Here's a quick startup list to set up S3 services.

1. Log in (or sign up) with Amazon

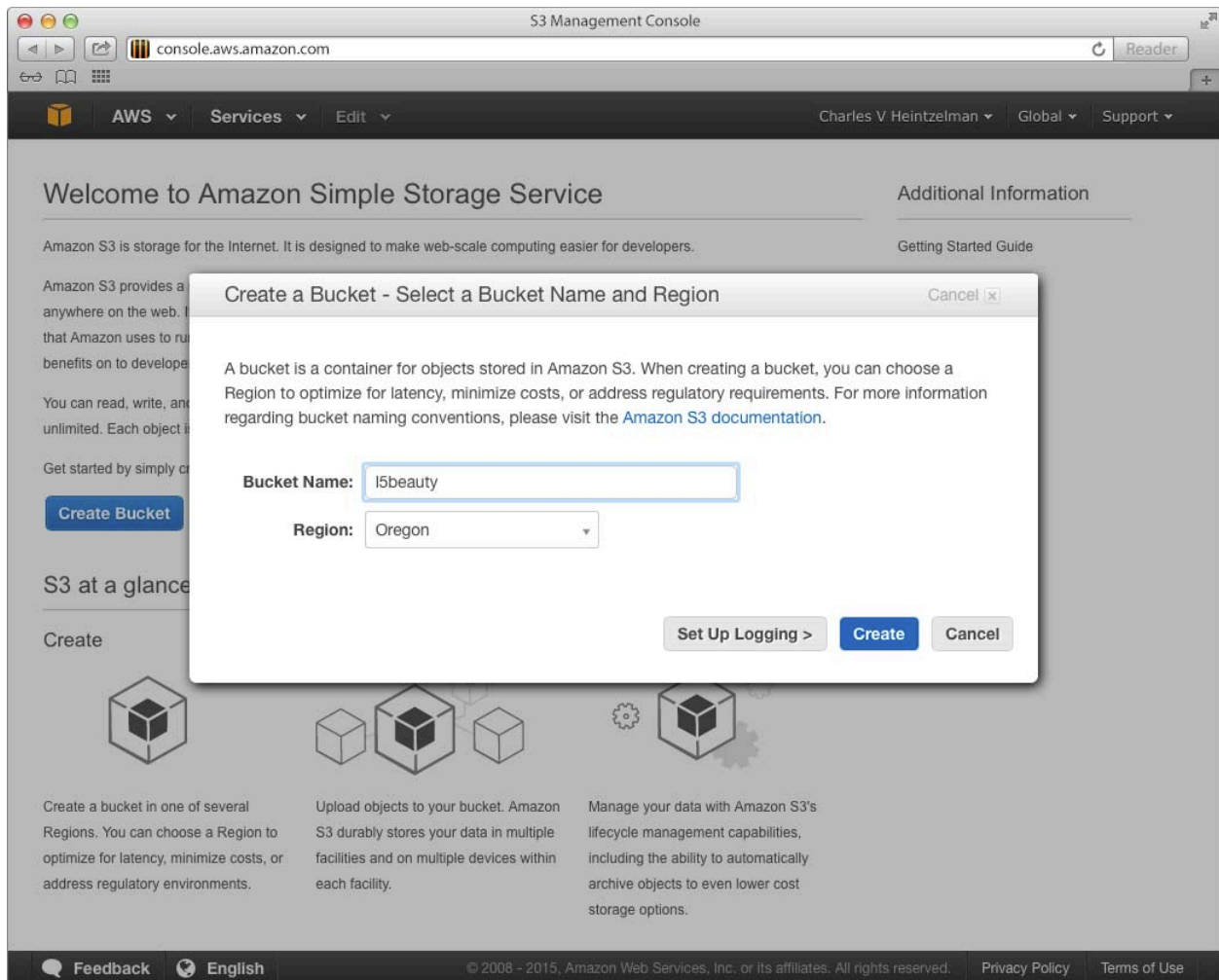
Go to `amazon.com` and log in if you have an existing account. If not, sign up for a new account..

2. Get a Web Services Account

Go to `aws.amazon.com` and sign up for a **Amazon Web Services Account**. You can sign up for a free account but you'll still need to enter a credit card.

3. Create a S3 Bucket

After you've created or logged into your AWS account, go to the **Console** and click on the **S3** link. If you've never created an S3 bucket before you'll be prompted to do so.



AWS Create Bucket

Here I'm using l5beauty as the bucket name and taking the default region of Oregon it provides for me. *(If you use a bucket name already in use Amazon will prompt you to pick a different name.)*

Event if you have created a bucket before, it's still a good idea to create a new one. Each project should have its own, unique bucket.

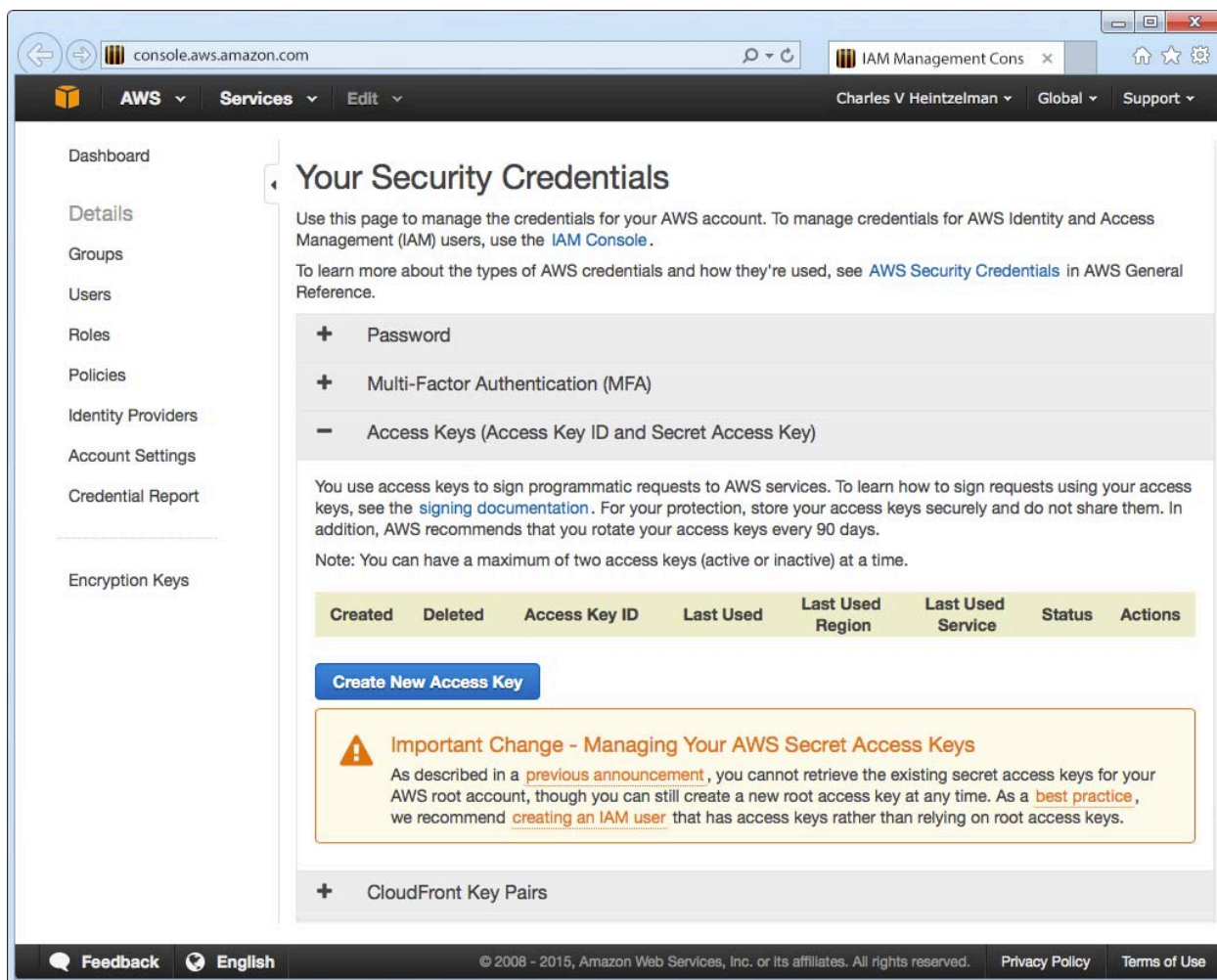


Remember this Bucket and Region

You'll need it later when we configure the application to use S3.

4. Create an Access Key

Click on your name at the top of the screen and then choose **Security Credentials**. This will present you with the Security Credentials screen. Click on the **Access Keys** area and you'll see a screen similar to the one below.



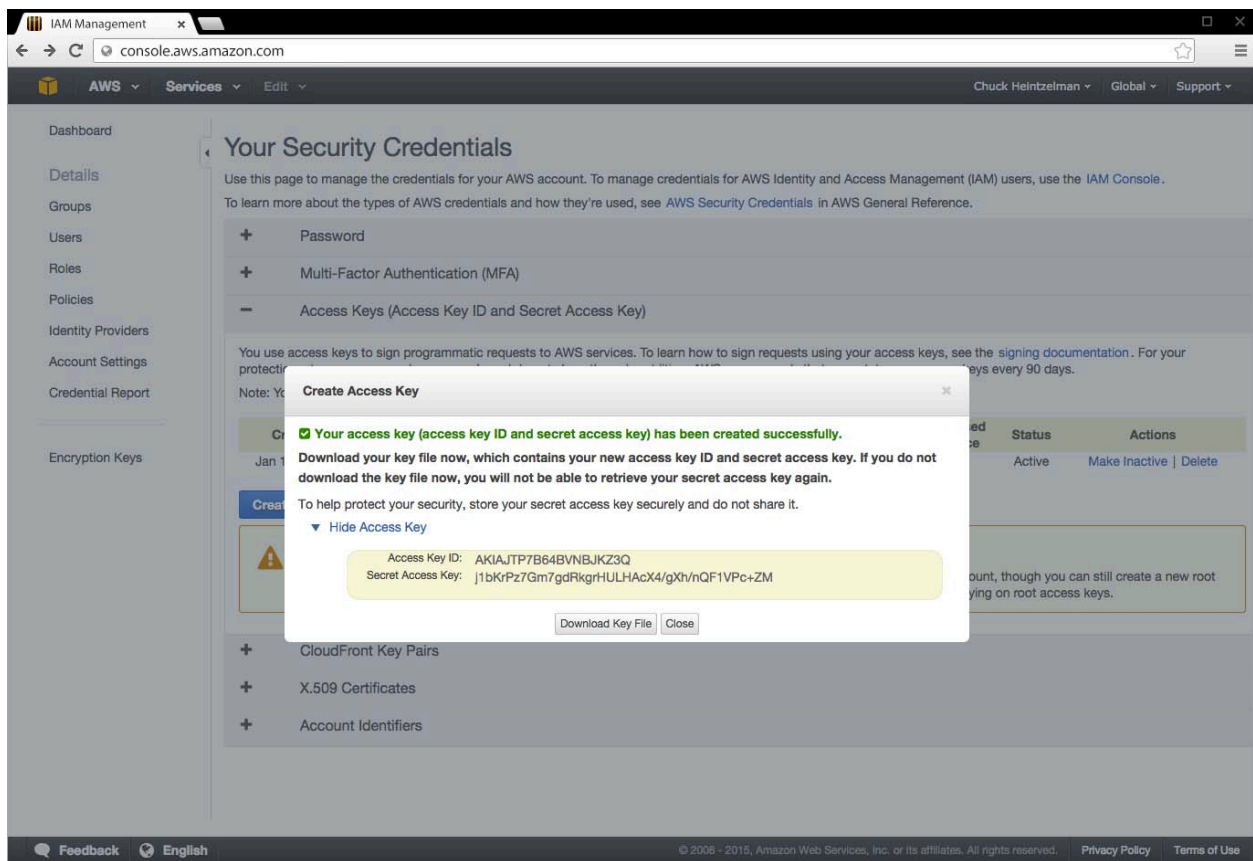
AWS Security Credentials

Click on the [Create New Access Key] button and you'll see a screen that follows.



Important

Write down the values from the screen below before closing the window. You may want to download the key file too. You won't have access to the secret key again.



Create New Access Key



S3 Is Now Set Up

For now. Later we'll make access public

Configuring L5Beauty to Use S3

Now that S3 services are set up, let's configure the blog to use them.

Use the table below to get the code for the region you used when creating your bucket.

Code	Name
ap-northeast-1	Asia Pacific (Tokyo)
ap-southeast-1	Asia Pacific (Singapore)
ap-southeast-2	Asia Pacific (Sydney)
eu-central-1	EU (Frankfurt)
eu-west-1	EU (Ireland)
sa-east-1	South America (Sao Paulo)
us-east-1	US East (N. Virginia)
us-west-1	US West (N. California)
us-west-2	US West (Oregon)

Edit the `.env` file, filling in the AWS parameters like below.

AWS paramters in `.env`

```
AWS_KEY=AKIAJTP7B64BVNBKZ3Q
AWS_SECRET=j1bKrPz7Gm7gdRkgrHULHAcX4/gXh/nQF1VPc+ZM
AWS_REGION=us-west-2
AWS_BUCKET=15beauty
```

USE YOUR OWN VALUES. (*I'll delete these values from my account shortly.*)

Edit the `config/blog.php` file as follows

Update of blog config

```
1 <?php
2 return [
3     'title' => 'My Blog',
4     'posts_per_page' => 5,
5     'uploads' => [
6         'storage' => 's3',
7         'webpath' => 'https://s3-us-west-2.amazonaws.com/15beauty',
8     ],
9 ];
```

The webpath follows the pattern: `https://s3-REGION.amazonaws.com/BUCKET/`. If this is incorrect, don't worry about it right now.

Installing an Additional Package

Laravel 5.1 requires an additional package when interfacing with S3. From the OS Console, install `flysystem-aws` as instructed below.

Requiring flysystem-aws

```
~/Code/l5beauty% composer require league/flysystem-aws-s3-v3
```

```
Using version ^1.0 for league/flysystem-aws-s3-v3
```

```
./composer.json has been updated
```

```
Loading composer repositories with package information
```

```
Updating dependencies (including require-dev)
```

```
- Installing mtdowling/jmespath.php (2.2.0)
```

```
  Downloading: 100%
```

```
- Installing guzzlehttp/promises (1.0.1)
```

```
  Loading from cache
```

```
- Installing psr/http-message (1.0)
```

```
  Loading from cache
```

```
- Installing guzzlehttp/psr7 (1.1.0)
```

```
  Downloading: 100%
```

```
- Installing guzzlehttp/guzzle (6.0.1)
```

```
  Loading from cache
```

```
- Installing aws/aws-sdk-php (3.0.6)
```

```
  Downloading: 100%
```

```
- Installing league/flysystem-aws-s3-v3 (1.0.3)
```

```
  Downloading: 100%
```

```
Writing lock file
```

```
Generating autoload files
```

```
Generating optimized class loader
```

Test The Upload Manager

Point your browser to your **l5beauty** project's administration area and go into uploads. Try creating directories, uploading files. Be sure to upload at least one image before moving on.

Also, verify from within your AWS Console that changes you make are reflected there.



Important

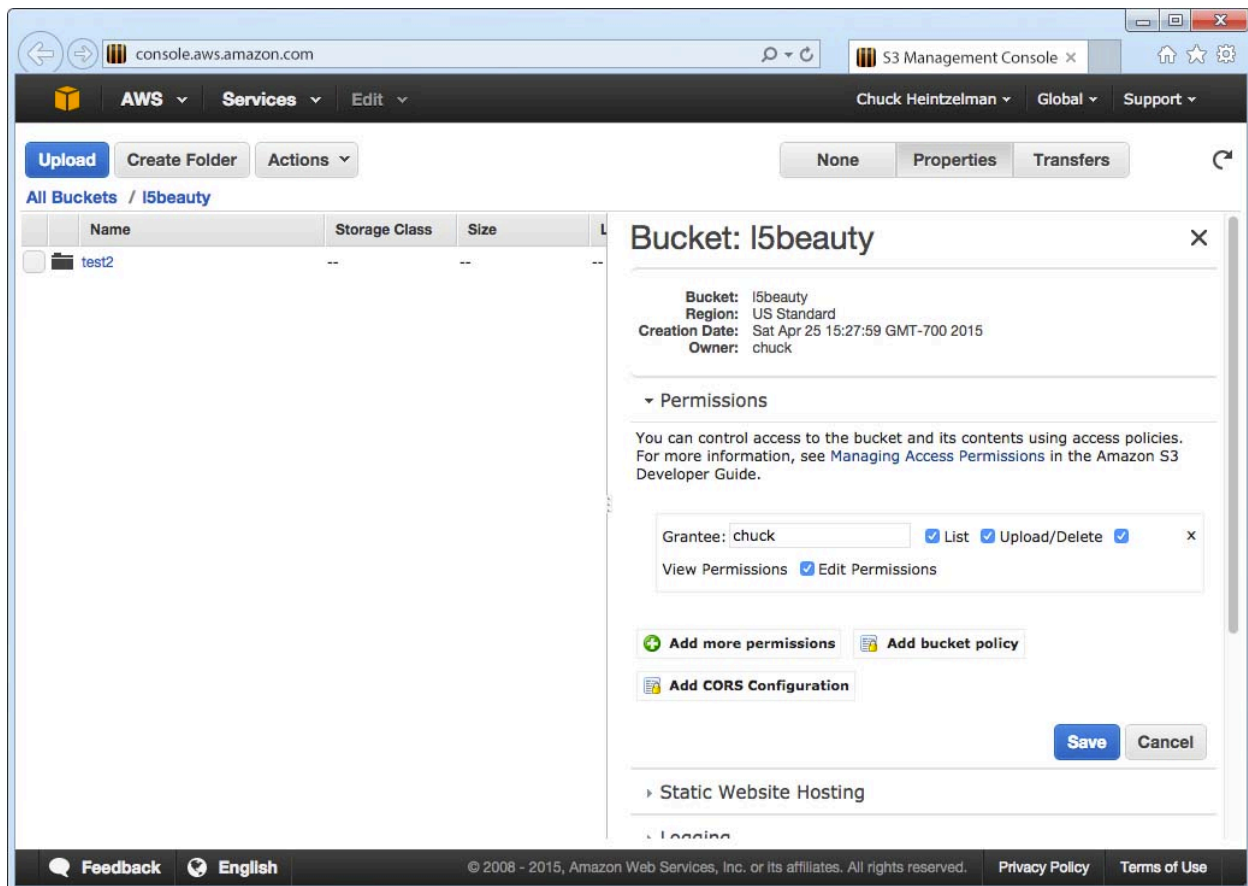
After uploading a file from the Upload Manager, view your bucket from the AWS console, click on the file and check the **Properties** tab. There will be a *Link*: value. Examine this URL and make any necessary changes to the `uploads.webpath` property of `config/blog.php`.

Fixing Bucket Permissions

Everything should work great from the administration area, but if a user visits your web page and you're displaying an image you've stored on S3, they won't have permissions to view the file.

To fix this we'll make everything on this bucket you created publicly viewable.

Go into the AWS Console and navigate to the bucket you just created. Then click on the **Properties** tab and click on the **Permissions** accordion to expand it. You should see a screen like the one below:



Bucket Permissions

Click on the **Add bucket policy** link and when the **Bucket Policy Editor** appears click on the **new policy** link.

The following screen will appear.

Step 1: Select Policy Type

A Policy is a container for permissions. The different types of policies you can create are an [IAM Policy](#), an [S3 Bucket Policy](#), an [SNS Topic Policy](#) and an [SQS Queue Policy](#).

Select Type of Policy S3 Bucket Policy

Step 2: Add Statement(s)

A statement is the formal description of a single permission. See a [description of elements](#) that you can use in statements.

Effect ☒ Allow ☐ Deny

Principal

Use a comma to separate multiple values.

AWS Service Amazon S3 ☐ All Services ('*')

Use multiple statements to add permissions for more than one service.

Actions 1 Action(s) Selected ☐ All Actions ('*')

Amazon Resource Name (ARN) arn:aws:s3:::5beauty/*

ARN should follow the following format: arn:aws:s3:::<bucket_name>/<key_name>.
Use a comma to separate multiple values.

[Add Conditions \(Optional\)](#)

Add Statement

Step 3: Generate Policy

A *policy* is a document (written in the [Access Policy Language](#)) that acts as a container for one or more statements.

Add one or more statements above to generate a policy.

This AWS Policy Generator is provided for informational purposes only, you are still responsible for your use of Amazon Web Services technologies and ensuring that your use is in compliance with all applicable terms and conditions. This AWS Policy Generator is provided as is without warranty of any kind, whether express, implied, or statutory. This AWS Policy Generator does not modify the applicable terms and conditions governing your use of Amazon Web Services technologies.

AWS Policy Generator

The *Actions* selected should be `GetObject`.

Click the **[Add Statement]** button and then the **[Generate Policy]** button and you'll see a policy like the one below.

Policy JSON Document

```
{
  "Id": "Policy1430071323597",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1430071313897",
      "Action": [
        "s3:GetObject"
      ]
    }
  ]
}
```

```
    ],  
    "Effect": "Allow",  
    "Resource": "arn:aws:s3:::l5beauty/*",  
    "Principal": "*"   
  }  
]  
}
```

Copy this text and paste it into the previous screen's **Bucket Policy Editor** box, click **Save** and you're done.



Congratulations

Your Upload Manager should now work, allowing you to manage any files uploaded for your blog in the cloud.

Recap

In this chapter we configured the file system and built an upload manager that allowed us to upload files to the public area of the **L5 Beauty** application. We added a `helpers.php` file to the application as a place to store one-off functions.

A couple new packages were pulled in. One to detect mime-types and one required to access the Amazon S3 service.

Instructions were provided to set up and configure the Amazon S3 service. And finally, we configured our Upload Manager to use this.

Quite a bit accomplished in this chapter.

Chapter 12 - Posts Administration

In this chapter we'll finish the post functionality in the blog's administration area. This includes modifying the structure of the posts table with a new migration, pulling in some additional assets, and adding the basic Create, Update, and Delete methods.

Chapter Contents

- [Modifying the Posts table](#)
 - [Requiring Doctrine](#)
 - [Creating the Migration](#)
 - [Running Migrations](#)
- [Updating the Models](#)
- [Adding Selectize.js and Pickadate.js](#)
 - [Pulling them in with Bower](#)
 - [Managing them with Gulp](#)
- [Creating the Request Classes](#)
- [Creating the PostFormFields Job](#)
- [Adding to helpers.php](#)
- [Updating the Post Model](#)
- [Updating the Controller](#)
- [The Post Views](#)
- [Removing the show route](#)
- [Recap](#)

Modifying the Posts table

The nice thing about database migrations in Laravel 5.1 is that they're like version control for your database. Right now the version of the posts table still exists from when we created the **10 Minute Blog** a few chapters back.

Let's make some changes to this table.

Requiring Doctrine

In Laravel 5.1 whenever columns in a database need to be modified, the Doctrine package is required. Use composer to install the doctrine/dbal package as instructed below.

Installing doctrine/dbal

```
vagrant@homestead:~/Code/15beauty$ composer require "doctrine/dbal"
Using version ^2.5 for doctrine/dbal
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing doctrine/lexer (v1.0.1)
  Downloading: 100%
```

[snip]

```
Writing lock file
Generating autoload files
Generating optimized class loader
```

Composer makes adding PHP packages very easy.

Creating the Migration

Create the migration skeleton with the artisan command in the Homestead VM.

Creating restructure_posts_table migration

```
~/Code/15beauty$ php artisan make:migration --table=posts \
    restructure_posts_table
Created Migration: 2015_04_28_052603_restructure_posts_table
```

Then edit the newly created migration to match what's below.

The Restructure Posts Table Migration

```
1 <?php
2
3 use Illuminate\Database\Schema\Blueprint;
4 use Illuminate\Database\Migrations\Migration;
5
6 class RestructurePostsTable extends Migration
7 {
8     /**
9      * Run the migrations.
10     */
```

```

11  public function up()
12  {
13      Schema::table('posts', function (Blueprint $table) {
14          $table->string('subtitle')->after('title');
15          $table->renameColumn('content', 'content_raw');
16          $table->text('content_html')->after('content');
17          $table->string('page_image')->after('content_html');
18          $table->string('meta_description')->after('page_image');
19          $table->boolean('is_draft')->after('meta_description');
20          $table->string('layout')->after('is_draft')
21              ->default('blog.layouts.post');
22      });
23  }
24
25  /**
26   * Reverse the migrations.
27   */
28  public function down()
29  {
30      Schema::table('posts', function (Blueprint $table) {
31          $table->dropColumn('layout');
32          $table->dropColumn('is_draft');
33          $table->dropColumn('meta_description');
34          $table->dropColumn('page_image');
35          $table->dropColumn('content_html');
36          $table->renameColumn('content_raw', 'content');
37          $table->dropColumn('subtitle');
38      });
39  }
40 }

```

Here's a quick breakdown of what changed and why.

subtitle

Our final blog will also require a subtitle.

content_html and content_raw

We'll do our editing in Markdown, but whenever content is saved we also want to save the HTML version.

page_image

Like the tags table, each post will allow a unique header image at the top.

meta_description

To populate the META tags on the post's page for search engines.

is_draft

We'll add a flag on each post so as to not inadvertently publish a *draft* post.

layout

This will give us the ability to use unique layouts on the posts.

Running Migrations

Now that the migration is created, run migrations from the Homestead VM.

Running Migrations

```
vagrant@homestead:~/Code/15beauty$ php artisan migrate
Migrated: 2015_04_28_052603_restructure_posts_table
```

Now the work on the database is complete!

Updating the Models

Let's update the Post model and the Tag model to establish the relationships between the two.

Update the app\Tag.php file with the content below.

The new Tag model

```
1 <?php
2 namespace App;
3
4 use Illuminate\Database\Eloquent\Model;
5
6 class Tag extends Model
7 {
8     protected $fillable = [
9         'tag', 'title', 'subtitle', 'page_image', 'meta_description',
10        'reverse_direction',
11    ];
12
13    /**
14     * The many-to-many relationship between tags and posts.
15     *
```



```

16     * @return BelongsToMany
17     */
18     public function posts()
19     {
20         return $this->belongsToMany('App\Post', 'post_tag_pivot');
21     }
22
23     /**
24      * Add any tags needed from the list
25      *
26      * @param array $tags List of tags to check/add
27      */
28     public static function addNeededTags(array $tags)
29     {
30         if (count($tags) === 0) {
31             return;
32         }
33
34         $found = static::whereIn('tag', $tags)->lists('tag')->all();
35
36         foreach (array_diff($tags, $found) as $tag) {
37             static::create([
38                 'tag' => $tag,
39                 'title' => $tag,
40                 'subtitle' => 'Subtitle for '.$tag,
41                 'page_image' => '',
42                 'meta_description' => '',
43                 'reverse_direction' => false,
44             ]);
45         }
46     }
47 }

```

\$fillable

Here we set the name of the columns that can be filled with an array. The `addNeededTags()` method will use this.

posts()

The many-to-many relationship between posts and tags.

addNeededTags()

A static function to add tags that aren't in the database already.



In the `posts()` method we're only passing two arguments to `belongsToMany()`. The first argument is the name of the model class. The second argument is the name of the table to use. The next two arguments are the *foreignKey* and the *otherKey*, but since we're using `post_id` and `tag_id`, the additional arguments to `belongsToMany()` can be omitted. (*Laravel is smart enough to figure them out.*)



Mass Assignment Protection

Laravel 5.1's models are built with *Mass Assignment Protection*. This means the model's `create()` method takes an array of column names and values, but only allows assignment of those columns that are in a white list. (The white list is the `$fillable` attribute).

Now update the `app\Post.php` file with the content below.

The new Post model

```

1  <?php
2  namespace App;
3
4  use App\Services\Markdowner;
5  use Illuminate\Database\Eloquent\Model;
6
7  class Post extends Model
8  {
9      protected $dates = ['published_at'];
10
11      /**
12       * The many-to-many relationship between posts and tags.
13       *
14       * @return BelongsToMany
15       */
16      public function tags()
17      {
18          return $this->belongsToMany('App\Tag', 'post_tag_pivot');
19      }
20
21      /**
22       * Set the title attribute and automatically the slug
23       *
24       * @param string $value
25       */
26      public function setTitleAttribute($value)

```

```

27     {
28         $this->attributes['title'] = $value;
29
30         if (! $this->exists) {
31             $this->setUniqueSlug($value, '');
32         }
33     }
34
35     /**
36      * Recursive routine to set a unique slug
37      *
38      * @param string $title
39      * @param mixed $extra
40      */
41     protected function setUniqueSlug($title, $extra)
42     {
43         $slug = str_slug($title.'-'. $extra);
44
45         if (static::whereSlug($slug)->exists()) {
46             $this->setUniqueSlug($title, $extra + 1);
47             return;
48         }
49
50         $this->attributes['slug'] = $slug;
51     }
52
53     /**
54      * Set the HTML content automatically when the raw content is set
55      *
56      * @param string $value
57      */
58     public function setContentRawAttribute($value)
59     {
60         $markdown = new Markdowner();
61
62         $this->attributes['content_raw'] = $value;
63         $this->attributes['content_html'] = $markdown->toHTML($value);
64     }
65
66     /**
67      * Sync tag relation adding new tags as needed
68      *

```

```

69      * @param array $tags
70      */
71      public function syncTags(array $tags)
72      {
73          Tag::addNeededTags($tags);
74
75          if (count($tags)) {
76              $this->tags()->sync(
77                  Tag::whereIn('tag', $tags)->lists('id')->all()
78              );
79              return;
80          }
81
82          $this->tags()->detach();
83      }
84  }

```

tags()

Similar to the posts() method in the Tag model, but here we're going the other way.

setTitleAttribute()

We changed this from the previous version to call the setUniqueSlug() method.

setUniqueSlug()

A function that recurses to set the unique slug when needed.

setContentRawAttribute()

Now when the content_raw value is set in the model, it will be automatically be converted to HTML and assigned to the content_html attribute.

syncTags()

Synchronizes the tags with the post.

Adding Selectize.js and Pickadate.js

Let's add two additional assets to our system. We'll use **bower** to pull the resources in and **gulp** to put them where we want them.

Pulling them in with Bower

The first is Selectize.js. This library is handy for setting up tagging and we'll use to to assign zero or more *Tags* to each *Post*. Follow the instructions below to pull it in with bower. (*I'm doing this from the Host OS, but you should also be able to do this within Homestead VM.*)

Adding Selectize.js with Bower

```
~/Code/15beauty% bower install selectize --save
bower not-cached    git://github.com/brianreavis/selectize.js.git**
bower resolve       git://github.com/brianreavis/selectize.js.git**
```

[snip]

```
selectize#0.12.1 vendor/bower_dl/selectize
├─ jquery#2.1.4
├─ microplugin#0.0.3
└─ sifter#0.4.1
```

Then let's pull in Pickadate.js. There's no shortage of libraries for picking dates and times, but I wanted to use this one because it works slick on small devices. Follow the instructions below to pull in Pickadate.js.

Adding Pickadate.js with Bower

```
~/Code/15beauty% bower install pickadate --save
bower not-cached    git://github.com/amsul/pickadate.js.git**
bower resolve       git://github.com/amsul/pickadate.js.git**
bower download      https://github.com/amsul/pickadate.js/archive/3.5.6.tar.gz
bower extract       pickadate** archive.tar.gz
bower resolved      git://github.com/amsul/pickadate.js.git#3.5.6
bower install       pickadate#3.5.6
```

```
pickadate#3.5.6 vendor/bower_dl/pickadate
└─ jquery#2.1.4
```

Managing them with Gulp

Now that these libraries are downloaded, update the `gulpfile.js` to match what's below.

Updated gulpfile.js

```
1  var gulp = require('gulp');
2  var rename = require('gulp-rename');
3  var elixir = require('laravel-elixir');
4
5  /**
6   * Copy any needed files.
7   *
8   * Do a 'gulp copyfiles' after bower updates
9   */
10 gulp.task("copyfiles", function() {
11
12     // Copy jQuery, Bootstrap, and FontAwesome
13     gulp.src("vendor/bower_dl/jquery/dist/jquery.js")
14         .pipe(gulp.dest("resources/assets/js/"));
15
16     gulp.src("vendor/bower_dl/bootstrap/less/**")
17         .pipe(gulp.dest("resources/assets/less/bootstrap"));
18
19     gulp.src("vendor/bower_dl/bootstrap/dist/js/bootstrap.js")
20         .pipe(gulp.dest("resources/assets/js/"));
21
22     gulp.src("vendor/bower_dl/bootstrap/dist/fonts/**")
23         .pipe(gulp.dest("public/assets/fonts"));
24
25     gulp.src("vendor/bower_dl/fontawesome/less/**")
26         .pipe(gulp.dest("resources/assets/less/fontawesome"));
27
28     gulp.src("vendor/bower_dl/fontawesome/fonts/**")
29         .pipe(gulp.dest("public/assets/fonts"));
30
31     // Copy datatables
32     var dtDir = 'vendor/bower_dl/datatables-plugins/integration/';
33
34     gulp.src("vendor/bower_dl/datatables/media/js/jquery.dataTables.js")
35         .pipe(gulp.dest('resources/assets/js/'));
36
37     gulp.src(dtDir + 'bootstrap/3/dataTables.bootstrap.css')
38         .pipe(rename('dataTables.bootstrap.less'))
39         .pipe(gulp.dest('resources/assets/less/others/'));
40
41     gulp.src(dtDir + 'bootstrap/3/dataTables.bootstrap.js')
```

```
42     .pipe(gulp.dest('resources/assets/js/'));
43
44     // Copy selectize
45     gulp.src("vendor/bower_dl/selectize/dist/css/**")
46         .pipe(gulp.dest("public/assets/selectize/css"));
47
48     gulp.src("vendor/bower_dl/selectize/dist/js/standalone/selectize.min.js")
49         .pipe(gulp.dest("public/assets/selectize/"));
50
51     // Copy pickadate
52     gulp.src("vendor/bower_dl/pickadate/lib/compressed/themes/**")
53         .pipe(gulp.dest("public/assets/pickadate/themes/"));
54
55     gulp.src("vendor/bower_dl/pickadate/lib/compressed/picker.js")
56         .pipe(gulp.dest("public/assets/pickadate/"));
57
58     gulp.src("vendor/bower_dl/pickadate/lib/compressed/picker.date.js")
59         .pipe(gulp.dest("public/assets/pickadate/"));
60
61     gulp.src("vendor/bower_dl/pickadate/lib/compressed/picker.time.js")
62         .pipe(gulp.dest("public/assets/pickadate/"));
63
64 });
65
66 /**
67  * Default gulp is to run this elixir stuff
68  */
69 elixir(function(mix) {
70
71     // Combine scripts
72     mix.scripts([
73         'js/jquery.js',
74         'js/bootstrap.js',
75         'js/jquery.dataTables.js',
76         'js/dataTables.bootstrap.js'
77     ],
78     'public/assets/js/admin.js',
79     'resources/assets'
80 );
81
82 // Compile Less
83 mix.less('admin.less', 'public/assets/css/admin.css');
```

84 `});`

This configuration is basically the same as before, but with the needed files from Selectize.js and Pickadate.js copied into the public asset directory.

Run `gulp copyfiles` to do the copying.

Running gulp copyfiles

```
~/Code/15beauty% gulp copyfiles
gulp copyfiles
[21:10:15] Using gulpfile ~/Projects/15beauty/gulpfile.js
[21:10:15] Starting 'copyfiles'...
[21:10:15] Finished 'copyfiles' after 48 ms
```

Creating the Request Classes

Just like we did in Chapter 10 with the Tags, we'll use Request classes to validate the create and update Post requests.

First create the requests using artisan in the Homestead VM. This will create skeletons for each of these classes in the `app/Http/Requests` directory.

Creating the Request Class Skeletons

```
~/Code/15beauty$ php artisan make:request PostCreateRequest
Request created successfully.
~/Code/15beauty$ php artisan make:request PostUpdateRequest
Request created successfully.
```

Update the newly created `PostCreateRequest.php` to match what's below.

Content of PostCreateRequest.php

```
1 <?php
2 namespace App\Http\Requests;
3
4 use Carbon\Carbon;
5
6 class PostCreateRequest extends Request
7 {
8     /**
9      * Determine if the user is authorized to make this request.
```



```
10     */
11     public function authorize()
12     {
13         return true;
14     }
15
16     /**
17      * Get the validation rules that apply to the request.
18      *
19      * @return array
20      */
21     public function rules()
22     {
23         return [
24             'title' => 'required',
25             'subtitle' => 'required',
26             'content' => 'required',
27             'publish_date' => 'required',
28             'publish_time' => 'required',
29             'layout' => 'required',
30         ];
31     }
32
33     /**
34      * Return the fields and values to create a new post from
35      */
36     public function postFillData()
37     {
38         $published_at = new Carbon(
39             $this->publish_date.' '.$this->publish_time
40         );
41         return [
42             'title' => $this->title,
43             'subtitle' => $this->subtitle,
44             'page_image' => $this->page_image,
45             'content_raw' => $this->get('content'),
46             'meta_description' => $this->meta_description,
47             'is_draft' => (bool)$this->is_draft,
48             'published_at' => $published_at,
49             'layout' => $this->layout,
50         ];
51     }
```

52 }

This is a standard request with the `authorize()` and `rules()` methods, but we're also adding the `postFillData()` to make it easy to pull all the data from the request to fill a new `Post` model with.

And update the `PostUpdateRequest.php` to match the following:

Content of `PostUpdateRequest.php`

```

1 <?php
2 namespace App\Http\Requests;
3
4 class PostUpdateRequest extends PostCreateRequest
5 {
6     //
7 }
```

NOTE: We're just inheriting the `authorize()` and `rules()` methods from the `PostCreateRequest` class. Yes, we could get by with a single class to handle both but I don't know how things may change in the future and like the idea of separate classes.

Creating the `PostFormFields` Job

Let's create a utility job we can call from the `PostController`. It'll be called the `PostFormFields` job. This job will get executed when we want to get a list of all the fields to populate post form.



Laravel Job Classes are Useful

Whenever you want to encapsulate a bit of action into its own class, a job class is one way to go. You dispatch to the job and don't have to worry about the details. You can even queue jobs to occur later.

Just like `helpers.php` contain one-off functions, I like to think of job classes as one-off *action* classes.

First create the job skeleton with `artisan`.

Creating `PostFormFields` Job Skeleton

```

~/Code/15beauty$ php artisan make:job PostFormFields
Command created successfully.
```

This creates the file in `app/Jobs`.

Edit the `PostFormFields` job to match the following.

Content of PostFormFields job

```
1 <?php
2
3 namespace App\Jobs;
4
5 use App\Post;
6 use App\Tag;
7 use Carbon\Carbon;
8 use Illuminate\Contracts\Bus\SelfHandling;
9
10 class PostFormFields extends Job implements SelfHandling
11 {
12     /**
13      * The id (if any) of the Post row
14      *
15      * @var integer
16      */
17     protected $id;
18
19     /**
20      * List of fields and default value for each field
21      *
22      * @var array
23      */
24     protected $fieldList = [
25         'title' => '',
26         'subtitle' => '',
27         'page_image' => '',
28         'content' => '',
29         'meta_description' => '',
30         'is_draft' => "0",
31         'publish_date' => '',
32         'publish_time' => '',
33         'layout' => 'blog.layouts.post',
34         'tags' => [],
35     ];
36
37     /**
38      * Create a new command instance.
39      *
40      * @param integer $id
41      */
```

```

42 public function __construct($id = null)
43 {
44     $this->id = $id;
45 }
46
47 /**
48  * Execute the command.
49  *
50  * @return array of fieldnames => values
51  */
52 public function handle()
53 {
54     $fields = $this->fieldList;
55
56     if ($this->id) {
57         $fields = $this->fieldsFromModel($this->id, $fields);
58     } else {
59         $when = Carbon::now()->addHour();
60         $fields['publish_date'] = $when->format('M-j-Y');
61         $fields['publish_time'] = $when->format('g:i A');
62     }
63
64     foreach ($fields as $fieldName => $fieldValue) {
65         $fields[$fieldName] = old($fieldName, $fieldValue);
66     }
67
68     return array_merge(
69         $fields,
70         ['allTags' => Tag::lists('tag')->all()]
71     );
72 }
73
74 /**
75  * Return the field values from the model
76  *
77  * @param integer $id
78  * @param array $fields
79  * @return array
80  */
81 protected function fieldsFromModel($id, array $fields)
82 {
83     $post = Post::findOrFail($id);

```

```

84
85     $fieldNames = array_keys(array_except($fields, ['tags']));
86
87     $fields = ['id' => $id];
88     foreach ($fieldNames as $field) {
89         $fields[$field] = $post->{$field};
90     }
91
92     $fields['tags'] = $post->tags()->lists('tag')->all();
93
94     return $fields;
95 }
96 }

```

The point of this job is to return an array of fields and values to use to populate a form.

If a Post isn't loaded (as will be the case in a create), then default values will be returned. If a Post is loaded (for updates), then the values will be pulled from the database.

Also, there's two extra fields returned, `tags` and `allTags`.

1. `tags` - an array of all the tags associated with the Post.
2. `allTags` - an array of all tags on file

Adding to helpers.php

We'll need a couple one-off functions so edit the `app/helpers.php` file and add the two functions below.

Additions to helpers.php

```

1  /**
2   * Return "checked" if true
3   */
4  function checked($value)
5  {
6      return $value ? 'checked' : '';
7  }
8
9  /**
10 * Return img url for headers
11 */

```

```

12 function page_image($value = null)
13 {
14     if (empty($value)) {
15         $value = config('blog.page_image');
16     }
17     if (! starts_with($value, 'http') && $value[0] !== '/') {
18         $value = config('blog.uploads.webpath') . '/' . $value;
19     }
20
21     return $value;
22 }

```

checked()

This helper function will be used in views to output the **checked** attribute in check boxes and radio buttons.

page_image()

This function returns the full path to an image in the uploaded area using the value from the configuration. If a value isn't specified then it pulls a default image from the blog config (which you'll need to set up yourself in as 'page_image' in config/blog.php if you wish to use.)

Updating the Post Model

You may have noticed how we're breaking apart the published_at into publish_date and publish_time. Let's add a couple methods to the Post model to make this easy. Update app/Post.php as specified below.

Updates to Post Model

```

1 <?php
2 // Add the following near the top of the class, after $dates
3 protected $fillable = [
4     'title', 'subtitle', 'content_raw', 'page_image', 'meta_description',
5     'layout', 'is_draft', 'published_at',
6 ];
7
8 // Add the following three methods
9
10 /**
11  * Return the date portion of published_at
12  */

```

```
13 public function getPublishDateAttribute($value)
14 {
15     return $this->published_at->format('M-j-Y');
16 }
17
18 /**
19  * Return the time portion of published_at
20  */
21 public function getPublishTimeAttribute($value)
22 {
23     return $this->published_at->format('g:i A');
24 }
25
26 /**
27  * Alias for content_raw
28  */
29 public function getContentAttribute($value)
30 {
31     return $this->content_raw;
32 }
```

The `$fillable` property will let us fill the data during creating.

The accessors `getPublishDateAttribute()` and `getPublishTimeAttribute()` allow us to use `$post->publish_date` and `$post->publish_time` as read-only properties.

We also added `getContentAttribute()` as an accessor that returns `$this->content_raw`. Now when you use `$post->content` it'll execute this function.

Updating the Controller

Now we'll update all needed functionality in the `PostController` class. (*Remember the file for this class is in the `app/Http/Controllers/Admin` directory.*)

Because of the `Request` classes and `PostFormFields` class created earlier, the size of the controller will stay relatively small.

Update the `PostController` class to match what's below.

Content of PostController.php

```
1 <?php
2
3 namespace App\Http\Controllers\Admin;
4
5 use App\Jobs\PostFormFields;
6 use App\Http\Requests;
7 use App\Http\Requests\PostCreateRequest;
8 use App\Http\Requests\PostUpdateRequest;
9 use App\Http\Controllers\Controller;
10 use App\Post;
11
12 class PostController extends Controller
13 {
14     /**
15      * Display a listing of the posts.
16      */
17     public function index()
18     {
19         return view('admin.post.index')
20             ->withPosts(Post::all());
21     }
22
23     /**
24      * Show the new post form
25      */
26     public function create()
27     {
28         $data = $this->dispatch(new PostFormFields());
29
30         return view('admin.post.create', $data);
31     }
32
33     /**
34      * Store a newly created Post
35      *
36      * @param PostCreateRequest $request
37      */
38     public function store(PostCreateRequest $request)
39     {
40         $post = Post::create($request->postFillData());
41         $post->syncTags($request->get('tags', []));
```



```
42
43     return redirect()
44         ->route('admin.post.index')
45         ->withSuccess('New Post Successfully Created.');
```

```
46 }
47
48 /**
49  * Show the post edit form
50  *
51  * @param int $id
52  * @return Response
53  */
54 public function edit($id)
55 {
56     $data = $this->dispatch(new PostFormFields($id));
57
58     return view('admin.post.edit', $data);
59 }
60
61 /**
62  * Update the Post
63  *
64  * @param PostUpdateRequest $request
65  * @param int $id
66  */
67 public function update(PostUpdateRequest $request, $id)
68 {
69     $post = Post::findOrFail($id);
70     $post->fill($request->postFillData());
71     $post->save();
72     $post->syncTags($request->get('tags', []));
73
74     if ($request->action === 'continue') {
75         return redirect()
76             ->back()
77             ->withSuccess('Post saved.');
```

```
78     }
79
80     return redirect()
81         ->route('admin.post.index')
82         ->withSuccess('Post saved.');
```

```
83 }
```

```

84
85  /**
86   * Remove the specified resource from storage.
87   *
88   * @param int $id
89   * @return Response
90   */
91  public function destroy($id)
92  {
93      $post = Post::findOrFail($id);
94      $post->tags()->detach();
95      $post->delete();
96
97      return redirect()
98          ->route('admin.post.index')
99          ->withSuccess('Post deleted.');
```

index()

Pass index the view \$posts will all the posts and file and return it.

create()

Use the `PostFormFields` job to return all the field values. Return the create view with these values passed in.

store()

Create the post with the fillable data from the request. Attach any tags and return to the index route with a success message.

edit()

Use the `PostFormFields` job to return all the field values for the post being edited. Return the edit view with these values passed in.

update()

Load the post. Update all the fillable fields. Save any changes and keep the tags in sync. Then return either back to the edit form or to the index list with a success message.

destroy()

Load the post. Unlink any associated tags. Delete the post and return to the index route with a success message.

A pretty slim controller really. About the only thing left to do are the views.

The Post Views

We'll create all the views the PostController referenced earlier.

First update the existing `index.blade.php` in the `resources/views/admin/post` directory to match what's below.

Content of `admin.post.index` View

```

1  @extends('admin.layout')
2
3  @section('content')
4      <div class="container-fluid">
5          <div class="row page-title-row">
6              <div class="col-md-6">
7                  <h3>Posts <small>&raquo; Listing</small></h3>
8              </div>
9              <div class="col-md-6 text-right">
10                 <a href="/admin/post/create" class="btn btn-success btn-md">
11                     <i class="fa fa-plus-circle"></i> New Post
12                 </a>
13             </div>
14         </div>
15
16         <div class="row">
17             <div class="col-sm-12">
18
19                 @include('admin.partials.errors')
20                 @include('admin.partials.success')
21
22                 <table id="posts-table" class="table table-striped table-bordered">
23                     <thead>
24                         <tr>
25                             <th>Published</th>
26                             <th>Title</th>
27                             <th>Subtitle</th>
28                             <th data-sortable="false">Actions</th>
29                         </tr>
30                     </thead>
31                     <tbody>
32                         @foreach ($posts as $post)
33                             <tr>
34                                 <td data-order="{{ $post->published_at->timestamp }}">

```

```

35         {{ $post->published_at->format('j-M-y g:ia') }}
36     </td>
37     <td>{{ $post->title }}</td>
38     <td>{{ $post->subtitle }}</td>
39     <td>
40         <a href="/admin/post/{{ $post->id }}/edit"
41             class="btn btn-xs btn-info">
42             <i class="fa fa-edit"></i> Edit
43         </a>
44         <a href="/blog/{{ $post->slug }}"
45             class="btn btn-xs btn-warning">
46             <i class="fa fa-eye"></i> View
47         </a>
48     </td>
49 </tr>
50 @endforeach
51 </tbody>
52 </table>
53 </div>
54 </div>
55
56 </div>
57 @stop
58
59 @section('scripts')
60     <script>
61         $(function() {
62             $("#posts-table").DataTable({
63                 order: [[0, "desc"]]
64             });
65         });
66     </script>
67 @stop

```

This is a simple view that sets up the table with all the posts and then initializes the table as a DataTable in the **scripts** section.

Next, in the `resources/views/admin/post` directory create a `create.blade.php` file with the following content.

Content of admin.post.create View

```

1  @extends('admin.layout')
2
3  @section('styles')
4      <link href="/assets/pickadate/themes/default.css" rel="stylesheet">
5      <link href="/assets/pickadate/themes/default.date.css" rel="stylesheet">
6      <link href="/assets/pickadate/themes/default.time.css" rel="stylesheet">
7      <link href="/assets/selectize/css/selectize.css" rel="stylesheet">
8      <link href="/assets/selectize/css/selectize.bootstrap3.css" rel="stylesheet">
9  @stop
10
11 @section('content')
12     <div class="container-fluid">
13         <div class="row page-title-row">
14             <div class="col-md-12">
15                 <h3>Posts <small>&raquo; Add New Post</small></h3>
16             </div>
17         </div>
18
19         <div class="row">
20             <div class="col-sm-12">
21                 <div class="panel panel-default">
22                     <div class="panel-heading">
23                         <h3 class="panel-title">New Post Form</h3>
24                     </div>
25                     <div class="panel-body">
26
27                         @include('admin.partials.errors')
28
29                         <form class="form-horizontal" role="form" method="POST"
30                             action="{{ route('admin.post.store') }}">
31                             <input type="hidden" name="_token" value="{{ csrf_token() }}">
32
33                             @include('admin.post._form')
34
35                         <div class="col-md-8">
36                             <div class="form-group">
37                                 <div class="col-md-10 col-md-offset-2">
38                                     <button type="submit" class="btn btn-primary btn-lg">
39                                         <i class="fa fa-disk-o"></i>
40                                         Save New Post
41                                     </button>

```

```

42         </div>
43     </div>
44 </div>
45
46 </form>
47
48 </div>
49 </div>
50 </div>
51 </div>
52 </div>
53
54 @stop
55
56 @section('scripts')
57     <script src="/assets/pickadate/picker.js"></script>
58     <script src="/assets/pickadate/picker.date.js"></script>
59     <script src="/assets/pickadate/picker.time.js"></script>
60     <script src="/assets/selectize/selectize.min.js"></script>
61     <script>
62         $(function() {
63             $("#publish_date").pickadate({
64                 format: "mmm-d-yyyy"
65             });
66             $("#publish_time").pickatime({
67                 format: "h:i A"
68             });
69             $("#tags").selectize({
70                 create: true
71             });
72         });
73     </script>
74 @stop

```

Here we added in the **Selectize** and **Pickadate** libraries. You'll also note that we're referencing an `admin.post._form` partial which isn't yet created.

Let's create that partial. In the `resources/views/admin/post` directory create the `_form.blade.php` file with the following content.

Content of the admin.post._form View

```

1 <div class="row">
2   <div class="col-md-8">
3     <div class="form-group">
4       <label for="title" class="col-md-2 control-label">
5         Title
6       </label>
7       <div class="col-md-10">
8         <input type="text" class="form-control" name="title" autofocus
9           id="title" value="{{ $title }}">
10      </div>
11    </div>
12    <div class="form-group">
13      <label for="subtitle" class="col-md-2 control-label">
14        Subtitle
15      </label>
16      <div class="col-md-10">
17        <input type="text" class="form-control" name="subtitle"
18          id="subtitle" value="{{ $subtitle }}">
19      </div>
20    </div>
21    <div class="form-group">
22      <label for="page_image" class="col-md-2 control-label">
23        Page Image
24      </label>
25      <div class="col-md-10">
26        <div class="row">
27          <div class="col-md-8">
28            <input type="text" class="form-control" name="page_image"
29              id="page_image" onchange="handle_image_change()"
30              alt="Image thumbnail" value="{{ $page_image }}">
31          </div>
32          <script>
33            function handle_image_change() {
34              $("#page-image-preview").attr("src", function () {
35                var value = $("#page_image").val();
36                if ( ! value) {
37                  value = {!! json_encode(config('blog.page_image')) !!};
38                  if (value == null) {
39                    value = '';
40                  }
41                }

```

```

42         if (value.substr(0, 4) != 'http' &&
43             value.substr(0, 1) != '/') {
44             value = {!! json_encode(config('blog.uploads.webpath')) !!}
45                 + '/' + value;
46         }
47         return value;
48     });
49 }
50 </script>
51 <div class="visible-sm space-10"></div>
52 <div class="col-md-4 text-right">
53     
55 </div>
56 </div>
57 </div>
58 </div>
59 <div class="form-group">
60     <label for="content" class="col-md-2 control-label">
61         Content
62     </label>
63     <div class="col-md-10">
64         <textarea class="form-control" name="content" rows="14"
65             id="content">{{ $content }}</textarea>
66     </div>
67 </div>
68 </div>
69 <div class="col-md-4">
70     <div class="form-group">
71         <label for="publish_date" class="col-md-3 control-label">
72             Pub Date
73         </label>
74         <div class="col-md-8">
75             <input class="form-control" name="publish_date" id="publish_date"
76                 type="text" value="{{ $publish_date }}">
77         </div>
78     </div>
79 <div class="form-group">
80     <label for="publish_time" class="col-md-3 control-label">
81         Pub Time
82     </label>
83     <div class="col-md-8">

```



```

84         <input class="form-control" name="publish_time" id="publish_time"
85             type="text" value="{{ $publish_time }}">
86     </div>
87 </div>
88 <div class="form-group">
89     <div class="col-md-8 col-md-offset-3">
90         <div class="checkbox">
91             <label>
92                 <input {{ checked($is_draft) }} type="checkbox" name="is_draft">
93                 Draft?
94             </label>
95         </div>
96     </div>
97 </div>
98 <div class="form-group">
99     <label for="tags" class="col-md-3 control-label">
100     Tags
101 </label>
102 <div class="col-md-8">
103     <select name="tags[]" id="tags" class="form-control" multiple>
104         @foreach ($allTags as $tag)
105             <option @if (in_array($tag, $tags)) selected @endif
106                 value="{{ $tag }}">
107                 {{ $tag }}
108             </option>
109         @endforeach
110     </select>
111 </div>
112 </div>
113 <div class="form-group">
114     <label for="layout" class="col-md-3 control-label">
115     Layout
116 </label>
117 <div class="col-md-8">
118     <input type="text" class="form-control" name="layout"
119         id="layout" value="{{ $layout }}">
120 </div>
121 </div>
122 <div class="form-group">
123     <label for="meta_description" class="col-md-3 control-label">
124     Meta
125 </label>

```



```

28         @include('admin.partials.success')
29
30         <form class="form-horizontal" role="form" method="POST"
31             action="{{ route('admin.post.update', $id) }}">
32             <input type="hidden" name="_token" value="{{ csrf_token() }}">
33             <input type="hidden" name="_method" value="PUT">
34
35         @include('admin.post._form')
36
37         <div class="col-md-8">
38             <div class="form-group">
39                 <div class="col-md-10 col-md-offset-2">
40                     <button type="submit" class="btn btn-primary btn-lg"
41                         name="action" value="continue">
42                         <i class="fa fa-floppy-o"></i>
43                         Save - Continue
44                     </button>
45                     <button type="submit" class="btn btn-success btn-lg"
46                         name="action" value="finished">
47                         <i class="fa fa-floppy-o"></i>
48                         Save - Finished
49                     </button>
50                     <button type="button" class="btn btn-danger btn-lg"
51                         data-toggle="modal" data-target="#modal-delete">
52                         <i class="fa fa-times-circle"></i>
53                         Delete
54                     </button>
55                 </div>
56             </div>
57         </div>
58
59         </form>
60
61     </div>
62 </div>
63 </div>
64 </div>
65
66 {{-- Confirm Delete --}}
67 <div class="modal fade" id="modal-delete" tabIndex="-1">
68     <div class="modal-dialog">
69         <div class="modal-content">

```

```

70     <div class="modal-header">
71         <button type="button" class="close" data-dismiss="modal">
72             &times;
73         </button>
74         <h4 class="modal-title">Please Confirm</h4>
75     </div>
76     <div class="modal-body">
77         <p class="lead">
78             <i class="fa fa-question-circle fa-lg"></i> &nbsp;
79             Are you sure you want to delete this post?
80         </p>
81     </div>
82     <div class="modal-footer">
83         <form method="POST" action="{ route('admin.post.destroy', $id) }">
84             <input type="hidden" name="_token" value="{ csrf_token() }">
85             <input type="hidden" name="_method" value="DELETE">
86             <button type="button" class="btn btn-default"
87                 data-dismiss="modal">Close</button>
88             <button type="submit" class="btn btn-danger">
89                 <i class="fa fa-times-circle"></i> Yes
90             </button>
91         </form>
92     </div>
93 </div>
94 </div>
95 </div>
96 </div>
97
98 @stop
99
100 @section('scripts')
101     <script src="/assets/pickadate/picker.js"></script>
102     <script src="/assets/pickadate/picker.date.js"></script>
103     <script src="/assets/pickadate/picker.time.js"></script>
104     <script src="/assets/selectize/selectize.min.js"></script>
105     <script>
106         $(function() {
107             $("#publish_date").pickadate({
108                 format: "mmm-d-yyyy"
109             });
110             $("#publish_time").pickatime({
111                 format: "h:i A"

```

```
112     });
113     $("#tags").selectize({
114         create: true
115     });
116 });
117 </script>
118 @stop
```

And with that, all the views for the posts management are done.

Removing the show route

The last thing to do before the administration area is complete is to remove the post show route.

Edit `app/Http/routes.php`, editing it as instructed below.

Removing the `admin.post.show` route

```
// Find the following line
resource('admin/post', 'PostController');
// Change it to this
resource('admin/post', 'PostController', ['except' => 'show']);
```



Congratulations!

The administration area of your blog is complete. Try it out. Add a few posts, edit them. Try deleting them.

The display of the blog pages doesn't look *beautiful* yet, we'll get to that next.

Recap

This was a fairly long chapter but a huge amount was accomplished. We created a migration to modify the posts table and then updated the Tag and Post models. Then **bower** was used to pull in the Selectize.js and Pickadate.js libraries which, of course, we managed using **gulp**.

Request classes (to handle form input) were created. As was a Laravel Job class to return the post form data. Finally, the controller and views were wrapped up.

All this work resulted in a clean and easy way to administer posts.

Next we'll display the posts to the user.

Chapter 13 - Cleaning Up the Blog

In this chapter we'll get the front end of our blog cleaned up. This includes both the index page showing the list of posts and the pages showing individual posts.

Chapter Contents

- Using the Clean Blog Template
 - Fetching Clean Blog with Bower
 - Gulping Clean Blog's Less Files
 - Copying Some Header Images
- Creating the BlogIndexData Command
- Updating the BlogController
- Building the Assets
 - Creating blog.js
 - Creating blog.less
 - Updating gulpfile.js
- The Blog Views
 - The blog.layouts.master view
 - The blog.layouts.index view
 - The blog.layouts.post view
 - The blog.partials.page-nav view
 - The blog.partials.page-footer view
- Adding a Few Model Methods
 - Update the Tag Model
 - Update the Post Model
- Updating the Blog Config
- Updating our Sample Data
 - Updating the Database Seeders
 - Updating the Model Factories
 - Seeding the Database
- Recap

Using the Clean Blog Template

Clean Blog³¹ is a free blogging template provided by Start Bootstrap. We'll use it as the basis of our blog pages.

³¹<http://startbootstrap.com/template-overviews/clean-blog>

Fetching Clean Blog with Bower

Set up bower to fetch the Clean Blog source with the command below:

Adding Clean Blog to Bower

```
/Code/15beauty$ bower install clean-blog --save
```

There may be a couple warning errors about the clean blog repository not having `bower.json` set up correctly, but that's okay.

Gulping Clean Blog's Less Files

Edit your `gulpfile.js` and in the `copyfiles` task, at the bottom of the function, add the following lines.

Copying the Clean Blog Less Files

```
1 // Copy clean-blog less files
2 gulp.src("vendor/bower_dl/clean-blog/less/**")
3   .pipe(gulp.dest("resources/assets/less/clean-blog"));
```

Now when you do a `gulp copyfiles` then the latest clean blog files will be copied.

We'll use these files in a little bit as part of the blog's CSS.

Copying Some Header Images

In order to have a few header images to play with we'll copy the four header images that the Clean Blog template provides into our upload area. Depending on how you configured your Upload Manager these images will end up either on your local file system (*in the public/uploads directory*) or on the Amazon S3 servers.

Point your browser to `http://15beauty.app/admin` and then click on the **Uploads** link in the menu bar. Upload the following files:

- `about-bg.jpg`
- `contact-bg.jpg`
- `home-bg.jpg`
- `post-bg.jpg`

The files are located in the `img` folder of the Clean Blog sources you just copied using Bower (which should be `vendor/bower_dl/clean-blog`).

Creating the BlogIndexData Job

The last time we touched the `BlogController` was back in Chapter 7 when we created the 10 Minute Blog. At that time we didn't have the tagging feature with our blog.

If a tag is specified in the query string, we'll need to gather the list of posts, filtering them so only posts with that tag displays. Rather than adding the logic to do this in the controller, let's create a one-off job to gather the index data.

First, create the job class as instructed below.

Creating the BlogIndexData Job

```
vagrant@homestead:~/Code/15beauty$ php artisan make:job BlogIndexData
Job created successfully.
```

Now edit the newly created `BlogIndexData.php` file. It's in your `app/Jobs` directory.

Content of BlogIndexData.php

```
1 <?php
2
3 namespace App\Jobs;
4
5 use App\Post;
6 use App\Tag;
7 use Carbon\Carbon;
8 use Illuminate\Contracts\Bus\SelfHandling;
9
10 class BlogIndexData extends Job implements SelfHandling
11 {
12     protected $tag;
13
14     /**
15      * Constructor
16      *
17      * @param string|null $tag
18      */
19     public function __construct($tag)
20     {
21         $this->tag = $tag;
22     }
23
24     /**
```



```
25     * Execute the command.
26     *
27     * @return array
28     */
29 public function handle()
30 {
31     if ($this->tag) {
32         return $this->tagIndexData($this->tag);
33     }
34
35     return $this->normalIndexData();
36 }
37
38 /**
39  * Return data for normal index page
40  *
41  * @return array
42  */
43 protected function normalIndexData()
44 {
45     $posts = Post::with('tags')
46         ->where('published_at', '<=', Carbon::now())
47         ->where('is_draft', 0)
48         ->orderBy('published_at', 'desc')
49         ->simplePaginate(config('blog.posts_per_page'));
50
51     return [
52         'title' => config('blog.title'),
53         'subtitle' => config('blog.subtitle'),
54         'posts' => $posts,
55         'page_image' => config('blog.page_image'),
56         'meta_description' => config('blog.description'),
57         'reverse_direction' => false,
58         'tag' => null,
59     ];
60 }
61
62 /**
63  * Return data for a tag index page
64  *
65  * @param string $tag
66  * @return array
```

```

67      */
68      protected function tagIndexData($tag)
69      {
70          $tag = Tag::where('tag', $tag)->firstOrFail();
71          $reverse_direction = (bool)$tag->reverse_direction;
72
73          $posts = Post::where('published_at', '<=', Carbon::now())
74              ->whereHas('tags', function ($q) use ($tag) {
75                  $q->where('tag', '=', $tag->tag);
76              })
77              ->where('is_draft', 0)
78              ->orderBy('published_at', $reverse_direction ? 'asc' : 'desc')
79              ->simplePaginate(config('blog.posts_per_page'));
80          $posts->addQuery('tag', $tag->tag);
81
82          $page_image = $tag->page_image ?: config('blog.page_image');
83
84          return [
85              'title' => $tag->title,
86              'subtitle' => $tag->subtitle,
87              'posts' => $posts,
88              'page_image' => $page_image,
89              'tag' => $tag,
90              'reverse_direction' => $reverse_direction,
91              'meta_description' => $tag->meta_description ?: \
92                  config('blog.description'),
93          ];
94      }
95  }

```

__construct()

Just stash the tag passed to the constructor.

handle()

A simple method. If a value for \$tag was passed during construction we call one method to gather the data, otherwise a different method is called.

normalIndexData()

This method returns the index data the normal way. That is, without a filter on the tag. The code is almost identical to what we did with the 10 minute blog, but now any tags the posts have are *Eager Loaded* (the with() method does this.). Also, we do filter the query to not include any *draft* posts.

tagIndexData()

Here we first load the Tag and then filter posts to match the tag. This is accomplished with the `whereHas()` method. *Don't forget that line continuation character (the backslash) where 'meta_description' is returned should not be typed, instead continue typing the next line without hitting enter!*

Notice all the extra data we're returning? Before, in the 10 minute blog, we only returned the `$posts` to the view. But now we return all kinds of information. Shortly, you'll see how this is used in the index view.

**Eager Loading**

Eager Loading helps with the [n+1 query problem](#)³². It loads queries having this issue in two queries instead of many. This can drastically increase the application's performance. See the official Laravel 5.1 documentation for a complete example.

Updating the BlogController

Update `BlogController.php` to match what's below.

Updated BlogController

```

1  <?php
2  namespace App\Http\Controllers;
3
4  use App\Jobs\BlogIndexData;
5  use App\Http\Requests;
6  use App\Post;
7  use App\Tag;
8  use Illuminate\Http\Request;
9
10 class BlogController extends Controller
11 {
12     public function index(Request $request)
13     {
14         $tag = $request->get('tag');
15         $data = $this->dispatch(new BlogIndexData($tag));
16         $layout = $tag ? Tag::layout($tag) : 'blog.layouts.index';
17
18         return view($layout, $data);

```

³²<http://stackoverflow.com/questions/97197/what-is-the-n1-selects-issue>

```
19   }
20
21   public function showPost($slug, Request $request)
22   {
23       $post = Post::with('tags')->whereSlug($slug)->firstOrFail();
24       $tag = $request->get('tag');
25       if ($tag) {
26           $tag = Tag::whereTag($tag)->firstOrFail();
27       }
28
29       return view($post->layout, compact('post', 'tag'));
30   }
31 }
```

index()

Pull any \$tag value from the request and use the BlogIndexData command to figure the data. Because we want the ability to have different index templates for different tags, we ask the Tag class for the template if a \$tag is used, otherwise we go with the default.

showPost()

Any associated tags are Eager Loaded with the post. If there's any \$tag passed in the query string, we convert \$tag over to the actual Tag record before passing it to the view.

Building the Assets

There's a bit more of coding to do, and the views to create, but first let's get all the blog's assets in place.

Creating blog.js

Create blog.js in the resources/assets/js directory with the following content.

Content of blog.js

```
1  /*
2   * Blog Javascript
3   * Copied from Clean Blog v1.0.0 (http://startbootstrap.com)
4   */
5
6  // Navigation Scripts to Show Header on Scroll-Up
7  jQuery(document).ready(function($) {
8      var MQL = 1170;
9
10     //primary navigation slide-in effect
11     if ($(window).width() > MQL) {
12         var headerHeight = $('.navbar-custom').height();
13         $(window).on('scroll', {
14             previousTop: 0
15         },
16         function() {
17             var currentTop = $(window).scrollTop();
18
19             //if user is scrolling up
20             if (currentTop < this.previousTop) {
21                 if (currentTop > 0 && $('.navbar-custom').hasClass('is-fixed')) {
22                     $('.navbar-custom').addClass('is-visible');
23                 } else {
24                     $('.navbar-custom').removeClass('is-visible is-fixed');
25                 }
26                 //if scrolling down...
27             } else {
28                 $('.navbar-custom').removeClass('is-visible');
29                 if (currentTop > headerHeight &&
30                     !$('.navbar-custom').hasClass('is-fixed')) {
31                     $('.navbar-custom').addClass('is-fixed');
32                 }
33             }
34             this.previousTop = currentTop;
35         });
36     }
37
38     // Initialize tooltips
39     $('[data-toggle="tooltip"]').tooltip();
40 });
```

This code implement tooltips and will cause the navigation bar to appear when the user scrolls up. It was copied from Clean Blog's `js/clean-blog.js` file.

Creating `blog.less`

In the `resources/assets/less` directory create a file named `blog.less` with the following content.

Content of `blog.less`

```
1  @import "bootstrap/bootstrap";
2  @import "fontawesome/font-awesome";
3  @import "clean-blog/clean-blog";
4
5  @import "//fonts.googleapis.com/css?family=Lora:400,700,\
6    400italic,700italic";
7  @import "//fonts.googleapis.com/css?family=Open+Sans:300italic,400italic,\
8    600italic,700italic,800italic,400,300,600,700,800'";
9
10 .intro-header .post-heading .meta a,
11 article a {
12   text-decoration: underline;
13 }
14
15 h2 {
16   padding-top: 22px;
17 }
18 h3 {
19   padding-top: 15px;
20 }
21
22 h2 + p, h3 + p, h4 + p {
23   margin-top: 5px;
24 }
25
26 // Adjust position of captions
27 .caption-title {
28   margin-bottom: 5px;
29 }
30 .caption-title + p {
31   margin-top: 0;
32 }
33
34 // Change the styling of dt/dd elements
```

```
35 dt {
36     margin-bottom: 5px;
37 }
38 dd {
39     margin-left: 30px;
40     margin-bottom: 10px;
41 }
```

This file pulls in Bootstrap, Font Awesome and Clean Blog. Then the fonts are imported and a touch of styling is added to a few elements. *(Again, careful of those line continuation characters on the 4th and 5th @import lines.)*

Updating gulpfile.js

Update `gulpfile.js` to match what's below. The only changes at this point are the addition of the scripts for `blog.js` and `blog.less`.

Final Version of `gulpfile.js`

```
1  var gulp = require('gulp');
2  var rename = require('gulp-rename');
3  var elixir = require('laravel-elixir');
4
5  /**
6   * Copy any needed files.
7   *
8   * Do a 'gulp copyfiles' after bower updates
9   */
10 gulp.task("copyfiles", function() {
11
12     // Copy jQuery, Bootstrap, and FontAwesome
13     gulp.src("vendor/bower_dl/jquery/dist/jquery.js")
14         .pipe(gulp.dest("resources/assets/js/"));
15
16     gulp.src("vendor/bower_dl/bootstrap/less/**")
17         .pipe(gulp.dest("resources/assets/less/bootstrap"));
18
19     gulp.src("vendor/bower_dl/bootstrap/dist/js/bootstrap.js")
20         .pipe(gulp.dest("resources/assets/js/"));
21
22     gulp.src("vendor/bower_dl/bootstrap/dist/fonts/**")
23         .pipe(gulp.dest("public/assets/fonts/"));
```

```
24
25     gulp.src("vendor/bower_dl/fontawesome/less/**")
26         .pipe(gulp.dest("resources/assets/less/fontawesome"));
27
28     gulp.src("vendor/bower_dl/fontawesome/fonts/**")
29         .pipe(gulp.dest("public/assets/fonts"));
30
31     // Copy datatables
32     var dtDir = 'vendor/bower_dl/datatables-plugins/integration/';
33
34     gulp.src("vendor/bower_dl/datatables/media/js/jquery.dataTables.js")
35         .pipe(gulp.dest('resources/assets/js/'));
36
37     gulp.src(dtDir + 'bootstrap/3/dataTables.bootstrap.css')
38         .pipe(rename('dataTables.bootstrap.less'))
39         .pipe(gulp.dest('resources/assets/less/others/'));
40
41     gulp.src(dtDir + 'bootstrap/3/dataTables.bootstrap.js')
42         .pipe(gulp.dest('resources/assets/js/'));
43
44     // Copy selectize
45     gulp.src("vendor/bower_dl/selectize/dist/css/**")
46         .pipe(gulp.dest("public/assets/selectize/css"));
47
48     gulp.src("vendor/bower_dl/selectize/dist/js/standalone/selectize.min.js")
49         .pipe(gulp.dest("public/assets/selectize/"));
50
51     // Copy pickadate
52     gulp.src("vendor/bower_dl/pickadate/lib/compressed/themes/**")
53         .pipe(gulp.dest("public/assets/pickadate/themes/"));
54
55     gulp.src("vendor/bower_dl/pickadate/lib/compressed/picker.js")
56         .pipe(gulp.dest("public/assets/pickadate/"));
57
58     gulp.src("vendor/bower_dl/pickadate/lib/compressed/picker.date.js")
59         .pipe(gulp.dest("public/assets/pickadate/"));
60
61     gulp.src("vendor/bower_dl/pickadate/lib/compressed/picker.time.js")
62         .pipe(gulp.dest("public/assets/pickadate/"));
63
64     // Copy clean-blog less files
65     gulp.src("vendor/bower_dl/clean-blog/less/**")
```



```

66         .pipe(gulp.dest("resources/assets/less/clean-blog"));
67     });
68
69     /**
70      * Default gulp is to run this elixir stuff
71      */
72     elixir(function(mix) {
73
74         // Combine scripts
75         mix.scripts([
76             'js/jquery.js',
77             'js/bootstrap.js',
78             'js/jquery.dataTables.js',
79             'js/dataTables.bootstrap.js'
80         ],
81             'public/assets/js/admin.js', 'resources//assets');
82
83         // Combine blog scripts
84         mix.scripts([
85             'js/jquery.js',
86             'js/bootstrap.js',
87             'js/blog.js'
88         ], 'public/assets/js/blog.js', 'resources//assets');
89
90         // Compile CSS
91         mix.less('admin.less', 'public/assets/css/admin.css');
92         mix.less('blog.less', 'public/assets/css/blog.css');
93     });

```

Run gulp twice. First `gulp copyfiles` to copy the needed clean-blog assets. Then just a plain `gulp` to combine everything.

The Blog Views

Let's wrap up the views for showing the blog and individual posts.

You can delete the `index.blade.php` and `post.blade.php` files that are in the `resources/views/blog` directory. They are still there from the **10 Minute Blog** chapter and we don't need them any longer.

The `blog.layouts.master` view

Create a folder named `layouts` in the `resources/views/blog` directory and create a new file there named `master.blade.php`. Update this file to match the contents below.

Content of blog.layouts.mater View

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1">
7      <meta name="description" content="{{ $meta_description }}">
8      <meta name="author" content="{{ config('blog.author') }}">
9
10     <title>{{ $title or config('blog.title') }}</title>
11
12     {{-- Styles --}}
13     <link href="/assets/css/blog.css" rel="stylesheet">
14     @yield('styles')
15
16     {{-- HTML5 Shim and Respond.js for IE8 support --}}
17     <!--[if lt IE 9]>
18     <script src="//oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
19     <script src="//oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js"></script>
20     <![endif]-->
21 </head>
22 <body>
23 @include('blog.partials.page-nav')
24
25 @yield('page-header')
26 @yield('content')
27
28 @include('blog.partials.page-footer')
29
30 {{-- Scripts --}}
31 <script src="/assets/js/blog.js"></script>
32 @yield('scripts')
33
34 </body>
35 </html>

```

This is the basic layout we'll use for other layouts.

The blog.layouts.index view

Create a index.blade.php view in the same folder with the contents below.

Content of blog.layouts.index View

```

1  @extends('blog.layouts.master')
2
3  @section('page-header')
4      <header class="intro-header"
5          style="background-image: url('{{ page_image($page_image) }}')">
6          <div class="container">
7              <div class="row">
8                  <div class="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
9                      <div class="site-heading">
10                         <h1>{{ $title }}</h1>
11                         <hr class="small">
12                         <h2 class="subheading">{{ $subtitle }}</h2>
13                     </div>
14                 </div>
15             </div>
16         </div>
17     </header>
18 @stop
19
20 @section('content')
21     <div class="container">
22         <div class="row">
23             <div class="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
24
25                 {{-- The Posts --}}
26                 @foreach ($posts as $post)
27                     <div class="post-preview">
28                         <a href="{{ $post->url($tag) }}">
29                             <h2 class="post-title">{{ $post->title }}</h2>
30                             @if ($post->subtitle)
31                                 <h3 class="post-subtitle">{{ $post->subtitle }}</h3>
32                             @endif
33                         </a>
34                         <p class="post-meta">
35                             Posted on {{ $post->published_at->format('F j, Y') }}
36                             @if ($post->tags->count())
37                                 in
38                                 {!! join(', ', $post->tagLinks()) !!}
39                             @endif
40                         </p>
41                     </div>

```

```

42     <hr>
43 @endforeach
44
45 {{-- The Pager --}}
46 <ul class="pager">
47
48     {{-- Reverse direction --}}
49     @if ($reverse_direction)
50         @if ($posts->currentPage() > 1)
51             <li class="previous">
52                 <a href="{!! $posts->url($posts->currentPage() - 1) !!}">
53                     <i class="fa fa-long-arrow-left fa-lg"></i>
54                     Previous {{ $tag->tag }} Posts
55                 </a>
56             </li>
57         @endif
58         @if ($posts->hasMorePages())
59             <li class="next">
60                 <a href="{!! $posts->nextPageUrl() !!}">
61                     Next {{ $tag->tag }} Posts
62                     <i class="fa fa-long-arrow-right"></i>
63                 </a>
64             </li>
65         @endif
66     @else
67         @if ($posts->currentPage() > 1)
68             <li class="previous">
69                 <a href="{!! $posts->url($posts->currentPage() - 1) !!}">
70                     <i class="fa fa-long-arrow-left fa-lg"></i>
71                     Newer {{ $tag ? $tag->tag : '' }} Posts
72                 </a>
73             </li>
74         @endif
75         @if ($posts->hasMorePages())
76             <li class="next">
77                 <a href="{!! $posts->nextPageUrl() !!}">
78                     Older {{ $tag ? $tag->tag : '' }} Posts
79                     <i class="fa fa-long-arrow-right"></i>
80                 </a>
81             </li>
82         @endif
83     @endif

```

```

84         </ul>
85     </div>
86
87 </div>
88 </div>
89 @stop

```

The `blog.layouts.index` view will show the blog index pages. It wraps the **page-header** in it's own section. The **content** loops through the `$posts` and displays navigation afterward.

The `blog.layouts.post` view

Next we'll create the view to show a particular post. Create `post.blade.php` view in `resources/views/blog/layouts` with the contents below.

Content of `blog.layouts.post` View

```

1  @extends('blog.layouts.master', [
2      'title' => $post->title,
3      'meta_description' => $post->meta_description ?: config('blog.description'),
4  ])
5
6  @section('page-header')
7      <header class="intro-header"
8          style="background-image: url('{{ page_image($post->page_image) }}')">
9          <div class="container">
10             <div class="row">
11                 <div class="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
12                     <div class="post-heading">
13                         <h1>{{ $post->title }}</h1>
14                         <h2 class="subheading">{{ $post->subtitle }}</h2>
15                         <span class="meta">
16                             Posted on {{ $post->published_at->format('F j, Y') }}
17                             @if ($post->tags->count())
18                                 in
19                                 {!! join(', ', $post->tagLinks()) !!}
20                             @endif
21                         </span>
22                     </div>
23                 </div>
24             </div>
25         </div>

```

```

26     </header>
27 @stop
28
29 @section('content')
30
31     {{-- The Post --}}
32     <article>
33         <div class="container">
34             <div class="row">
35                 <div class="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
36                     {!! $post->content_html !!}
37                 </div>
38             </div>
39         </div>
40     </article>
41
42     {{-- The Pager --}}
43     <div class="container">
44         <div class="row">
45             <ul class="pager">
46                 @if ($tag && $tag->reverse_direction)
47                     @if ($post->olderPost($tag))
48                         <li class="previous">
49                             <a href="{!! $post->olderPost($tag)->url($tag) !!}">
50                                 <i class="fa fa-long-arrow-left fa-lg"></i>
51                                 Previous {{ $tag->tag }} Post
52                             </a>
53                         </li>
54                     @endif
55                     @if ($post->newerPost($tag))
56                         <li class="next">
57                             <a href="{!! $post->newerPost($tag)->url($tag) !!}">
58                                 Next {{ $tag->tag }} Post
59                                 <i class="fa fa-long-arrow-right"></i>
60                             </a>
61                         </li>
62                     @endif
63                 @else
64                     @if ($post->newerPost($tag))
65                         <li class="previous">
66                             <a href="{!! $post->newerPost($tag)->url($tag) !!}">
67                                 <i class="fa fa-long-arrow-left fa-lg"></i>

```

```

68         Next Newer {{ $tag ? $tag->tag : '' }} Post
69         </a>
70     </li>
71 @endif
72 @if ($post->olderPost($tag))
73     <li class="next">
74         <a href="{!! $post->olderPost($tag)->url($tag) !!}">
75             Next Older {{ $tag ? $tag->tag : '' }} Post
76             <i class="fa fa-long-arrow-right"></i>
77         </a>
78     </li>
79 @endif
80 @endif
81 </ul>
82 </div>
83
84 </div>
85 @stop

```

Like the `blog.layouts.index` view this one has a **page-header** and a **content** section.

That's it for the layouts.

The `blog.partials.page-nav` view

Create a `partials` directory in the `resources/views/blog` folder. Inside it, create a `page-nav.blade.php` file with the following.

Content of the `blog.partials.page-nav` View

```

1  {{-- Navigation --}}
2  <nav class="navbar navbar-default navbar-custom navbar-fixed-top">
3      <div class="container-fluid">
4          {{-- Brand and toggle get grouped for better mobile display --}}
5          <div class="navbar-header page-scroll">
6              <button type="button" class="navbar-toggle" data-toggle="collapse"
7                  data-target="#navbar-main">
8                  <span class="sr-only">Toggle navigation</span>
9                  <span class="icon-bar"></span>
10                 <span class="icon-bar"></span>
11                 <span class="icon-bar"></span>
12             </button>
13             <a class="navbar-brand" href="/">{{ config('blog.name') }}</a>

```

```
14     </div>
15
16     {{-- Collect the nav links, forms, and other content for toggling --}}
17     <div class="collapse navbar-collapse" id="navbar-main">
18         <ul class="nav navbar-nav">
19             <li>
20                 <a href="/">Home</a>
21             </li>
22         </ul>
23     </div>
24 </div>
25 </nav>
```

The menu on the navbar at top will only have a single option, **Home**.

The `blog.partials.page-footer` view

Finally, create a `page-footer.blade.php` file in the same directory with the content below.

Content of the `blog.partials.page-footer` View

```
1 <hr>
2 <footer>
3     <div class="container">
4         <div class="row">
5             <div class="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
6                 <p class="copyright">Copyright &copy; {{ config('blog.author') }}</p>
7             </div>
8         </div>
9     </div>
10 </footer>
```

That should wrap up the views.

Adding a Few Model Methods

In order for the views to work, there's a few Model methods that need to be added.

Update the Tag Model

Update `app/Tag.php`, adding the method below to the Tag class.

Updates to the Tag Model

```

1  /**
2   * Return the index layout to use for a tag
3   *
4   * @param string $tag
5   * @param string $default
6   * @return string
7   */
8  public static function layout($tag, $default = 'blog.layouts.index')
9  {
10     $layout = static::whereTag($tag)->pluck('layout');
11
12     return $layout ?: $default;
13 }

```

The `layout()` method returns a Tag's layout, or if there isn't a tag, or the tag doesn't have a layout, then a default value is returned.

Update the Post Model

Update `app/Post.php`, adding the `use` statement and the four methods below to the `Post` class.

Updates to the Post Model

```

1  <?php
2  // Add the following use statement with the others at the top of the file
3  use Carbon\Carbon;
4
5  // Add the 4 methods below to the class
6  /**
7   * Return URL to post
8   *
9   * @param Tag $tag
10  * @return string
11  */
12  public function url(Tag $tag = null)
13  {
14     $url = url('blog/'.$this->slug);
15     if ($tag) {
16         $url .= '?tag='.urlencode($tag->tag);
17     }

```

```

18
19     return $url;
20 }
21
22 /**
23  * Return array of tag links
24  *
25  * @param string $base
26  * @return array
27  */
28 public function tagLinks($base = '/blog?tag=%TAG%')
29 {
30     $tags = $this->tags()->lists('tag');
31     $return = [];
32     foreach ($tags as $tag) {
33         $url = str_replace('%TAG%', urlencode($tag), $base);
34         $return[] = '<a href="'. $url. '">'.e($tag). '</a>';
35     }
36     return $return;
37 }
38
39 /**
40  * Return next post after this one or null
41  *
42  * @param Tag $tag
43  * @return Post
44  */
45 public function newerPost(Tag $tag = null)
46 {
47     $query =
48         static::where('published_at', '>', $this->published_at)
49             ->where('published_at', '<=', Carbon::now())
50             ->where('is_draft', 0)
51             ->orderBy('published_at', 'asc');
52     if ($tag) {
53         $query = $query->whereHas('tags', function ($q) use ($tag) {
54             $q->where('tag', '=', $tag->tag);
55         });
56     }
57
58     return $query->first();
59 }

```

```

60
61  /**
62   * Return older post before this one or null
63   *
64   * @param Tag $tag
65   * @return Post
66   */
67  public function olderPost(Tag $tag = null)
68  {
69      $query =
70          static::where('published_at', '<', $this->published_at)
71              ->where('is_draft', 0)
72              ->orderBy('published_at', 'desc');
73      if ($tag) {
74          $query = $query->whereHas('tags', function ($q) use ($tag) {
75              $q->where('tag', '=', $tag->tag);
76          });
77      }
78
79      return $query->first();
80  }

```

url()

This method returns the URI to the particular post with an optional **tag** in the query string. The `blog.layouts.index` view uses it to link to a post details page.

tagLinks()

This method returns an array of links, each link going to the index page for a particular tag the post has been, uh, tagged with.

newerPost()

Returns the next `Post` coming after `$this` or `null` if there are no newer posts.

olderPost()

Returns the previous `Post` coming before `$this` or `null` if there are no older posts.

Updating the Blog Config

Update the `blog/config.php` file so it looks similar to the one below. Note, use your own values here.

The Blog Config File

```
1 <?php
2 return [
3     'name' => "L5 Beauty",
4     'title' => "Laravel 5.1 Beauty",
5     'subtitle' => 'A clean blog written in Laravel 5.1',
6     'description' => 'This is my meta description',
7     'author' => 'Chuck Heintzelman',
8     'page_image' => 'home-bg.jpg',
9     'posts_per_page' => 10,
10    'uploads' => [
11        'storage' => 'local',
12        'webpath' => '/uploads/',
13    ],
14 ];
```

Again, use your own values here. Especially the uploads section. If you're using Amazon S3 this will look different.

Updating our Sample Data

Back in Chapter 10, when we created the **10 Minute Blog**, we set up the Database Seeder to generate random data using Model Factories.

But now, the database has changed.

Let's update the seeder and factories and re-seed the database to populate tags and other fields.

Updating the Database Seeders

In the database/seeds directory there's currently a single file named DatabaseSeeder.php. Edit this to match what's below and then create the additional two files after it.

Content of DatabaseSeeder.php

```
1 <?php
2
3 use Illuminate\Database\Seeder;
4 use Illuminate\Database\Eloquent\Model;
5
6 class DatabaseSeeder extends Seeder
7 {
8     /**
9      * Run the database seeds.
10     *
11     * @return void
12     */
13     public function run()
14     {
15         Model::unguard();
16
17         $this->call('TagTableSeeder');
18         $this->call('PostTableSeeder');
19
20         Model::reguard();
21     }
22 }
```

In the same directory create TagTableSeeder.php with the following content.

Content of TagTableSeeder

```
1 <?php
2
3 use App\Tag;
4 use Illuminate\Database\Seeder;
5
6 class TagTableSeeder extends Seeder
7 {
8     /**
9      * Seed the tags table
10     */
11     public function run()
12     {
13         Tag::truncate();
14     }
15 }
```

```
15     factory(Tag::class, 5)->create();
16 }
17 }
```

Next created `PostTableSeeder.php` in the same directory with the following.

Content of `PostTableSeeder.php`

```
1  <?php
2
3  use App\Post;
4  use App\Tag;
5  use Illuminate\Database\Seeder;
6  use Illuminate\Support\Facades\DB;
7
8  class PostTableSeeder extends Seeder
9  {
10     /**
11      * Seed the posts table
12      */
13     public function run()
14     {
15         // Pull all the tag names from the file
16         $tags = Tag::lists('tag')->all();
17
18         Post::truncate();
19
20         // Don't forget to truncate the pivot table
21         DB::table('post_tag_pivot')->truncate();
22
23         factory(Post::class, 20)->create()->each(function ($post) use ($tags) {
24
25             // 30% of the time don't assign a tag
26             if (mt_rand(1, 100) <= 30) {
27                 return;
28             }
29
30             shuffle($tags);
31             $postTags = [$tags[0]];
32
33             // 30% of the time we're assigning tags, assign 2
34             if (mt_rand(1, 100) <= 30) {
35                 $postTags[] = $tags[1];
```

```

36     }
37
38     $post->syncTags($postTags);
39 });
40 }
41 }

```

This last seeder is a tiny bit longer because we randomly tie some of the tags to the posts.

Updating the Model Factories

Now update the Model Factories by editing `ModelFactory.php` in the `database/factories` directory to match what's below.

Content of ModelFactory.php

```

1  <?php
2
3  $factory->define(App\User::class, function ($faker) {
4      return [
5          'name' => $faker->name,
6          'email' => $faker->email,
7          'password' => str_random(10),
8          'remember_token' => str_random(10),
9      ];
10 });
11
12 $factory->define(App\Post::class, function ($faker) {
13     $images = ['about-bg.jpg', 'contact-bg.jpg', 'home-bg.jpg', 'post-bg.jpg'];
14     $title = $faker->sentence(mt_rand(3, 10));
15     return [
16         'title' => $title,
17         'subtitle' => str_limit($faker->sentence(mt_rand(10, 20)), 252),
18         'page_image' => $images[mt_rand(0, 3)],
19         'content_raw' => join("\n\n", $faker->paragraphs(mt_rand(3, 6))),
20         'published_at' => $faker->dateTimeBetween('-1 month', '+3 days'),
21         'meta_description' => "Meta for $title",
22         'is_draft' => false,
23     ];
24 });
25
26 $factory->define(App\Tag::class, function ($faker) {

```

```
27 $images = ['about-bg.jpg', 'contact-bg.jpg', 'home-bg.jpg', 'post-bg.jpg'];
28 $word = $faker->word;
29 return [
30     'tag' => $word,
31     'title' => ucfirst($word),
32     'subtitle' => $faker->sentence,
33     'page_image' => $images[mt_rand(0, 3)],
34     'meta_description' => "Meta for $word",
35     'reverse_direction' => false,
36 ];
37 });
```

Seeding the Database

Now seed the database. First we'll dump the autoloader to make sure the new Seed classes are known.

Dumping the Autoloader

```
~/Code/15beauty% composer dumpauto
Generating autoload files
```

Then seed the database from the Homestead VM.

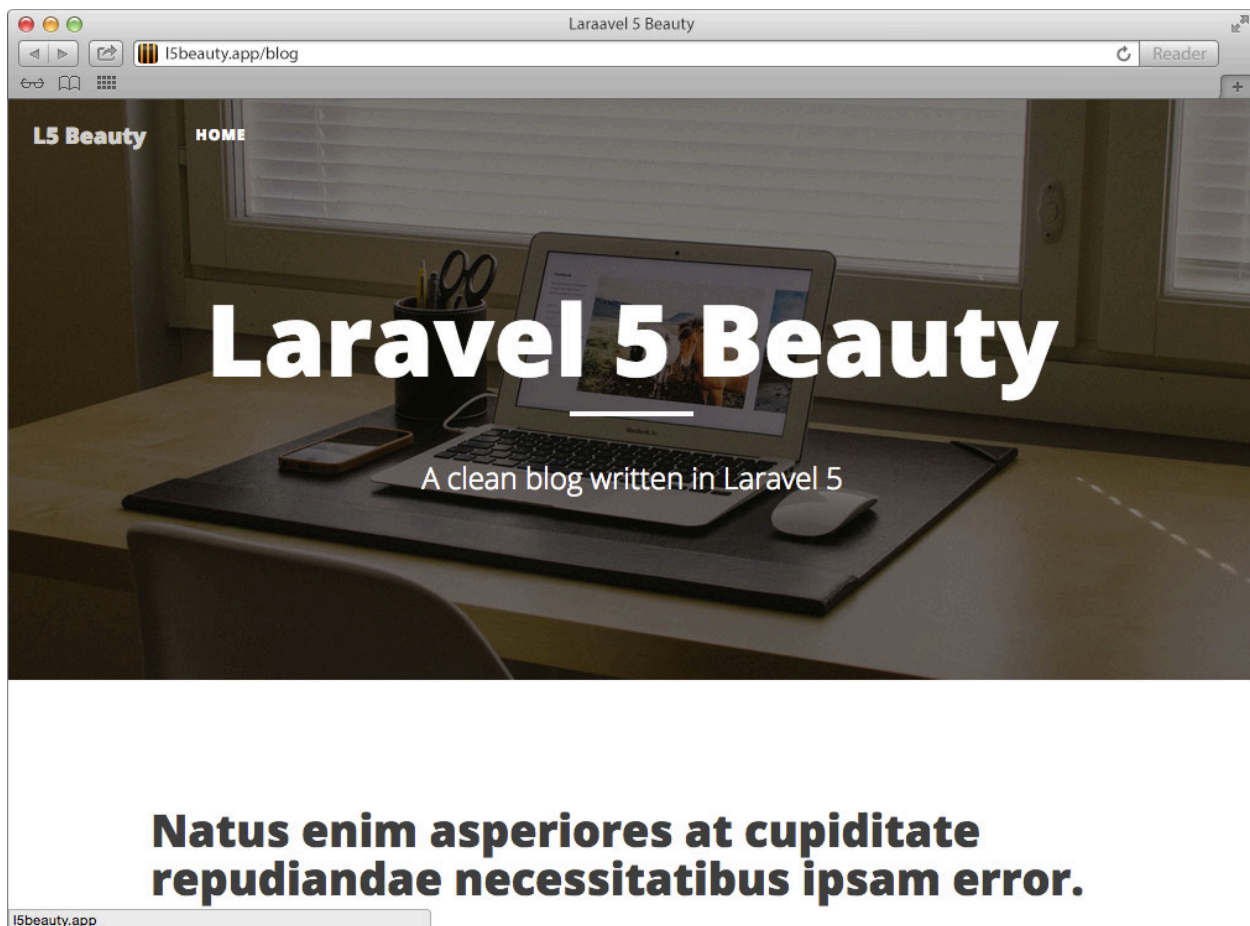
Seeding the Database

```
vagrant@homestead:~/Code/15beauty$ php artisan db:seed
Seeded: TagTableSeeder
Seeded: PostTableSeeder
```



The Blog Is Completely Usable

Browse around. Look at posts, navigate back and forth. You'll see screens similar to the one below.



Example Home Page

Recap

This chapter focused on cleaning up the blog pages. To do this we used the Clean Blog template by StartBootstrap as a model. Clean Blog was pulled in with **Bower**, then we made use of much of the assets as we built the index and post pages.

The blogging application is complete and usable. It's time to start adding a few features to our blog. In the next chapter we'll add a "Contact Us" form as we delve into Laravel 5.1 Queues.

Chapter 14 - Sending Mail and Using Queues

In this chapter we'll add a *Contact Us* form to the blog. To do this we'll explore Laravel 5.1's mailing functions and set up a queue for asynchronous processing.

Contents

- [Setting Up for Emails](#)
 - [Configuring for Gmail](#)
 - [Configuring for Mailgun](#)
 - [Testing Mail with Tinker](#)
- [Adding a Contact Us Form](#)
 - [Adding the Link and Route](#)
 - [Creating the FormRequest](#)
 - [Adding the Controller](#)
 - [Creating the Views](#)
 - [Sending the Mail](#)
- [About Queues](#)
 - [How they Work](#)
 - [The Different Queue Drivers](#)
 - [Using the Database Driver](#)
- [Queuing the Contact Us Email](#)
 - [Changing the Controller](#)
 - [Where's the Email](#)
 - [Running queue:work](#)
- [Automatically Processing the Queue](#)
 - [Running queue:listen with supervisord](#)
 - [Using a Scheduled Command](#)
- [Queing Jobs](#)
- [Recap](#)

Setting Up for Emails

In order to use Laravel 5.1's mail functionality it first needs to be configured. Configuration is easy. Look at the mail settings that come out of the box in the `.env` file.

Mail Configuration in .env

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

As you can see, just a few simple settings.

Configuring for Gmail

Let's say you have a gmail account you wish to configure. Here's how to do it.

First, edit `config/mail.php` as instructed below.

Changes to config/mail.php

```
1 // Find the following line
2 'from' => ['address' => null, 'name' => null],
3 // Change it to
4 'from' => ['address' => env('MAIL_FROM'), 'name' => env('MAIL_NAME')],
```

This sets up who the emails are from which is required by Gmail and is good practice for other providers.

Next, edit `.env`, changing the mail configuration to what's below (replacing USERNAME, PASSWORD, FROM, etc. your own settings).

Gmail Configuration in .env

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=YOUR@EMAIL.COM
MAIL_PASSWORD=YOUR-GMAIL-PASSWORD
MAIL_FROM=YOUR@EMAIL.COM
MAIL_NAME=YOUR-NAME
```



Are You Using 2-Step Authentication

If your Gmail account uses 2-Step Authentication then the password required will not be the same one you log into Gmail with. You'll need to set up an App Password. See [Google Help](https://support.google.com/accounts/answer/185833)³³ for instructions on how to do this.

³³<https://support.google.com/accounts/answer/185833>

The section after the next one (Testing Mail with Tinker) will explain how you can test your mail configuration.

Configuring for Mailgun

Another popular option is to use Mailgun to send your email. I use Mailgun. It's completely free for the first 10,000 emails you send each month. After that it's a penny for every 20 emails.

To configure to send through mailgun, first edit `config/services.php` to match what's below.

Mailgun Settings in `config/services.php`

```
1  'mailgun' => [  
2      'domain' => env('MAILGUN_DOMAIN'),  
3      'secret' => env('MAILGUN_SECRET'),  
4  ],
```

Yes, we're just setting it up to read the values from the `.env` file.

Next, Mailgun requires the *Guzzle Http* library, so use composer to require it.

Requiring Guzzle Http

```
~/Code/15beauty% composer require "guzzlehttp/guzzle=~5.0"
```

```
./composer.json has been updated
```

```
Loading composer repositories with package information
```

```
Updating dependencies (including require-dev)
```

```
- Installing react/promise (v2.2.0)
```

```
  Loading from cache
```

```
- Installing guzzlehttp/streams (3.0.0)
```

```
  Loading from cache
```

```
- Installing guzzlehttp/ringphp (1.1.0)
```

```
  Loading from cache
```

```
- Installing guzzlehttp/guzzle (5.3.0)
```

```
  Loading from cache
```

```
Writing lock file
```

```
Generating autoload files
```

```
Generating optimized class loader
```

Finally, edit `.env` and change the settings for Mailgun.

Mailgun Configuration in .env

```
MAIL_DRIVER=mailgun
MAIL_FROM=YOUR@EMAIL.COM
MAIL_NAME=YOUR-NAME
MAILGUN_DOMAIN=THE-DOMAIN-SETUP-IN-MAILGUN
MAILGUN_SECRET=THE-API-KEY-FOR-DOMAIN
```

The nice thing about Mailgun is that email is sent via an API, which is faster than using SMTP.

Testing Mail with Tinker

Laravel 5.1's emailer uses views to send the email. So let's first create a simple test view.

Create the `resources/views/emails` directory and within it create the `test.blade.php` file with the following content.

Content of emails.test view

```
1 <p>
2   This is a test, an email test.
3 </p>
4 <p>
5   The variable <code>$testVar</code> contains the value:
6 </p>
7 <ul>
8   <li><strong>{{ $testVar }}</strong></li>
9 </ul>
10 <hr>
11 <p>
12   That is all.
13 </p>
```

Now fire up the artisan `tinker` command and send an email to yourself as instructed below.

Testing Email with Tinker

```
~/Code/15beauty$ php artisan tinker
Psy Shell v0.4.4 (PHP 5.6.2 â€” cli) by Justin Hileman
>>> Mail::send('emails.test',
... ['testVar' => 'Just a silly test'],
... function($message) {
...     $message->to('YOUR@EMAIL.com')
...     ->subject('A simple test');
... });
=> 1
>>> exit
```

The first argument to `Mail::send()` is the view's name. The second is an array of any variables the view requires (and `emails.test` requires `$testVar`). The third is a closure to do additional processing on the message. Here we just set the to address and the subject line.

You can do many things in this closure. Here's just a few.

- `->from($address, $name = null)` - Add a from address to the message.
- `->sender($address, $name = null)` - Set the sender of the message.
- `->to($address, $name = null)` - Add a recipient to the message.
- `->cc($address, $name = null)` - Add a carbon copy recipient to the message.
- `->bcc($address, $name = null)` - Add a blind carbon copy.
- `->replyTo($address, $name = null)` - Add a reply-to recipient.
- `->subject($subject)` - Set the subject of the message.
- `->attach($file, array $options = [])` - Attach a file to the message.

The above tinker example used a Gmail configuration, which returns 1 indicating success. If you use the Mailgun driver in your configuration, a successful return value will look different.

Testing Email with Tinker (Mailgun config)

```
~/Code/15beauty$ php artisan tinker
Psy Shell v0.4.4 (PHP 5.6.2 â€” cli) by Justin Hileman
>>> Mail::send('emails.test',
... ['testVar' => 'Just a silly test'],
... function($message) {
...     $message->to('YOUR@EMAIL.com')
...     ->subject('A simple test');
... });
=> <GuzzleHttp\Message\Response #0000000024da8555000000017f74f2c8> {}
>>> exit
```

Adding a Contact Us Form

Now that we know Laravel's mailer will work, let's create a form to email us contact information from the user.

Adding the Link and Route

A link to the contact form should appear on every blog page. Edit the navbar partials view as below.

Changes to `blog.partials.page-nav View`

```
1 // Change the following area
2 {{-- Collect the nav links, forms, and other content for toggling --}}
3 <div class="collapse navbar-collapse" id="navbar-main">
4     <ul class="nav navbar-nav">
5         <li>
6             <a href="/">Home</a>
7         </li>
8     </ul>
9 </div>
10
11 // To the following
12 {{-- Collect the nav links, forms, and other content for toggling --}}
13 <div class="collapse navbar-collapse" id="navbar-main">
14     <ul class="nav navbar-nav">
15         <li>
16             <a href="/">Home</a>
17         </li>
18     </ul>
19     <ul class="nav navbar-nav navbar-right">
20         <li>
21             <a href="/contact">Contact</a>
22         </li>
23     </ul>
24 </div>
```

Easy, just a link to `/contact`.

Next create a route for `/contact`.

Changes to routes.php

```
1 // After the following line
2 get('blog/{slug}', 'BlogController@showPost');
3
4 // Add these two lines
5 $router->get('contact', 'ContactController@showForm');
6 Route::post('contact', 'ContactController@sendContactInfo');
```

Notice we used `$router` directly instead of the `get()` function. Then instead of the `post()` function, the `Route` facade was used. This is simply to illustrate there's multiple ways to set up the routes. Normally, I just use the helper functions directly, but some people prefer the `$router` variable or the `Route` facade.



Routing Shortcut Functions

Laravel 5.1 also provides the following shortcut functions for routing. Any of these can be used directly on the `$router` variable or with the `Route` facade.

- `delete($uri, $action)` registers a new DELETE route.
- `get($uri, $action)` registers a new GET route.
- `patch($uri, $action)` registers a new PATCH route.
- `post($uri, $action)` registers a new POST route.
- `put($uri, $action)` registers a new PUT route.
- `resource($name, $controller, $options)` registers a resource controller.

When you group routes together, there's no short helper function so either `Route::group()` or `$router->group()` must be used.

Creating the FormRequest

We know the contact form will contain a name, email address, and a message. The Laravel 5.1 “way” to validate forms is through `FormRequest` objects which we used quite a bit in the administration area of our blog.

Let's create the `FormRequest` now, so it's all ready when we build the controller. First use `artisan` to create the skeleton.

Creating the FormRequest with Artisan

- 1 ~/Code/15beauty\$ php artisan make:request ContactMeRequest
 - 2 Request created successfully.
-

Update its contents to match what's below.

Content of ContactMeRequest.php

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

class ContactMeRequest extends Request
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     */
    public function rules()
    {
        return [
            'name' => 'required',
            'email' => 'required|email',
            'message' => 'required',
        ];
    }
}
```

No need to comment on the request. After building the administration side of this blog you should be thoroughly familiar with Form Requests.

Adding the Controller

Let's create the controller we specified in the routes file.

First create the skeleton of the controller with artisan.

Creating ContactController with Artisan

```
~/Code/15beauty$ php artisan make:controller --plain ContactController  
Controller created successfully.
```

Make the content of ContactController.php match what's below.

Content of ContactController.php

```
1  <?php  
2  namespace App\Http\Controllers;  
3  
4  use App\Http\Requests\ContactMeRequest;  
5  use Illuminate\Support\Facades\Mail;  
6  
7  class ContactController extends Controller  
8  {  
9      /**  
10     * Show the form  
11     *  
12     * @return View  
13     */  
14     public function showForm()  
15     {  
16         return view('blog.contact');  
17     }  
18  
19     /**  
20     * Email the contact request  
21     *  
22     * @param ContactMeRequest $request  
23     * @return Redirect  
24     */  
25     public function sendContactInfo(ContactMeRequest $request)  
26     {  
27         $data = $request->only('name', 'email', 'phone');  
28         $data['messageLines'] = explode("\n", $request->get('message'));  
29  
30         Mail::send('emails.contact', $data, function ($message) use ($data) {  
31             $message->subject('Blog Contact Form: '.$data['name'])  
32                 ->to(config('blog.contact_email'))  
33                 ->replyTo($data['email']);  
34         });
```

```

35
36     return back()
37         ->withSuccess("Thank you for your message. It has been sent.");
38     }
39 }

```

In the `sendContactInfo()` method, we use the `ContactMeRequest` to validate. Then we fill `$data` with the form fields. For the message field, we break the message into individual lines to pass to the view as `messageLines`.

Then we use the `Mail` facade to send the message. You could optionally have the `sendContactInfo()` method take an `Illuminate\Mail\Mailer` object as an argument (Laravel 5.1 is smart enough to automatically inject it) and use this object to send mail.

See the [Facade Class Reference](http://laravel.com/docs/5.1/facades#facade-class-reference)³⁴ in the Laravel 5.1 documentation for a list of facades and the equivalent class to use if you want to access the instance directly.

After the message is sent, we redirect back to the contact page, flashing a success message.

ADD `contact_email` to your blog config file.

It is needed in the controller just created (`config('blog.contact_email')`). It's easy to add this configuration value, just edit `config/blog.php` and add an additional option.

Creating the Views

Two views need to be created for the contact form. The one which will display the form and the one that formats the email to be sent.

Create `contact.blade.php` in the `resources/views/blog` directory.

Content of `blog.contact` View

```

1  @extends('blog.layouts.master', ['meta_description' => 'Contact Form'])
2
3  @section('page-header')
4      <header class="intro-header"
5          style="background-image: url('{{ page_image('contact-bg.jpg') }}')">
6          <div class="container">
7              <div class="row">
8                  <div class="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
9                      <div class="site-heading">
10                         <h1>Contact Me</h1>
11                         <hr class="small">

```

³⁴<http://laravel.com/docs/5.1/facades#facade-class-reference>

```

12         <h2 class="subheading">
13             Have questions? I have answers (maybe).
14         </h2>
15     </div>
16 </div>
17 </div>
18 </div>
19 </header>
20 @stop
21
22 @section('content')
23     <div class="container">
24         <div class="row">
25             <div class="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
26                 @include('admin.partials.errors')
27                 @include('admin.partials.success')
28                 <p>
29                     Want to get in touch with me? Fill out the form below to send me a
30                     message and I will try to get back to you within 24 hours!
31                 </p>
32                 <form action="/contact" method="post">
33                     <input type="hidden" name="_token" value="{!! csrf_token() !!}">
34                     <div class="row control-group">
35                         <div class="form-group col-xs-12">
36                             <label for="name">Name</label>
37                             <input type="text" class="form-control" id="name" name="name"
38                                 value="{{ old('name') }}">
39                         </div>
40                     </div>
41                     <div class="row control-group">
42                         <div class="form-group col-xs-12">
43                             <label for="email">Email Address</label>
44                             <input type="email" class="form-control" id="email" name="email"
45                                 value="{{ old('email') }}">
46                         </div>
47                     </div>
48                     <div class="row control-group">
49                         <div class="form-group col-xs-12 controls">
50                             <label for="phone">Phone Number</label>
51                             <input type="tel" class="form-control" id="phone" name="phone"
52                                 value="{{ old('phone') }}">
53                         </div>

```

```

54         </div>
55         <div class="row control-group">
56             <div class="form-group col-xs-12 controls">
57                 <label for="message">Message</label>
58                 <textarea rows="5" class="form-control" id="message"
59                     name="message">{{ old('message') }}</textarea>
60             </div>
61         </div>
62         <br>
63         <div class="row">
64             <div class="form-group col-xs-12">
65                 <button type="submit" class="btn btn-default">Send</button>
66             </div>
67         </div>
68     </form>
69 </div>
70 </div>
71 </div>
72 @endsection

```

The `blog.contact` view should be easy to follow. Notice how we included the **errors** and **success** partials from the administration area? They perfectly fit what was needed.

Now create `contact.blade.php` in the `resources/views/emails` directory. This is what will format the email we'll send to ourselves.

Content of `email.contact` view

```

1 <p>
2     You have received a new message from your website contact form.
3 </p>
4 <p>
5     Here are the details:
6 </p>
7 <ul>
8     <li>Name: <strong>{{ $name }}</strong></li>
9     <li>Email: <strong>{{ $email }}</strong></li>
10    <li>Phone: <strong>{{ $phone }}</strong></li>
11 </ul>
12 <hr>
13 <p>
14     @foreach ($messageLines as $messageLine)
15         {{ $messageLine }}<br>

```

```

16   @endforeach
17 </p>
18 <hr>

```

Nothing too fancy there. Just a smidge of formatting.

Sending the Mail

The contact form should now be fully functional. Fire up your web browser, point it to your `http://15beauty.app` project and test it out.

You may notice there's a slight delay between clicking **[Send]** and receiving a response of success back. This delay can be especially long when you're using the `smtp` mail driver.

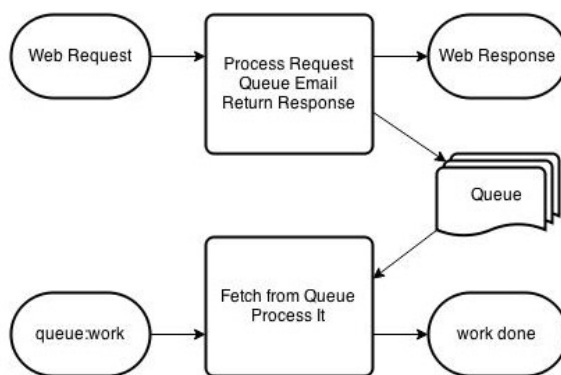
Does this delay really need to be there? The answer is no, not if we set up a queue to handle running tasks in the back ground.

About Queues

Queues allow you to defer the processing of time consuming tasks, such as emails. This allows your web requests to respond quicker to the user.

How they Work

Queues are simple to understand.



Queue Flow

A web request hits the controller where it's processed. During the processing something is added to the queue (an Email in this figure) and then the response is returned.

Somewhere in the background a queue worker runs. It fetches the next thing from the queue and processes it.

Bam!

The Different Queue Drivers

Laravel provides drivers for several different queue implementation. They are:

- **sync** - The sync driver effectively short circuits the entire queuing process. When something is queued and the sync driver is being used, the item is fully processed immediately and *synchronously*.
- **database** - The database driver stores queued items in the local database. Specifically, in a jobs table.
- **beanstalkd** - The beanstalkd driver expects **beanstalkd** to be configured and running. You'll also have to do a `composer require "pda/pheanstalk=~3.0"` to use it.
- **sqs** - The sqs driver will queue to your **Amazon SQS** queue. Also composer require `aws/aws-sdk-php` is required to use it.
- **iron** - The iron driver will queue to your **IronMQ** account and composer required `"iron-io/iron_mq=~1.5"` is required.
- **redis** - The redis driver will store queued items in the Redis database. It requires composer require `"presdis/predis=~1.0"` to operate.

Using the Database Driver

We'll be using the Database Driver for our queue. To do this we'll need to create the jobs table and run migrations as instructed below.

Creating and Running Queue Jobs Migration

```
vagrant@homestead:~/Code/15beauty$ php artisan queue:table
Migration created successfully!
vagrant@homestead:~/Code/15beauty$ php artisan migrate
Migrated: 2015_06_06_130436_create_jobs_table
```

Finally, edit `.env` and change the `QUEUE_DRIVER` setting from `sync` to `database`.

Queuing the Contact Us Email

Now that the queue is set up, we're ready to queue emails from the Contact Us form instead of waiting for the delivery to finish before responding to the user.

Changing the Controller

To queue the email, there's only a single small change to make. Update your `ContactController` class as instructed.

Change to ContractController.php

```
1 // Find the line below
2 Mail::send('emails.contact', $data, function ($message) use ($data) {
3
4 // And change it to match what's below
5 Mail::queue('emails.contact', $data, function ($message) use ($data) {
```

That's it! Instead of `Mail::send()` we call `Mail::queue()` and Laravel will automatically queue it for us.

Where's the Email

Test out the Contact Us form again. After you click *[Send]* there should be no delay before you see the **Success** message.

Still, you can wait forever for the email and it will never arrive.

Why?

Because there's no process running in the background, watching for and handling items arriving in the queue.

Running queue:work

To process the next item on the queue, we can manually run artisan's `queue:work` command.

This command will do nothing if the queue is empty. But if there's an item on the queue it will fetch the item and attempt to execute it.

Running Artisan queue:work

```
vagrant@homestead:~/Code/15beauty$ php artisan queue:work
Processed: mailer@handleQueuedMessage
```

As you can see here it handled the queued email message. Now the email should arrive in your inbox within moments.

Automatically Processing the Queue

Of course, having to manually log into our server and run the `artisan queue:work` each time we want to process the next item on the queue is ridiculous.

There's a few options to automate this.

One is load up `artisan queue:listen` in the startup scripts of your server. This command automatically calls `artisan queue:work` when items appear in the queue.

The problem with this technique is something will invariably happen. The `queue:listen` command will hang. Or it will stop running. A better way to run `queue:listen` is with **supervisord**.

Running queue:listen with supervisord

supervisord is a *nix utility to monitor and control processes. We're not delving into how to install this utility, but if you have it and get it installed, below is a portion of `/etc/supervisord.conf` that works well.

Portion of `supervisord.conf` for `queue:listen`

```
[program:l5beauty-queue-listen]
command=php /PATH/T0/l5beauty/artisan queue:listen
user=NONROOT-USER
process_name=%(program_name)s_%(process_num)d
directory=/PATH/T0/l5beauty
stdout_logfile=/PATH/T0/l5beauty/storage/logs/supervisord.log
redirect_stderr=true
numprocs=1
```

You'll need to replace the `/PATH/T0/` to match your local install. Likewise, the `user` setting will be unique to your installation.

Using a Scheduled Command

Another option for low volume sites is to schedule `queue:work` to run every minute. Or even every 5 minutes. This is best done using Laravel 5.1's command scheduler.

Edit `app/Console/Kernel.php` and make the changes below.

Editing Console Kernel

```
1 // Replace the following method
2 /**
3  * Define the application's command schedule.
4  *
5  * @param Schedule $schedule
6  * @return void
7  */
8 protected function schedule(Schedule $schedule)
9 {
10     // Run once a minute
11     $schedule->command('queue:work')->cron('* * * * *');
12 }
```

This will run the `queue:work` command once a minute. You can change this frequency in many ways.

Various Run Frequencies in Console Kernel

```
// Run every 5 minutes
$schedule->command('queue:work')->everyFiveMinutes();

// Run once a day
$schedule->command('queue:work')->daily();

// Run Mondays at 8:15am
$schedule->command('queue:work')->weeklyOn(1, '8:15');
```

The second step in setting up the scheduled command is to modify your machine's crontab. Edit crontab and add the following line.

Crontab Line for Artisan Scheduler

```
* * * * * php /path/to/artisan schedule:run 1>> /dev/null 2>&1
```

This will call artisan to run anything currently scheduled, sending any output to the null device.

Queuing Jobs

Another great use for queues are asynchronous jobs. These are jobs you execute as normal with `$this->dispatch(new JobName)` from your controller, but they'll simply be placed in the queue to be run later by `queue:work` or whatever method is processing queue items in your application.

Here's how to do it.

First, when creating the job class, use the `--queued` option.

Example of a Queued Job

```
~/Projects/newbeauty$ php artisan make:job --queued TestJob
Job created successfully.
```

Now, if you examine the template Laravel 5.1 created for `TestJob` you'll notice few small changes at the top.

Difference in Queued Jobs

```
1 // These three use statements are new
2 use Illuminate\Queue\SerializesModels;
3 use Illuminate\Queue\InteractsWithQueue;
4 use Illuminate\Contracts\Queue\ShouldQueue;
5
6 // The class will also implement ShouldQueue
7 class TestJob extends Job implements SelfHandling, ShouldQueue
8 {
9
10 // And the class uses two traits
11     use InteractsWithQueue, SerializesModels;
```

ShouldQueue

By having the `TestJob` class implement `ShouldQueue`, the `handle()` method won't be called. Instead, `TestJob` will be constructed and the instance will be pushed onto the queue. When the item is processed from the queue, then the `handle()` method will be called..

InteractsWithQueue

This will make several queue interaction methods available such as `$this->delete()` to delete the item from the queue or `$this->release()` to release the item back onto the queue. Normally you won't need these methods.

SerializesModels

When the job is serialized to be placed on the queue, this Trait will look for properties of the Job that are models and serialize them correctly.

Queued Jobs are an excellent way to run time consuming processes which you don't want the user to have to wait for.

Recap

The main thing accomplished in this chapter was adding a **Contact** form to the blog, but we covered several interesting topics to do it. We talked about sending mail with Mailgun and testing mail with Tinker. Several alternative routing methods were presented.

We discussed queues and set up a database queue. Finally, contact emails were sent through the queue.

A pretty solid chapter.

Chapter 15 - Adding Comments, RSS, and a Site Map

In this chapter we'll add comments and social links to the blog. Then we'll create a RSS feed for the Laravel 5.1 Beauty blog. Finally, we'll add a Site Map which finishes the **l5beauty** project.

Contents

- [The Problem with Comments](#)
- [Adding Disqus Comments](#)
 - [Creating a Disqus account](#)
 - [Passing the Slug to the page](#)
 - [Creating the Disqus Partial](#)
 - [Updating the Footer](#)
- [Adding Social Links](#)
- [Creating a RSS Feed](#)
 - [Pulling in the Composer Package](#)
 - [Creating the RSS Feed Service](#)
 - [Updating the Blog Configuration](#)
 - [Adding rss Route, Link, and Method](#)
- [Create a Site Map](#)
 - [Creating the SiteMap Service](#)
 - [Adding the sitemap.xml Route and Method](#)
- [Recap](#)

The Problem with Comments

The main thing missing from the project right now is the ability for users to leave comments. Unfortunately, blog comments have multiple issues with them.

First of all there's the moderation, approving, and general management of the comments. Yes, with Laravel 5.1 we could add this functionality to the blog's administration, allowing users to sign up, allowing the users to make comments and so forth. It'd be easy enough to create management screens to handle all of this.

But the real problem with comments is SPAM.

How do we deal with that? **CAPTCHA** or **reCAPTCHA**? Blacklists and/or Whitelists? Creating a SPAM Honeypot trap like Maksim Surguy? Or by integrating **Akismet**?

Frankly, I don't want the headaches. I want to use a proven, third-party solution that will keep my involvement to a minimum.

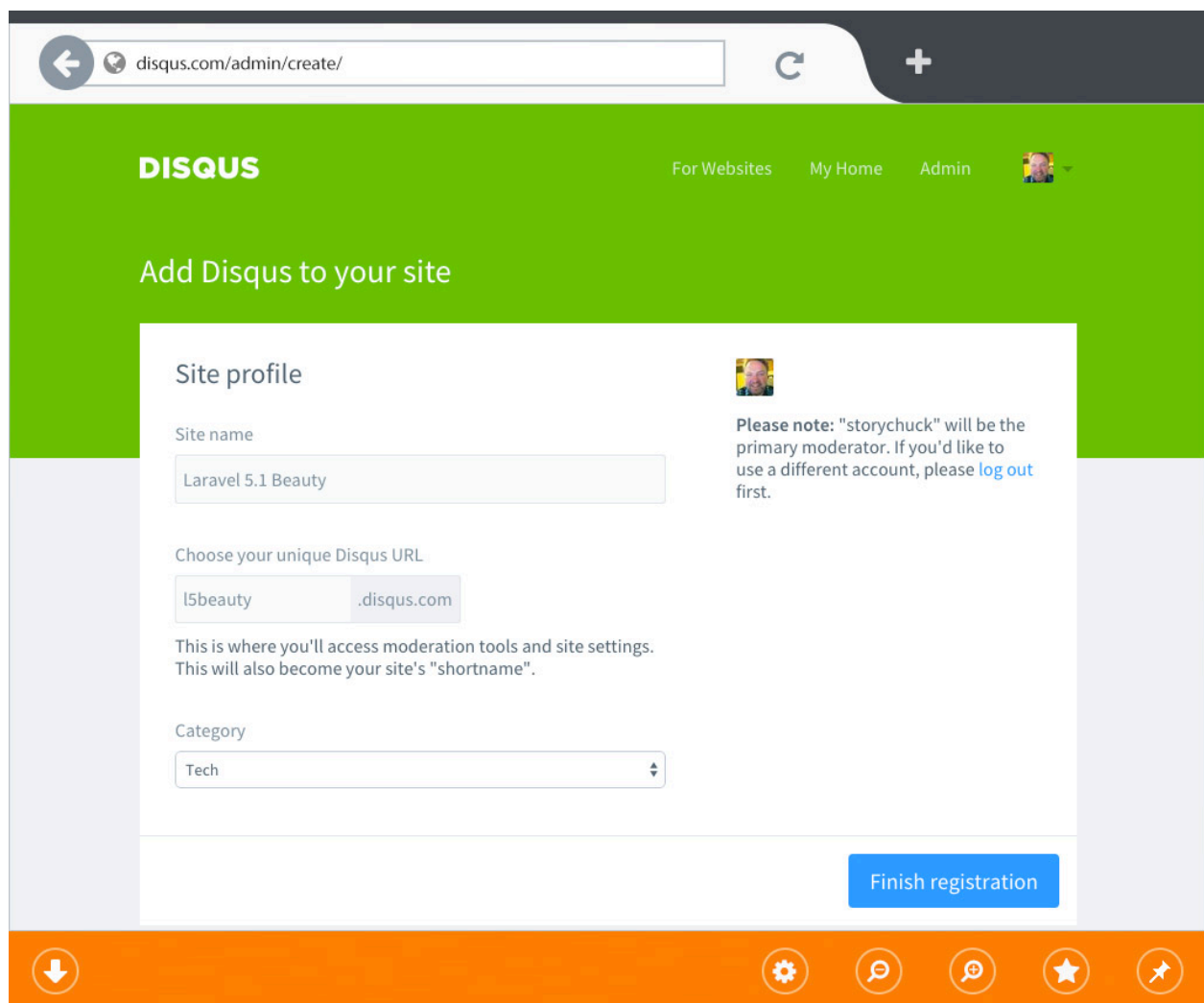
Adding Disqus Comments

We'll use **Disqus** for comments on the Laravel 5.1 Beauty blog.

Creating a Disqus account

Head over to `disqus.com` and sign up for a free Disqus account. Once you have an account, you'll want to click on the link to **Add Disqus to your site** *(If you can't find the link, try clicking on the gear icon at the top right of the screen.)*

You should see an image like the one below.



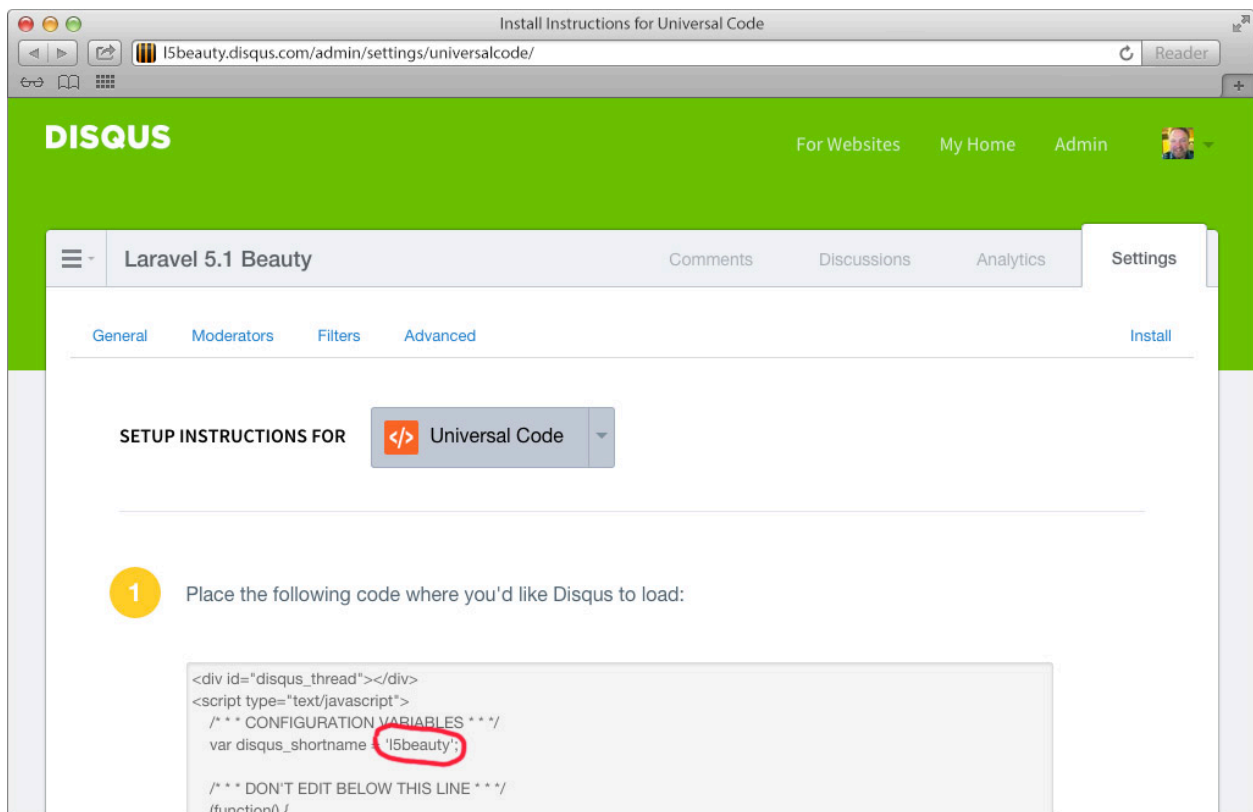
The screenshot shows a web browser window with the address bar displaying `disqus.com/admin/create/`. The page has a green header with the Disqus logo and navigation links: "For Websites", "My Home", "Admin", and a user profile icon. The main heading is "Add Disqus to your site". Below this is a "Site profile" form with the following fields:

- Site name:** A text input field containing "Laravel 5.1 Beauty".
- Choose your unique Disqus URL:** Two input fields, the first containing "l5beauty" and the second containing ".disqus.com".
- Category:** A dropdown menu with "Tech" selected.

To the right of the form, there is a note: "Please note: 'storychuck' will be the primary moderator. If you'd like to use a different account, please [log out](#) first." A blue "Finish registration" button is located at the bottom right of the form. The browser's address bar and a mobile-style navigation bar with icons are also visible.

Add Disqus to your site

After you fill out the simple form, choose **Universal Code** from the next screen. Then you'll see a screen like the one below.



Disqus Universal Code

Note the value of the variable `disqus_shortcode` in this code.

Passing the Slug to the page

Next we'll add the variable `$slug` to the pages showing posts. To do this modify the one line in the `showPost()` method of `BlogController` as instructed below.

Update to BlogController

```
1 // Change the line below
2     return view($post->layout, compact('post', 'tag'));
3
4 // to this
5     return view($post->layout, compact('post', 'tag', 'slug'));
```

Easy! Now the post view will have an additional variable.

Creating the Disqus Partial

Create a new file in the `resources/views/blog/partials` named `disqus.blade.php` with the content below.

Content of blog.partials.disqus view

```

1  @if (App::environment() === 'production')
2      <div id="disqus_thread"></div>
3      <script type="text/javascript">
4          var disqus_shortcode = 'SHORTNAME HERE';
5          @if (isset($slug))
6              var disqus_identifier = 'blog-{{ $slug }}';
7          @endif
8          /* This comment removes bogus PhpStorm error */
9          (function() {
10             var dsq = document.createElement('script');
11             dsq.type = 'text/javascript';
12             dsq.async = true;
13             dsq.src = '//' + disqus_shortcode + '.disqus.com/embed.js';
14             (document.getElementsByTagName('head')[0] ||
15              document.getElementsByTagName('body')[0]).appendChild(dsq);
16         })();
17     </script>
18     <noscript>
19         Please enable JavaScript to view the
20         <a href="http://disqus.com/?ref_noscript">comments powered by Disqus.</a>
21     </noscript>
22     <a href="http://disqus.com" class="dsq-brlink">
23         comments powered by <span class="logo-disqus">Disqus</span>
24     </a>
25 @endif

```

Be sure to use the correct shortcode noted earlier for **SHORTNAME HERE** in the above snippet.

If `$slug` is set then the Disqus identifier will group all the comments together for that page. `$slug` is now set for all post pages.

You'll also note that the whole template is wrapped in a big `@if` statement. This is because I didn't want to show comments when I'm working on the blog locally. Feel free to change this to suit your needs.

Updating the Footer

Finally, update the page-footer.blade.php in the `resources/views/blog/partials` directory to match what's below.

Updated blog.partials.page-footer

```

1 <hr>
2 <div class="container">
3   <div class="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
4     @include('blog.partials.disqus')
5   </div>
6 </div>
7 <hr>
8 <footer>
9   <div class="container">
10    <div class="row">
11      <div class="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
12        <p class="copyright">Copyright &copy; {{ config('blog.author') }}</p>
13      </div>
14    </div>
15  </div>
16 </footer>

```

Voila! Your blog now has comments.

Adding Social Links

Let's make one, final, finishing touch in the footer and add links to your facebook account, twitter account, and so forth.

Edit page-footer.blade.php again, updating it to the final version below.

Final Version of blog.partials.page-footer view

```

1 <hr>
2 <div class="container">
3   <div class="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
4     @include('blog.partials.disqus')
5   </div>
6 </div>
7 <hr>
8 <footer>
9   <div class="container">
10    <div class="row">
11      <div class="col-lg-8 col-lg-offset-2 col-md-10 col-md-offset-1">
12        <ul class="list-inline text-center">

```

```

13     <li>
14         <a href="{{ url('rss') }}" data-toggle="tooltip"
15             title="RSS feed">
16             <span class="fa-stack fa-lg">
17                 <i class="fa fa-circle fa-stack-2x"></i>
18                 <i class="fa fa-rss fa-stack-1x fa-inverse"></i>
19             </span>
20         </a>
21     </li>
22     <li>
23         <a href="https://twitter.com/PERSONALIZE" data-toggle="tooltip"
24             title="My Twitter Page">
25             <span class="fa-stack fa-lg">
26                 <i class="fa fa-circle fa-stack-2x"></i>
27                 <i class="fa fa-twitter fa-stack-1x fa-inverse"></i>
28             </span>
29         </a>
30     </li>
31     <li>
32         <a href="https://www.facebook.com/PERSONALIZE" data-toggle="tooltip"
33             title="My Facebook Page">
34             <span class="fa-stack fa-lg">
35                 <i class="fa fa-circle fa-stack-2x"></i>
36                 <i class="fa fa-facebook fa-stack-1x fa-inverse"></i>
37             </span>
38         </a>
39     </li>
40     <li>
41         <a href="https://www.google.com/+PERSONALIZE" data-toggle="tooltip"
42             title="My Google+ Page">
43             <span class="fa-stack fa-lg">
44                 <i class="fa fa-circle fa-stack-2x"></i>
45                 <i class="fa fa-google-plus fa-stack-1x fa-inverse"></i>
46             </span>
47         </a>
48     </li>
49     <li>
50         <a href="http://www.linkedin.com/in/PERSONALIZE/"
51             data-toggle="tooltip" title="My LinkedIn Page">
52             <span class="fa-stack fa-lg">
53                 <i class="fa fa-circle fa-stack-2x"></i>
54                 <i class="fa fa-linkedin fa-stack-1x fa-inverse"></i>

```

```

55         </span>
56     </a>
57 </li>
58 <li>
59     <a href="https://github.com/PERSONALIZE" data-toggle="tooltip"
60         title="My GitHub Pages">
61         <span class="fa-stack fa-lg">
62             <i class="fa fa-circle fa-stack-2x"></i>
63             <i class="fa fa-github fa-stack-1x fa-inverse"></i>
64         </span>
65     </a>
66 </li>
67 </ul>
68 <p class="copyright">Copyright &copy; {{ config('blog.author') }}</p>
69 </div>
70 </div>
71 </div>
72 </footer>

```

Be sure to update all those **PERSONALIZE** values with your own settings. If you don't have a particular social account, just remove the entire item from the list.

Did you notice that first one? The RSS feed? That's not yet created, so let's do it next.

Creating a RSS Feed

A RSS feed is a must for any blogging application. Creating one in Laravel 5.1 for the **L5Beauty** application is quick and easy.

Pulling in the Composer Package

We'll use the `suin/php-rss-writer` package to make creating RSS files easy.

The first step is to pull in the package with composer.

Requiring the suin/php-rss-writer Package with Composer

```
~/Code/15beauty$ composer require suin/php-rss-writer
Using version ^1.3 for suin/php-rss-writer
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing suin/php-rss-writer (1.3)
  Downloading: 100%

Writing lock file
Generating autoload files
Generating optimized class loader
```

Creating the RSS Feed Service

Let's create a service class to create and return the RSS feed. In the app/Services directory create the file RssFeed.php with the following content.

Content of RssFeed.php

```
1 <?php
2
3 namespace App\Services;
4
5 use App\Post;
6 use Carbon\Carbon;
7 use Illuminate\Support\Facades\Cache;
8 use Suin\RssWriter\Channel;
9 use Suin\RssWriter\Feed;
10 use Suin\RssWriter\Item;
11
12 class RssFeed
13 {
14     /**
15      * Return the content of the RSS feed
16      */
17     public function getRSS()
18     {
19         if (Cache::has('rss-feed')) {
20             return Cache::get('rss-feed');
21         }
22     }
23 }
```

```
22
23     $rss = $this->buildRssData();
24     Cache::add('rss-feed', $rss, 120);
25
26     return $rss;
27 }
28
29 /**
30  * Return a string with the feed data
31  *
32  * @return string
33  */
34 protected function buildRssData()
35 {
36     $now = Carbon::now();
37     $feed = new Feed();
38     $channel = new Channel();
39     $channel
40         ->title(config('blog.title'))
41         ->description(config('blog.description'))
42         ->url(url())
43         ->language('en')
44         ->copyright('Copyright (c) '.config('blog.author'))
45         ->lastBuildDate($now->timestamp)
46         ->appendTo($feed);
47
48     $posts = Post::where('published_at', '<=', $now)
49         ->where('is_draft', 0)
50         ->orderBy('published_at', 'desc')
51         ->take(config('blog.rss_size'))
52         ->get();
53     foreach ($posts as $post) {
54         $item = new Item();
55         $item
56             ->title($post->title)
57             ->description($post->subtitle)
58             ->url($post->url())
59             ->pubDate($post->published_at->timestamp)
60             ->guid($post->url(), true)
61             ->appendTo($channel);
62     }
63 }
```

```

64     $feed = (string)$feed;
65
66     // Replace a couple items to make the feed more compliant
67     $feed = str_replace(
68         '<rss version="2.0">',
69         '<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">',
70         $feed
71     );
72     $feed = str_replace(
73         '<channel>',
74         '<channel>'. "\n". '    <atom:link href="'.url('/rss').
75         '" rel="self" type="application/rss+xml" />',
76         $feed
77     );
78
79     return $feed;
80 }
81 }

```

getRSS()

This method returns the entire feed as a string. We cache the results for 2 hours so the feed isn't constantly being built.

buildRSSData()

This method build the feed itself from the posts table.

Updating the Blog Configuration

Add the `rss_size` entry to `config/blog.php` as shown below. We used this config value in the `RSSFeed` service class to determining how many posts to allow in the feed.

Content of config/blog.php

```

1 <?php
2 return [
3     'name' => "L5 Beauty",
4     'title' => "Laravel 5.1 Beauty",
5     'subtitle' => 'A clean blog written in Laravel 5.1',
6     'description' => 'This is my meta description',
7     'author' => 'Chuck Heintzelman',
8     'page_image' => 'home-bg.jpg',
9     'posts_per_page' => 10,
10    'rss_size' => 25,

```

```
11     'contact_email' => env('MAIL_FROM'),
12     'uploads' => [
13         'storage' => 'local',
14         'webpath' => '/uploads/',
15     ],
16 ];
```

Adding rss Route, Link, and Method

Only three things left to implement the RSS feed. First add a route in `app/Http/routes.php` as instructed below.

Adding the rss Route

```
1 // After the following line
2 Route::post('contact', 'ContactController@sendContactInfo');
3
4 // Add the new route
5 get('rss', 'BlogController@rss');
```

Next, Update the `blog.layouts.master` view as instructed below.

Update to `blog.layouts.master` View

```
1 // Change the line below
2 <title>{{ $title or config('blog.title') }}</title>
3
4 // To the following
5 <title>{{ $title or config('blog.title') }}</title>
6
7 <link rel="alternate" type="application/rss+xml" href="{{ url('rss') }}"
8     title="RSS Feed {{ config('blog.title') }}">
```

Finally, update `BlogController` as instructed below.

BlogController updates for the RSS Feed

```
1 // Add the following use statement at the top of the file
2 use App\Services\RssFeed;
3
4 // Add the following method
5 public function rss(RssFeed $feed)
6 {
7     $rss = $feed->getRSS();
8
9     return response($rss)
10         ->header('Content-type', 'application/rss+xml');
11 }
```

That's it. Point your browser to `http://15beauty.app/rss` and you'll see the feed.

Create a Site Map

To keep our blog search engine friendly we'll add one final feature to the blog ... a Site Map.

We'll use the same technique as we did with the RSS Feed, to build it on-the-fly, but cache the results so it's only rebuilt a maximum of once every couple hours.

Creating the SiteMap Service

Create a new file named `SiteMap.php` in the `app/Services` directory with the content below.

Content of SiteMap.php

```
1 <?php
2
3 namespace App\Services;
4
5 use App\Post;
6 use Carbon\Carbon;
7 use Illuminate\Support\Facades\Cache;
8
9 class SiteMap
10 {
11     /**
12      * Return the content of the Site Map
13      */
```

```

14 public function getSiteMap()
15 {
16     if (Cache::has('site-map')) {
17         return Cache::get('site-map');
18     }
19
20     $siteMap = $this->buildSiteMap();
21     Cache::add('site-map', $siteMap, 120);
22     return $siteMap;
23 }
24
25 /**
26  * Build the Site Map
27  */
28 protected function buildSiteMap()
29 {
30     $postsInfo = $this->getPostsInfo();
31     $dates = array_values($postsInfo);
32     sort($dates);
33     $lastmod = last($dates);
34     $url = trim(url(), '/') . '/';
35
36     $xml = [];
37     $xml[] = '<?xml version="1.0" encoding="UTF-8"?>';
38     $xml[] = '<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">';
39     $xml[] = '    <url>';
40     $xml[] = '        <loc>$url</loc>';
41     $xml[] = '        <lastmod>$lastmod</lastmod>';
42     $xml[] = '        <changefreq>daily</changefreq>';
43     $xml[] = '        <priority>0.8</priority>';
44     $xml[] = '    </url>';
45
46     foreach ($postsInfo as $slug => $lastmod) {
47         $xml[] = '    <url>';
48         $xml[] = '        <loc>{$url}blog/$slug</loc>';
49         $xml[] = '        <lastmod>$lastmod</lastmod>';
50         $xml[] = '    </url>';
51     }
52
53     $xml[] = '</urlset>';
54
55     return join("\n", $xml);

```

```

56     }
57
58     /**
59      * Return all the posts as $url => $date
60      */
61     protected function getPostsInfo()
62     {
63         return Post::where('published_at', '<=', Carbon::now())
64             ->where('is_draft', 0)
65             ->orderBy('published_at', 'desc')
66             ->lists('updated_at', 'slug')
67             ->all();
68     }
69 }

```

Adding the sitemap.xml Route and Method

First edit your routes file as instructed below.

Addition of sitemap.xml route

```

1  // After this line
2  get('rss', 'BlogController@rss');
3
4  // Add the following line
5  get('sitemap.xml', 'BlogController@siteMap');

```

Then update the BlogController as instructed below.

Updates to BlogController for SiteMap

```

1  // Add the use below to the top of the file
2  use App\Services\SiteMap;
3
4  // Add the following method
5  public function siteMap(SiteMap $siteMap)
6  {
7      $map = $siteMap->getSiteMap();
8
9      return response($map)
10         ->header('Content-type', 'text/xml');
11 }

```

And point your browser to `http://15beauty.app/sitemap.xml` to make sure it works.



It's A Wrap!

That's the end of the **15beauty** project. You've developed a complete blogging system in Laravel 5.1 and hopefully have learned a lot along the way!

Recap

This chapter put all the final pieces together for the blog. We started using Disqus for comments and then created a RSS Feed for the project. Finally, the project was finished by adding a Site Map.

There's one more chapter left. It contains a review of what was learned and some general topics about features in Laravel 5.1.

Chapter 16 - General Recap and Looking Forward

This is the end of the book. The **L5Beauty** project is complete. You may be thinking *Now What?* After a few chapters on setting up Laravel and Homestead, and one chapter on basic testing, this book has been all about writing the the blogging application and often with a minimum of explanation. The focus has been building and finishing the application.

After all, the fastest way to learn to code is to write code.

But now it's time to slow down, look back, and think about what's been built and where to go from here.

This chapter presents a mishmash of Laravel 5.1 topics. Some areas discussed below were used in developing the blogging application but others were not needed in the application.

Contents

- [Testing](#)
- [Eloquent Models and the Fluent Query Builder](#)
- [Advanced Routing](#)
- [Migrations, Seeding, and Model Factories](#)
- [Dependency and Method Injection](#)
- [Facades vs. helpers vs. IoC objects](#)
- [Laravel Elixir](#)
- [Tinker](#)
- [Artisan Commands](#)
- [Events](#)
- [Form Requests](#)
- [Blade Template Engine](#)
- [Flysystem](#)
- [Queues](#)
- [Blog Features to Add](#)
- [Final Recap and Thank You](#)

Testing

Laravel 5.1 was designed with testing in mind. We touched on testing briefly back in Chapter 6 - Testing and even provided a fairly complete list of testing methods available. But the subject of testing in Laravel 5.1 could easily fill its own book.

Here's just a few subjects about testing that can be explored:

- Mocking Facades, Jobs, and Events
- Using Model Factories when testing
- Testing Laravel 5.1 JSON APIs
- Using Sessions when testing
- Using DatabaseTransactions and DatabaseMigrations
- Disabling Middleware when testing
- Behat and other testing frameworks

Those are topics just off the top of my head. If I thought about it more, this list could double in size.

The best place to start learning about testing in Laravel 5.1 is the official documentation at `laravel.com`.

Eloquent Models and the Fluent Query Builder

Like the topic of testing, Eloquent Models and the Query Builder could easily fill its own book. The Models are an ActiveRecord implementation which work across a variety of databases. The Query Builder allows you to use a fluent interface to develop SQL queries.

We used Eloquent Models throughout this entire book when we created the Tag and Post models, and when we made use of the User model. We even set up a many-to-many relationship between these two models using a pivot table.

The query builder was used when we looked for the next or previous posts. Or in the BlogIndexData job when we paginated across a group of posts.

The database documentation at `laravel.com` was re-written for Laravel 5 and provides an excellent primer on what can be achieved in Laravel 5.1.

Advanced Routing

You can't really write a Laravel 5.1 web application without doing some routing. It's the glue that ties web requests to the code to execute. We did some basic routing, used the **auth** middleware, and even **grouped** some of our routes together in the **l5beauty** project.

But there's much more you can do with routing. Here's a partial list.

- You can specify a route has optional parameters.
- You can have sub-domains have their own group of routes.
- You can *name* individual routes, allowing your application to refer to the route by name.
- You can even constrain route parameters using regular expressions.

Migrations, Seeding, and Model Factories

As stated earlier in this book, Migrations are like version control for your database schemas. As such, I believe they *must be used* for any Laravel 5.1 application which has a database.

They're especially important when a team of developers works on a project.

We used migrations to create the database for the blog. We also made use of Model Factories in the Database seeds in order to create some *fake* data for the blog.

But we only touched on the few of the methods provided by the *Faker* library.

Point your browser to github.com/fzaninotto/Faker³⁵ to see a complete list of what you can do with this excellent library.

Dependency and Method Injection

One of the most powerful features of Laravel 5.1 is how it can automatically inject dependencies into your class constructors and controller methods.

Let's say you have a controller that looks like the snippet below:

```
class MyController extends Controller
{
    public function __construct(MyRepository $repository)
    {
        ...
    }

    public function someMethod(MyCoolClass $object)
    {
        ...
    }
}
```

³⁵<https://github.com/fzaninotto/Faker>

When Laravel 5.1 needs to construct `MyController` it will see the `$repository` argument and if it can construct a `MyRepository` class it will and pass the newly constructed instance to the `MyController` constructor.

Automatically! How cool is that?

If you were to try it using straight PHP code like below you'll get an error.

```
$controller = new App\Http\Controllers\MyController;
```

But, if you let Laravel 5.1 construct it, this will work.

```
$controller = app('App\Http\Controllers\MyController');
```

Likewise, Laravel 5.1 will examine the controller methods and automatically inject classes it can figure out how to construct.

So, if you have a route pointing to the `someMethod()` method of the example above, and Laravel can determine how to construct `MyCoolClass`, it'll inject an instance to the method when it calls it.

Facades vs. helpers vs. IoC objects

Often there's concern whether to use Facades, helpers, or IoC objects within your code. For example, to access the currently authenticated user you could:

- Use the Auth facade to access `Auth::user()`,
- Use the `auth()` helper and access `auth()->user()`,
- or inject the `Illuminate\Auth\Guard` instance in your class and access it as `$this->auth->user()` for example.

My take on this is that it's all on the table. Use whichever is quickest and easiest.

Often, I'll just use `Auth::user()` until I discover that my class is using `Auth::user()` in many places. Only then will I inject the `Guard` instance in my class's constructor and change the usage.

Laravel Elixir

We used Laravel Elixir to compile LESS files into CSS files and to combine multiple JavaScript files into single files. We also added a Gulp task to copy Bower assets into the `resources/assets` directory of the **l5beauty** project.

There's much more you can do with Elixir.

- You can compile **CoffeeScript** files into JavaScript.
- You can compile **Sass** files into CSS.
- You can compile EcmaScript into plain JavaScript.
- You can version your CSS and JavaScript files.
- You can combine CSS files into single files.
- You can even write your own Elixir extensions.

Tinker

We used `artisan tinker` a couple times during the development of the **l5beauty** project. First to set up the user, and then to send a test email.

Tinker uses the powerful **Psy Shell**. It's a REPL shell for PHP and the great thing about Tinker is that it bootstraps your application. It's like you're running a shell inside your application's environment.

Before Tinker this used to be my “debugging” process when I was testing something:

1. Create a closure in `app/Http/routes.php` to test something.
2. Point my browser to this route.
3. Look at the output, go back to #1 change things and refresh the browser.

But now I use Tinker all the time for quick tests and checks.

Use it! It'll save lots of time.

Artisan Commands

This book barely touched Artisan commands. We used a few of the built-in commands such as:

- `artisan migrate` - To run database migrations
- `artisan tinker` - To fire up Tinker
- `artisan db:seed` - To seed the database
- `artisan make:controller` - To create a skeleton for a Controller
- `artisan make:migration` - To create a skeleton for a database migration
- `artisan make:model` - To create an Eloquent Model class
- `artisan queue:work` - To process next item in the queue
- `artisan route:list` - To list application's routes
- `artisan schedule:run` - To run scheduled commands

But there are many Artisan commands we didn't need to use during creation of the **l5beauty** project. Execute `php artisan` without any options to see a list.

And we didn't create our own Artisan commands. Laravel makes the process quick and easy and now with Laravel 5.1 it's even easier using the new *Signature* format of artisan commands.

With Laravel 5.1 you can even have your Artisan commands output progress bars.

Check out the official Laravel 5.1 documentation at laravel.com for all the details.

Events

One area of Laravel 5.1 which the **l5beauty** project did not use at all are Events.

Events provide a simple observer pattern, allowing you to fire events within your application and not worry about who is listening for the events. You could have code like this sprinkled through your code.

```
event(new UserLoggedIn($user));  
// or  
event(new NewsletterWasMailed());  
// or even  
event(new ProductWasPurchased($orderDetails));
```

Other classes can be set up to listen for these events and perform some action when the event occurs.

You can even have Listeners queue themselves to be executed later!

It's a powerful system. The structure Laravel uses is:

1. Event classes are placed in the `app/Events` folder.
2. Listener classes are placed in the `app/Listeners` folder.
3. Events are mapped to Listeners in `app/Providers/EventServiceProvider.php`.

Really, that's the basics of Laravel 5.1 Events. Yes, you can have Listeners queued. And a really neat feature in Laravel 5.1 is that you can have Events broadcast to your application (the web page the user is currently on) over a web socket connection.

Find out more about Events in the Laravel 5.1 official documentation.

Form Requests

Form Requests are great. They allow the validation of form input to be separated from the Controller and wrapped in a nice little class.

We used Form Requests extensively throughout the **l5beauty** project.

Blade Template Engine

One of the issues I've always had with PHP “*templates*” is that PHP itself really has no need for a template engine. You can mix HTML and PHP together. Thus I've shied away from *Smarty* and other PHP template solutions because I didn't want the additional overhead of processing the templates into HTML code.

Not so with Blade.

See, Blade uses fast substitution, replacing it's Blade-syntax with PHP equivalents. The resulting file is pure PHP + HTML. Laravel even caches these files.

Every “view” in the **l5beauty** project uses a Blade template. Even so, there are some blade directives we didn't use. What follows is a fairly complete list of all Blade directives:

- `{{ $var }}` - Echo escaped variable content
- `{!! $var !!}` - Echo variable content
- `{-- Comment --}` - Comment
- `{{ $var or 'default' }}` - Echo content or default value
- `@{{ ignore }}` - Display the text AND curly braces
- `@extends('layout.name')` - Use a layout view
- `@yield('name')` - Output a section's content
- `@section('name')` - Start a section
- `@endsection` - End a section, don't yield it
- `@show` - End a section and yield it
- `@parent` - Bring in parent section's content
- `@include('view.name')` - Include another view
- `@include('view.name', ['key' => 'value'])` - Include a view, passing new variables
- `@if / @elseif / @else / @endif` - Just like PHP's control structure
- `@unless / @endunless` - The opposite of `@if`
- `@for / @endfor` - A for loop
- `@forelse / @empty / @endforelse` - A for loop with a special condition if loop is empty
- `@while / @endwhile` - A while loop
- `@lang('message.name')` - Output language specific string
- `@choice('message.name', 1)` - Output language specific string pluralized correctly
- `@inject('name', 'class')` - Create an instance of `class` (or pull it from IoC) as variable `$name`

I especially like the last one, `@inject()`. It's new in Laravel 5.1.

Flysystem

Laravel 5.1's filesystem is built upon the Flysystem PHP package by Frank de Jonge. We used it in the **l5beauty** project when we developed the Upload Manager. First we used the local file system, then (and optionally) we used the Amazon S3 driver to store uploads in the cloud.

Besides **local** and **Amazon S3** there's a couple other drivers you can use.

1. **The FTP Driver** - Using this driver you can connect to any standard FTP server.
2. **The Rackspace Driver** - Like the Amazon S3 driver, this driver allows you to manage files in the cloud.

Queues

We used Laravel 5.1 queuing system in the **l5beauty** project to queue emails from the contact form. I'll only say two things about queues.

1. Learn how to use them.
2. Use them every time it's appropriate.

And with Laravel 5.1's Database Queue Driver, there's no reason not to use them when needed.

Blog Features to Add

Although the **l5beauty** project is a complete, working blog. There are a few features I suggest you add on your own.

A Newsletter System

You could add a Newsletter Sign-up Form, and even a Newsletter system. For this I suggest you use Mail Chimp.

I've created several such systems for projects in the past. It's easy to do and would be an excellent addition to the **L5 Beauty Blog**.

Dropzone Upload

With the Upload Manager of the blog I focused on the management of the files and only implemented a basic file upload.

It would be a nice feature to integrate **DropzoneJS** and allow drag-and-drop file uploads.

Comments

Some people don't like Disqus. If you don't, why not create your own comment system?

Page Caching

How about caching the pages of the blog? This would be a great feature to add.

My approach would be to implement a page cache as middleware. Specify which routes you want to have cached, and then implement the middleware's handle method with something like:

Pseudo Code for a Page Caching Middleware

```
// in the handle method
public function handle($request, Closure $next)
{
    $cacheKey = $this->getCacheKey($request);
    if (Cache::has($cacheKey)) {
        return Cache::get($cacheKey);
    }

    $response = $next($request);

    Cache::put($cacheKey, $response, 30);

    return $response;
}
```

Final Recap and Thank You

That's all. The song is over (so to speak).

I sincerely hope you've found this book helpful. That's been my #1 goal in writing this, to help others come into the world of Laravel and discover what a great place it is.

Stop by my blog at [LaravelCoding.com](http://laravelcoding.com)³⁶ and see what I'm up to. I'm a very inconsistent blogger, but occasionally I do update things.

Thank you again for purchasing this book.

— Chuck Heintzelman, July 5, 2015

³⁶<http://laravelcoding.com>