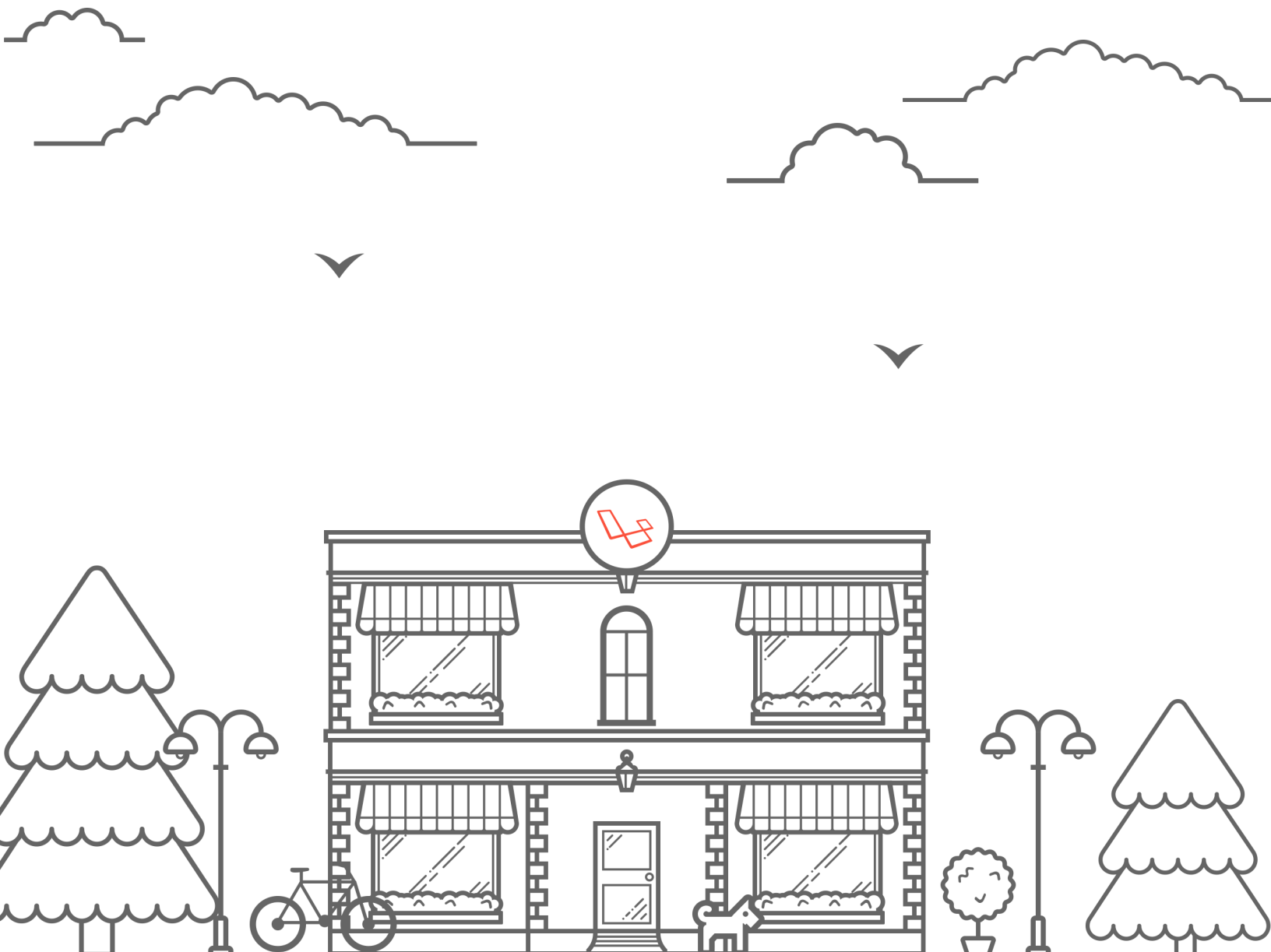
Learning Laravel 5

Building Practical Applications

by Nathan Wu



Learning Laravel 5

Building Practical Applications

Nathan Wu

Contents

Book Information

Book Description

Learning Laravel 5: Building Practical Applications is the easiest way to learn web development using Laravel. Throughout 5 chapters, instructor Nathan Wu will teach you how to build many real-world applications from scratch. This bestseller is also completely about you. It has been structured very carefully, teaching you all you need to know from installing your Laravel 5 app to deploying it to a live server.

When you have completed this book you will have created a dynamic website and have a good knowledge to become a good web developer.

We first start with the basics. You will learn some main concepts and create a simple website. After that we progress to building more advanced web applications.

Learn by doing!

If youâ€™re looking for a genuinely effective book that helps you to build your next amazing applications, this is the number one book for you.

Requirements

The projects in this book are intended to help people who have grasped the basics of PHP and HTML to move forward, developing more complex projects, using Laravel advanced techniques. The fundamentals of the PHP are not covered, you will need to:

- Have a basic knowledge of PHP, HTML, CSS.
- Love Laravel, like we do.

What You Will Get

- Lifetime access to the online book. (Read 70% of the book for FREE!)
- Digital books: PDF, MOBI, EPUB (Premium Only)
- Full source code (Premium Only)
- Access new chapters of the book while it's being written (Premium Only)
- A community of 10000+ students learning together.
- Amazing bundles and freebies to help you become a successful developer.
- iPhone, iPad and Android Accessibility.

Book Structure

Note: This book is still under active development, that means some chapters and its content may change. The book also may have some errors and bugs. For any feedback, please send us an email. Thank you.

Chapter 1 - Installing Laravel

There are many ways to install Laravel. In this chapter, you will learn how to setup Laravel Homestead (a Vagrant-based virtual machine), and run your Laravel projects on it.

Chapter 2 - Building Our First Website

This book is meant to help you build the skills to create web applications as quickly and reliably as possible. We present you with four projects in various states of completion to explain and practice the various concepts being presented. Our first app, which is a simple website, will walk you through the structure of a Laravel app, and show some main concepts of Laravel. You will also create a good template for our next applications.

Chapter 3 - Building A Support Ticket System

After having a good template, we will start building a support ticket system to learn some Laravel features, such as Eloquent ORM, Eloquent Relationships, Migrations, Requests, Laravel Collective, sending emails, etc.

While the project design is simple, it provides an excellent way to explore Laravel. You will also know how to construct your app structure the right way.

Chapter 4 - Building A Blog Application

Throughout the projects in this book up to this point, we've learned many things. It's time to use our skills to build a complete blog system. You will learn to make an admin control panel to create and manage your posts, users, roles, permissions, etc.

Chapter 5 - Deploying Our Laravel Applications

Finally, we learn how to create our own web server and deploy our Laravel app to it. Launching your first Laravel 5 application is that easy!

Feedback

Feedback from our readers is always welcome. Let us know what you liked or may have disliked.

Simply send an email to support@learninglaravel.net.

We're always here.

Translation

We're also looking for translators who can help to translate our book to other languages.

Feel free to contact us at support@learninglaravel.net.

Here is a list of our current translators:

List of Translators

Book Status, Changelog and Contributors

You can always check the book status, changelog and view the list of contributors at:

Book Status

Changelog

Contributors

Translators

Changelog

Current Version

Latest version the book:

- Version: 1.5
- Status: Complete. The book now supports Laravel 5.2.
- Updated: May 15th, 2016

Learning Laravel 5

Chapter 1: Installing Laravel

There are many ways to install Laravel. We can install Laravel directly on our main machine, or we can use all-in-one server stacks such as MAMP, XAMPP, etc. We have a huge selection of ways to choose.

In this book, I will show you the most popular one: **Laravel Homestead**.

Introducing CLI (Command Line Interface)

> If you haven't heard about CLI, Terminal or Git, this section is for you. If you know how to use the CLI already, you may skip this section.

Working with Laravel requires a lot of interactions with the CLI, thus you will need to know how to use it.

CLI for MAC OSX

Luckily, on Mac, you can find a good CLI called **Terminal** at **/Applications/Utilities**.

Most of what you do in the **Terminal** is enter specific text strings, then press **Return** to execute them.

Alternatively, you can use iTerms 2.

CLI for Windows

Unfortunately, the default CLI for Windows (cmd.exe) is not good, you may need another one.

The most popular one called **Git Bash**. You can download and install it here:

http://msysgit.github.io

Most of what you do in **Git Bash** is enter specific text strings, then press **Enter** to execute them.

CLI for Linux

On Linux, the CLI is called **Terminal** or **Konsole**. If you know how to install and use Linux, I guess you've known how to use the CLI already.

Installing Laravel Using Homestead

What is Homestead?

Nowadays, many developers are using a virtual machine (VM) to develop dynamic websites and applications. You can run a web server, a database server and all your scripts on that virtual machine. You can create many VM instances and work on various projects. If you don't want any VM anymore, you can safely delete it without affecting anything. You can even re-create the VM in minutes!

We call this: "Virtualization."

There are many options for virtualization, but the most popular one is VirtualBox from Oracle. VirtualBox will help us to install and run many virtual machines as we like on our Windows, Mac, Linux or Solaris operating systems. After that, we will use a tool called Vagrant to manage and configure our virtual development environments.

In 2014, Taylor Otwell - the creator of Laravel - has introduced Homestead.

Homestead is a Vagrant based Virtual Machine (VM) and it is based on Ubuntu. It includes everything we need to start developing Laravel applications. That means, when we install Homestead, we have a virtual server that has PHP, Nginx, databases and other packages. We can start creating our Laravel application right away.

Here is a list of included software:

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

How to install Homestead?

There are 2 methods to install Homestead.

Method 1: Using Composer.

Method 2 (new): Using Git Clone.

In May 2015, the Laravel official documentation has been updated. The recommended way to install Homestead is using Git.

It's better, faster.

Definitely.

However, you can still choose the old way if you like.

Install Homestead Using Method 1 (required Composer)

> **Note**: this is the old method, if you don't install Homestead yet, you should use method 2!

There are four steps to install Homestead.

Step 1: Install VirtualBox

Step 2: Install Vagrant

Step 3: Install Composer

Step 4: Install Homestead

Let's start by installing VirtualBox and Vagrant first.

Step 1 - Installing VirtualBox

First, we need to go to:

https://www.virtualbox.org/wiki/Downloads

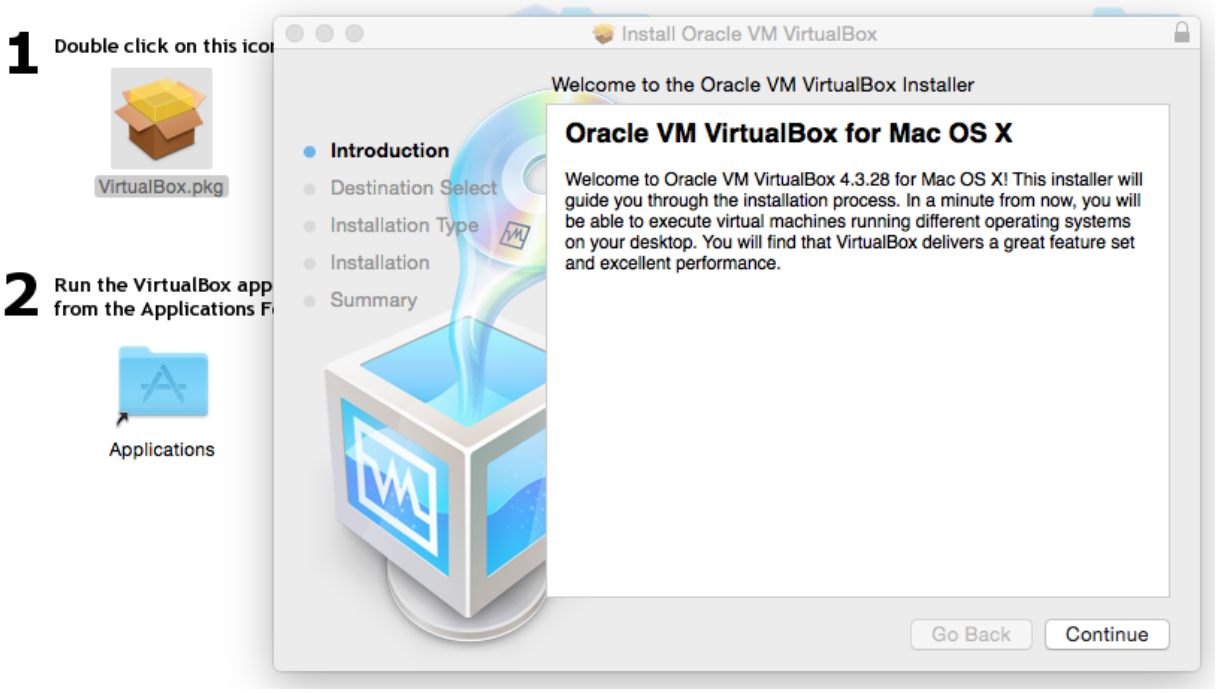
Choose a VirtualBox for your platform and install it.

Make sure that you download the correct version for your operating system.

The **stable release is version 4.3.28**. You can use a newer version if you want, but if you have any problems, try this version.

If you're using Windows, double click the .exe setup file to install VirtualBox.

If you're using Mac, simply open the VirtualBox .dmg file and click on the .pkg file to install.



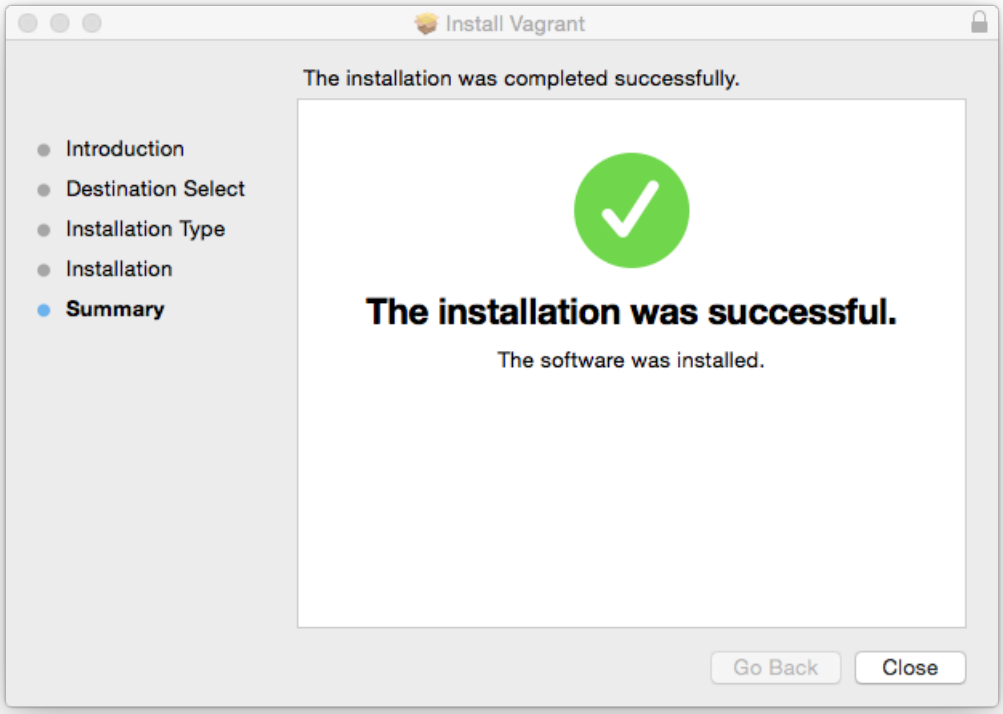
Installing VirtualBox

Step 2 - Installing Vagrant

The next step is to install Vagrant. Please go to:

http://www.vagrantup.com/downloads.html

If you're using Mac, download the .dmg file -> Open the downloaded file -> Click on the Vagrant.pkg file to install it.



Installing Vagrant

If you still don't know how to install, there is an official guide on Vagrant website:

http://docs.vagrantup.com/v2/installation

Step 3 - Installing Composer

What is Composer? Composer is a cross-platform dependency manager for PHP libraries. We use Composer to install, remove and update PHP packages. You will learn more about it in this book. For now, we need to install Composer and use it to install Homestead.

For Mac Users

You can find the official installation guide here:

https://getcomposer.org/doc/00-intro.md#installation-linux-unix-osx

To install Composer, we have to use Terminal app, or any other terminal emulator applications (such as iTerms). On Mac, you can find it at **Applications-> Utilities-> Terminal**.

Most of what you do in the Terminal app is enter specific text strings, then press Return to execute them.

Let's install Composer by executing this command:

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

If you see this error:

```
The detect_unicode setting must be disabled.
```

Use this command instead:

```
curl -s getcomposer.org/installer | php -d detect_unicode=Off
```

The installer will download composer.phar to your working directory. To access composer easily from anywhere on our system, we need to move the composer.phar file to /usr/local/bin directory. Execute this command:

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

If it asks for your password, enter your Mac Admin password.

> Note: In OSX Yosemite, there is no /usr directory by default. If you see this error:
>
> ```
> /usr/local/bin/composer: No such file or directory
> ```
>
> then you must create /usr/local/bin manually by entering this command first:
>
> ```
> sudo mkdir /usr/local/bin
> ```

We can test if we have installed Composer by typing **composer** into Terminal.
We should see something like this:

```
▲ ~ composer
   ______
  / ____/___  ____ ___  ____  ____  ________  _____
 / /   / __ \/ __ `__ \/ __ \/ __ \/ ___/ _ \/ ___/
/ /___/ /_/ / / / / / / /_/ / /_/ (__  )  __/ /
\____/\____/_/ /_/ /_/ .___/\____/____/\___/_/
                    /_/
Composer version 1.0-dev (e64470c987fdd6bff03b85eed823eb4b865a4152) 2015-05-28 14:52:12

Usage:
 command [options] [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
 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
 --profile             Display timing and memory usage information
 --working-dir (-d)    If specified, use the given directory as working directory.

Available commands:
 about            Short information about Composer
 archive          Create an archive of this composer package
 browse           Opens the package's repository URL or homepage in your browser.
 clear-cache      Clears composer's internal package cache.
 clearcache       Clears composer's internal package cache.
 config           Set config options
 create-project   Create new project from a package into given directory.
 depends          Shows which packages depend on the given package
 diagnose         Diagnoses the system to identify common errors.
 dump-autoload    Dumps the autoloader
 dumpautoload     Dumps the autoloader
 global           Allows running commands in the global composer dir ($COMPOSER_HOME).
 help             Displays help for a command
 home             Opens the package's repository URL or homepage in your browser.
 info             Show information about packages
 init             Creates a basic composer.json file in current directory.
 install          Installs the project dependencies from the composer.lock file if present, or fal
```

Chap1 Pic 1

Good job! You now have Composer on your system!

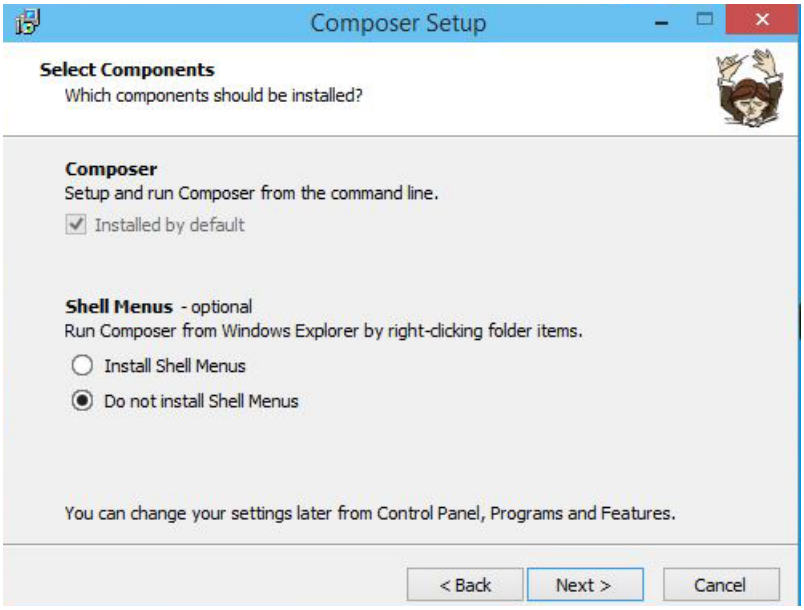
For Windows User

You can find the official installation guide here:

https://getcomposer.org/doc/00-intro.md#installation-windows

The easiest and fastest way to install Composer on Windows is download and run:

Composer-Setup.exe



Install Composer on Windows

Follow the simple instructions, you should have Composer installed.

We can now use Composer to install Homestead by executing some commands.

But... how to execute commands on Windows?

To execute commands on Windows, we will need to use a Bash emulation (behaves just like the git command in Linux, Unix or the Terminal app in Mac). The most popular one is Git Bash.

You can download and install it here:

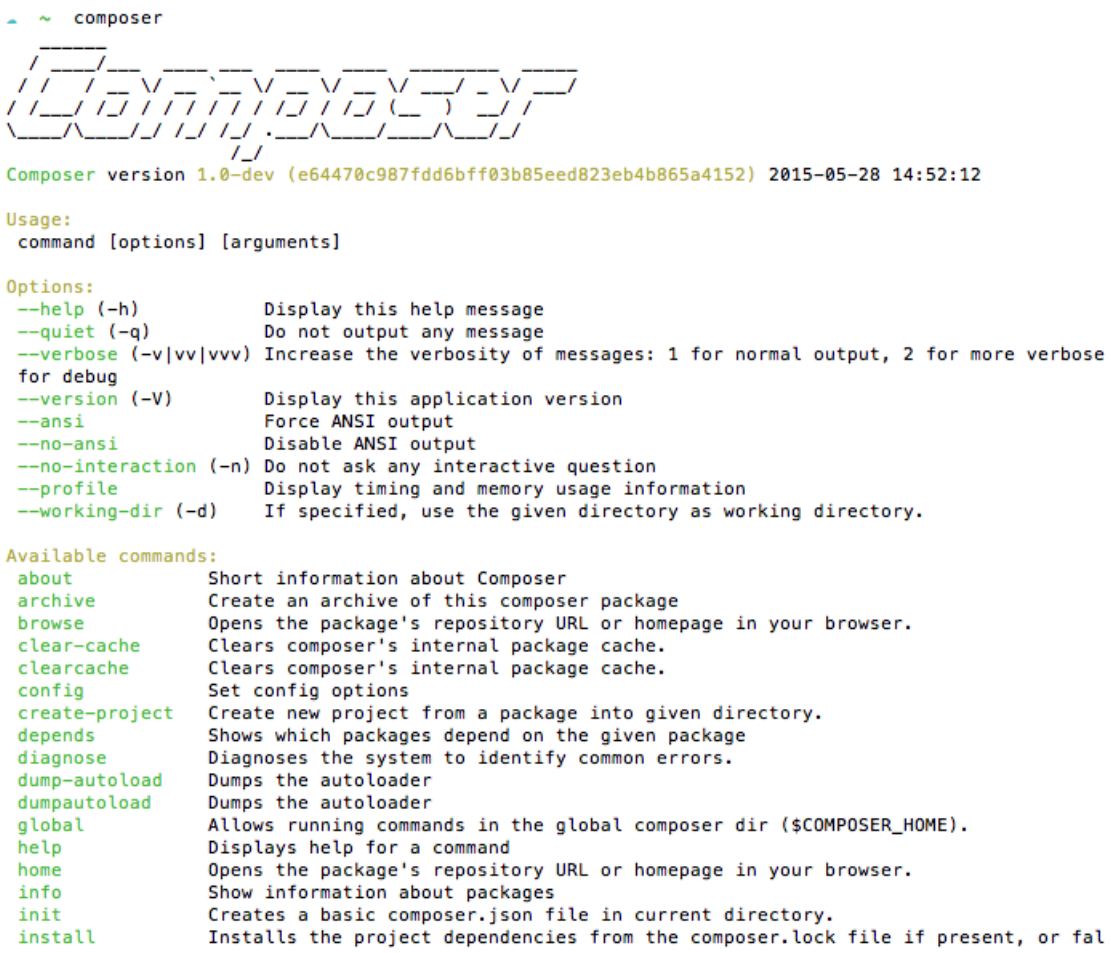
http://msysgit.github.io

When it asks something, you should accept the default options.

Most of what you do in Git Bash is enter specific text strings, then press **Enter** to execute them.

After that, open Git Bash and type **composer**, hit **Enter**!

We should see something like this:



Chap1 Pic 1

Good job! You now have Composer on your system!

Step 4 - Install Homestead (Using Composer)

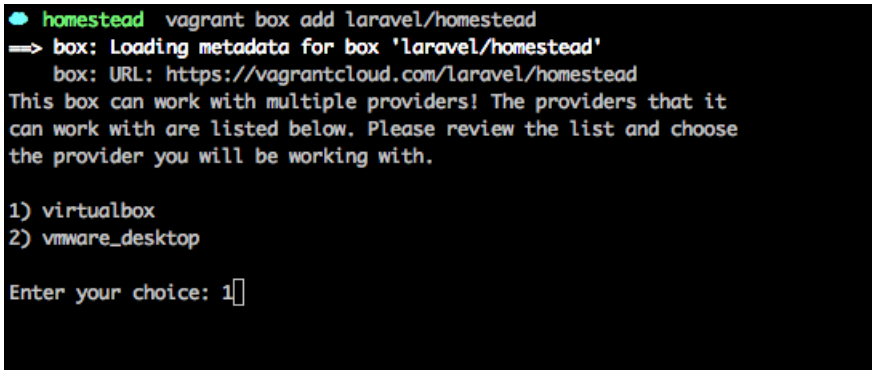
You can find Homestead's official documentation here:

http://laravel.com/docs/4.2/homestead

First, we need to add the Vagrant Box, enter the following command to your terminal (Git Bash):

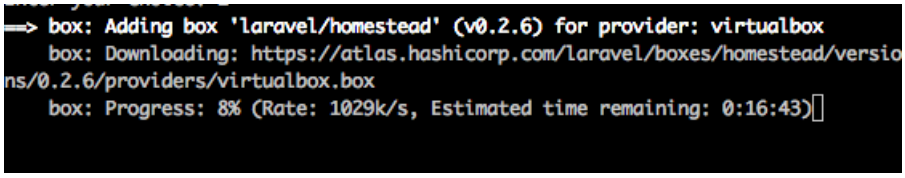
```
vagrant box add laravel/homestead
```

*Note: If you have multiple VM apps (such as VMware of Parallels), make sure to choose VirtualBox:



Make sure to choose Virtual Box

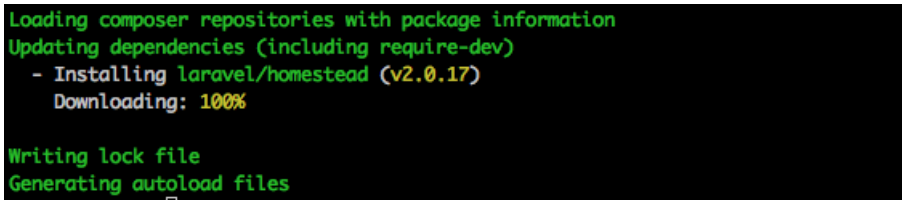
It may take a few minutes to download Homestead to your system.



Downloading Homestead

You're now ready to install **Homestead** using **Composer**, enter this command:

```
composer global require "laravel/homestead=~2.0"
```



Downloading Homestead 2.0

The next step is initialize Homestead to create **Homestead.yaml** configuration file

```
homestead init
```

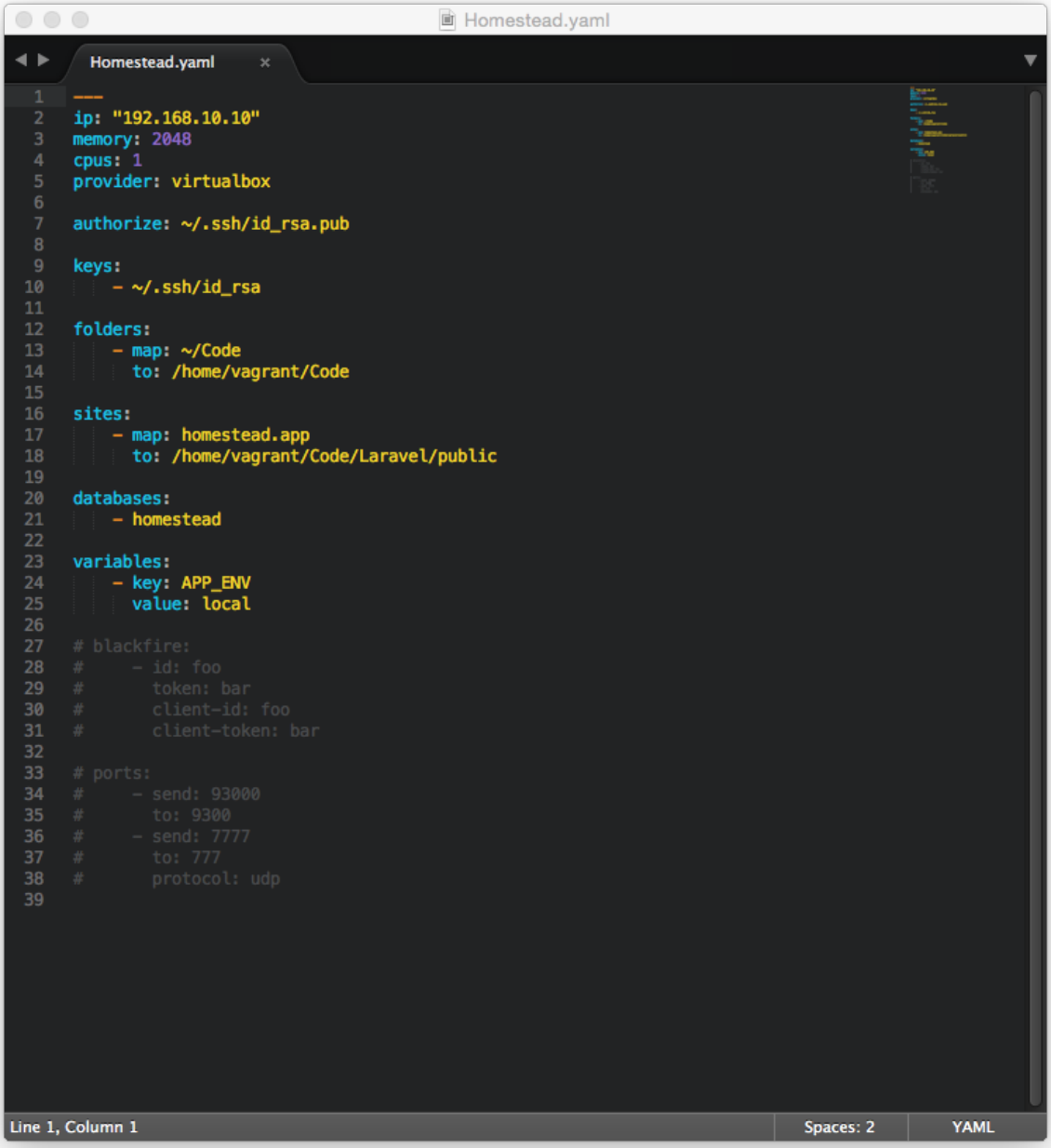
> Note: If you're seeing this error: "**command not found: homestead**". You need to configure your **$PATH**. Please check out this guide to understand what **$PATH** is and how to edit it:
>
> http://www.cyberciti.biz/faq/appleosx-bash-unix-change-set-path-environment-variable
>
> Basically, you will need to add this line to your $PATH variable by editing **.bash_profile** file or **.zshrc file** (if you're using zsh)

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

If you can run the **homestead init** command successfully, you may edit the **Homestead.yaml** file by using **homestead edit** command:

```
homestead edit
```

Your default text editor will be launched. You will see:



Homestead configuration file

Alternatively, you can find the **Homestead.yaml** in the ~/**.homestead** directory (the directory is hidden by default, make sure that you can see hidden files if you want to find it). Open the **Homestead.yaml** file with a text editor to edit it.

Install Homestead Using Method 2 (requires Git)

There are three steps to install Homestead using this method.

Step 1: Install VirtualBox

Step 2: Install Vagrant

Step 3: Install Homestead

Let's start by installing VirtualBox and Vagrant first.

Step 1 - Installing VirtualBox

First, we need to go to:

https://www.virtualbox.org/wiki/Downloads

Choose a VirtualBox for your platform and install it.

Make sure that you download the correct version for your operating system.

The **stable release is version** 4.3.28. You can use a newer version if you want, but if you have any problems, try this version.

If you're using Windows, double click the .exe setup file to install VirtualBox.

If you're using Mac, simply open the VirtualBox .dmg file and click on the .pkg file to install.

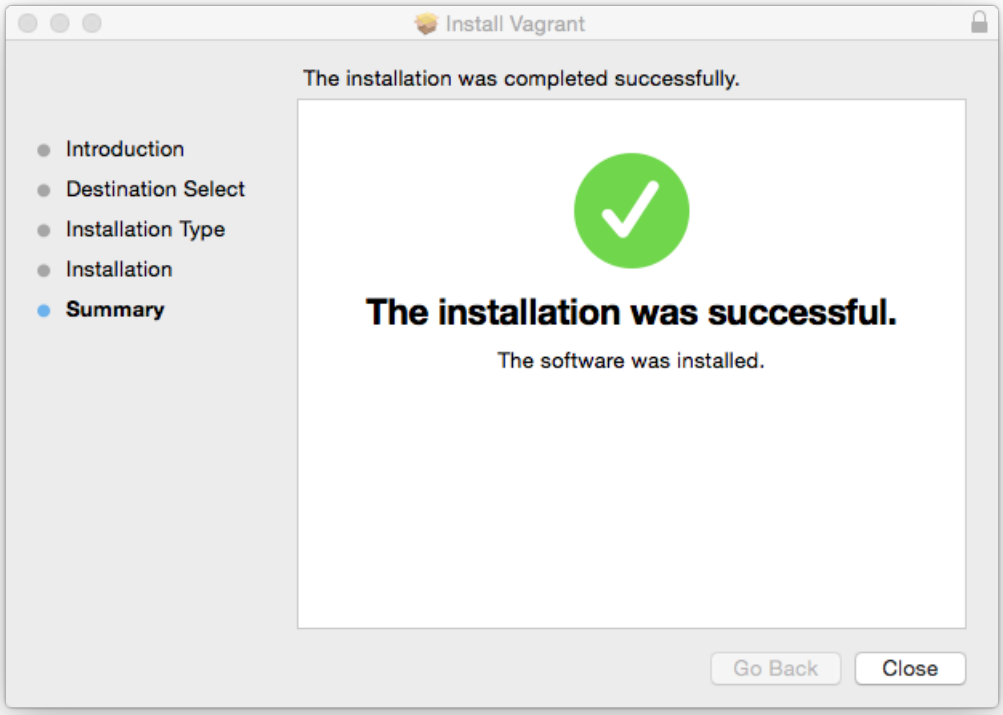
Installing VirtualBox

Step 2 - Installing Vagrant

The next step is to install Vagrant. Please go to:

http://www.vagrantup.com/downloads.html

If you're using Mac, download the .dmg file -> Open the downloaded file -> Click on the Vagrant.pkg file to install it.



Installing Vagrant

If you still don't know how to install, there is an official guide on Vagrant website:

http://docs.vagrantup.com/v2/installation

Step 3 - Install Homestead (Using Git Clone)

You can install Homestead just by **cloning the Homestead Repository**.

You will need to install **Git** first if you don't have it on your system.

> **Note**: if you don't know how to run a command, please read **Introducing CLI (Command Line Interface)** section.

Install Git on Mac

The easiest way is to install the **Xcode Command Line Tools**. You can do this by simply running this command:

```
xcode-select --install
```

Click **Install** to download **Command Line Tools** package.

Alternatively, you can also find the **OSX Git installer** at this website:

http://git-scm.com/download/mac

Install Git on Windows

You can download **GitHub for Windows** to install Git:

https://windows.github.com

Install Git on Linux/Unix

You can install Git by running this command:

```
$ sudo yum install git
```

If youâ€™re on a Debian-based distribution, use this:

```
$ sudo apt-get install git
```

For more information and other methods, you can see this guide:

https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

When you have Git installed. Enter the following command to your Terminal (or Git Bash):

```
git clone https://github.com/laravel/homestead.git Homestead
```

Cloning Homestead

Once downloaded, go to the **Homestead directory** by using **cd** command:

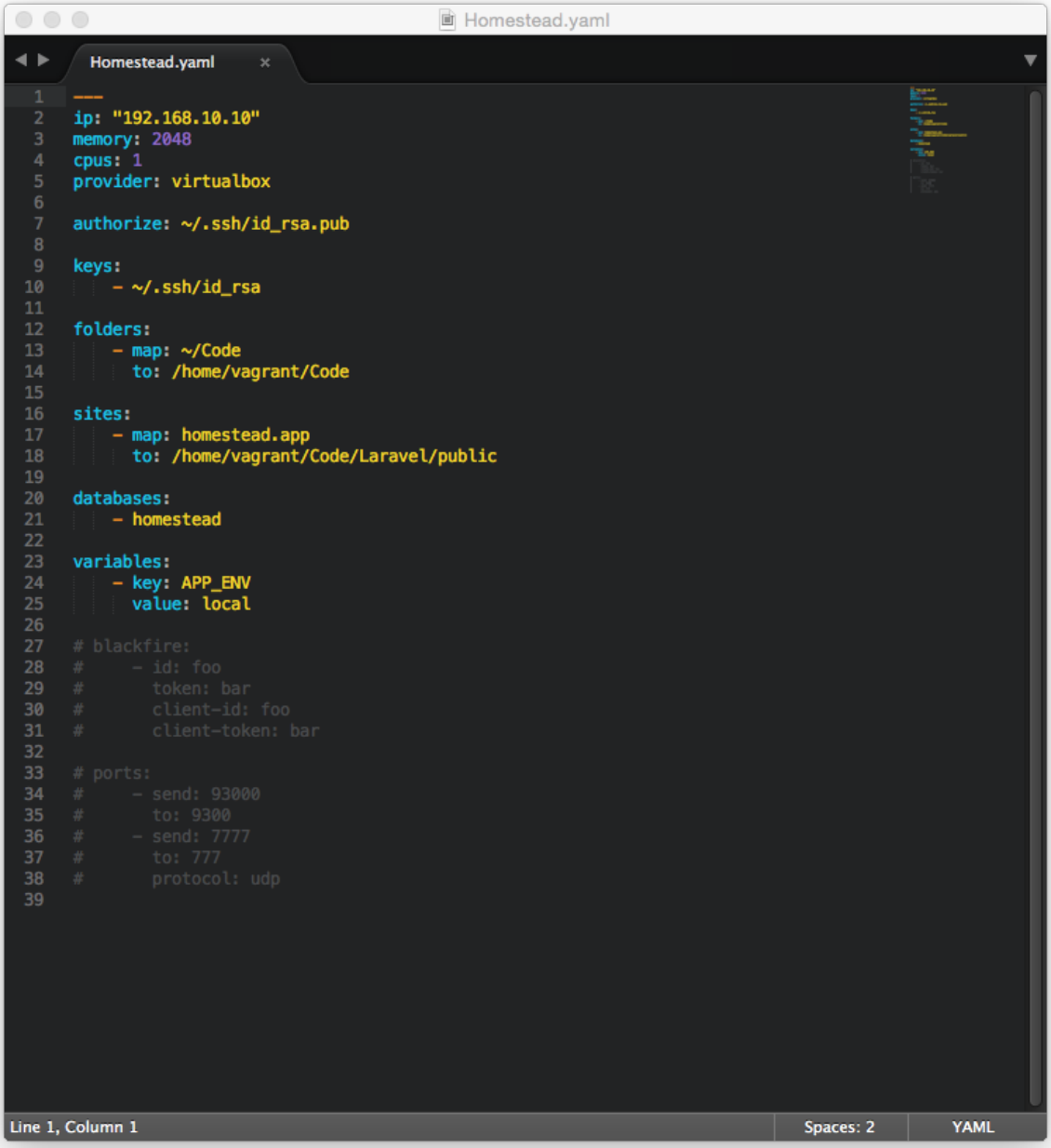
```
cd Homestead
```

Run this command to create **Homestead.yaml** file

```
bash init.sh
```

The **Homestead.yaml** file will be placed in your ~/.**homestead** directory. Open it with a text editor to edit it.

> Please note that the ~/.**homestead** directory is hidden by default, make sure that you can see hidden files.



Homestead configuration file

If you know how to use **VI** or **VIM**, use this command to edit the file:

```
vi ~/.homestead/Homestead.yaml
```

> **Note**: Your system path may be different. Try to find Homestead.yaml.

Configure Homestead

The structure of the **Homestead.yaml** is simple. There are 7 sections. Let's see what they do.

First section - Configure VM

```
ip: "192.168.10.10"
memory: 2048
cpus: 1
provider: virtualbox
```

As you can see, we can configure the IP address, memory, cpus and provider of our VM. This section is not important, so we can just leave it as it is.

Second and third section - Configure SSH

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

keys:
    - ~/.ssh/id_rsa
```

Basically, we need to generate an SSH key for Homestead to authenticate the user and connect to the VM. If you're working with Git, you may have an SSH key already. If you don't have it, simply run this command to generate it:

```
ssh-keygen -t rsa -C "you@homestead"
```

The command will generate an SSH key for you and put it in the ~/**.ssh** directory automatically, you don't need to do anything else.

Fourth section - Configure shared folder

We use **folders** section to specify the directory that we want to share with our Homestead environment. If we add, edit or change any files on our local machine, the files will be updated automatically on our Homestead VM.

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

We can see that the ~/**Code** directory has been put there by default. This is where we put all the files, scripts on our local machine. Feel free to change the link if you want to put your codes elsewhere.

The **/home/vagrant/Code** is a path to the Code directory on our VM. Usually, we don't need to change it.

Fifth section - Map a domain

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

This section allows us to map a domain to a folder on our VM. For example, we can map **homestead.app** to the public folder of our Laravel project, and then we can easily access our Laravel app via this address: "http://homestead.app".

Remember that, when we add any domain, we must edit the **hosts** file on our local machine to redirect requests to our Homestead environment.

On Linux or Mac, you can find the **hosts** file at **etc/hosts** or **/private/etc/hosts**. You can edit the hosts file using this command:

```
sudo open /etc/hosts
```

If you know how to use **VI** or **VIM**, use this command to edit the file:

```
sudo vim /etc/hosts
```

On Windows, you can find the **hosts** file at **C:\Windows\System32\drivers\etc\hosts**.

After opening the file, you need to add this line at the end of the file:

```
192.168.10.10  homestead.app
```

Done! When we launch Homestead, we can access the site via this address.

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

> Please note that we can change the address (homestead.app) to whatever we like.

All sites will be accessible by HTTP via port 8000 and HTTPS via port 44300 by default.

Sixth section - Configure database

```
databases:
    - homestead
```

This is the database name of our VM. As usual, we just leave it as it is.

Seventh section - Add custom variables

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

If we want to add some custom variables to our VM, we can add them here. It's not important, so let's move to the next fun part.

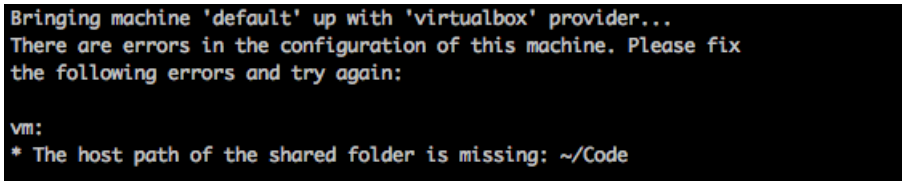
Launching Homestead

Once we have edited **Homestead.yaml** file, **cd** to the **Homestead** directory, run this command to boot our virtual machine:

```
vagrant up
```

It may take a few minutes...

If you see this error:



Homestead error

It means that you don't have **Code** directory in your main machine. You can create one, or change the link to any folder that you like.

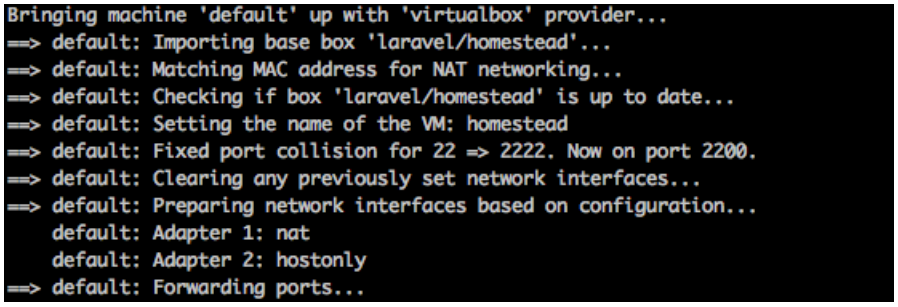
Executing this command to create **Code** folder:

```
sudo mkdir ~/Code
```

> **Note**: if you have any errors when creating Laravel, try to set right permissions for the Code folder by running:

```
chmod -R 0777 ~/Code
```

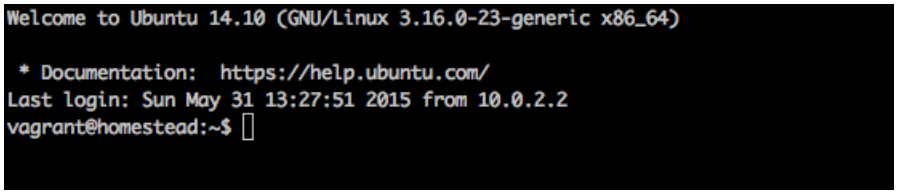
If everything is going fine, we should see:



Booting Homestead

Now we can access our VM using:

```
vagrant ssh
```



SSH Homestead

To make sure that everything is ok, run **ls** command:

Run ls command

If you can see the **Code** directory there, you have Homestead installed correctly!

Excellent! Let's start installing Laravel!

Installing Laravel

When you have installed Homestead, create a new Laravel app is so easy!

As I've mentioned before, the **Code** directory is where we will put our Laravel apps. Let's go there!

```
cd Code
```

You should notice that the directory is empty. There are two methods to install Laravel.

Install Laravel Via Laravel Installer

This method is recommended. It's newer and faster. You should use this method to create your Laravel application.

First, we need to use **Composer** to download the **Laravel installer**.

```
composer global require "laravel/installer=~1.1"
```

This command is used to download **Laravel 5.0**, if you want to use the latest version, use:

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

> **Note**: It is recommended to use Laravel 5.0 to learn the basics of Laravel Framework. You can upgrade to a newer version later. However, feel free to use the latest version if you want because the book will be updated frequently to support newer versions.

Use Composer to install Laravel installer

Once downloaded, you can create a new Laravel project by using this command:

```
laravel new nameOfYourSite
```

You're free to change the **nameOfYourSite** to whatever you like, but remember to edit the **sites section** of **Homestead.yaml** to match your site's name.

For instance, in **Homestead.yaml**, we specify the name of our app is **Laravel**

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

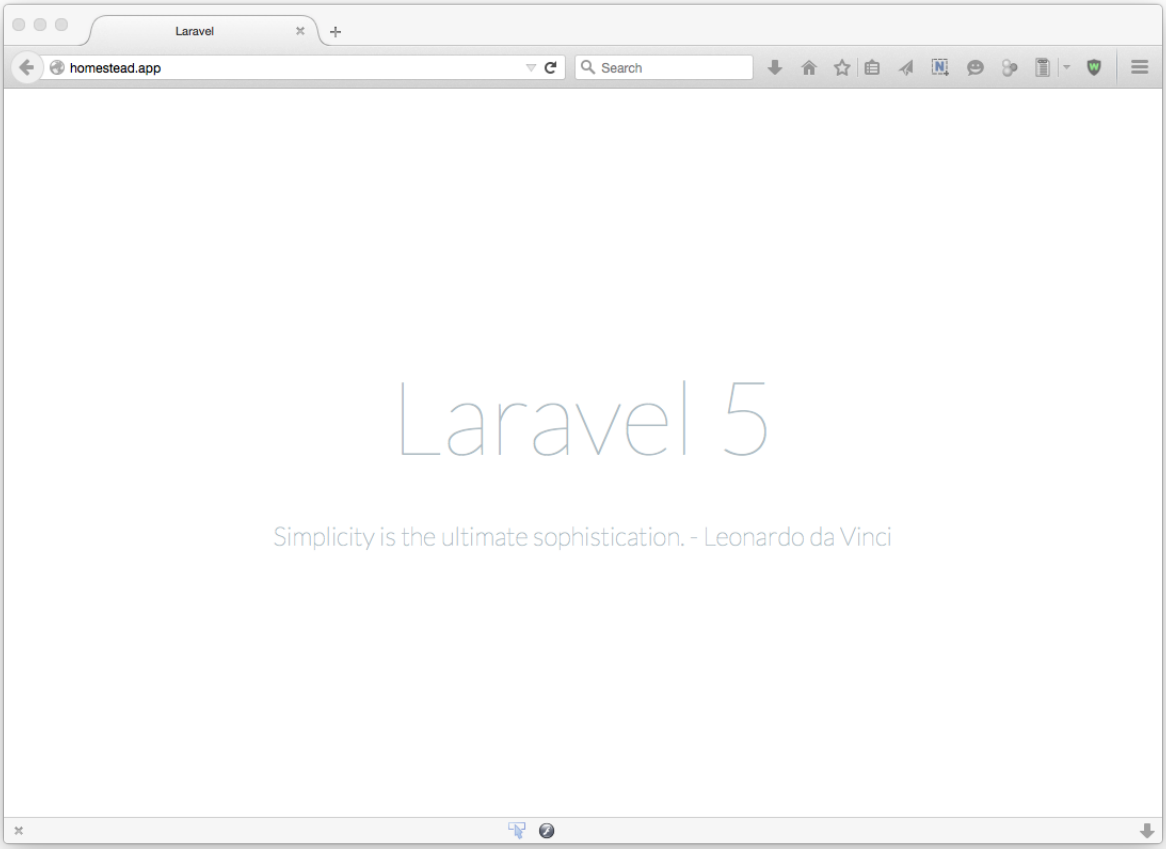
We will need to run this command to create **Laravel** site

```
laravel new Laravel
```

You should see this:

Create a new Laravel app

Open your web browser, go to: **http://homestead.app**



Laravel is ready!

Congratulations! You've installed Laravel! It's time to create something amazing!

Install Laravel Via Composer Create-Project

If you don't like to use Laravel Installer, or you have any problems with it, feel free to use **Composer Create-Project** to create a new Laravel app.

```
composer create-project laravel/laravel nameOfYourSite "~5.0.0"
```

This command is used to download **Laravel 5.0**, if you want to use the latest version, use:

```
composer create-project laravel/laravel nameOfYourSite
```

You're free to change the **nameOfYourSite** to whatever you like, but remember to edit the **sites section** of **Homestead.yaml** to match your site's name.

For instance, in **Homestead.yaml**, we specify the name of our app is **Laravel**

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

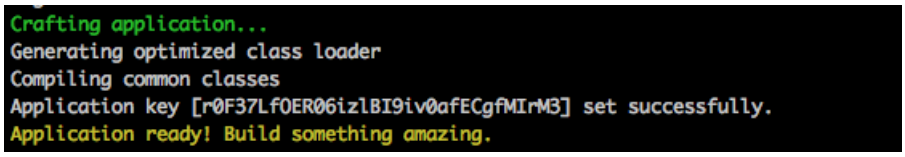
We will need to run this command to create **Laravel** site

```
composer create-project laravel/laravel Laravel
```

Alternatively, we can create a new **Laravel** folder, **cd** to it, and create our Laravel app there:

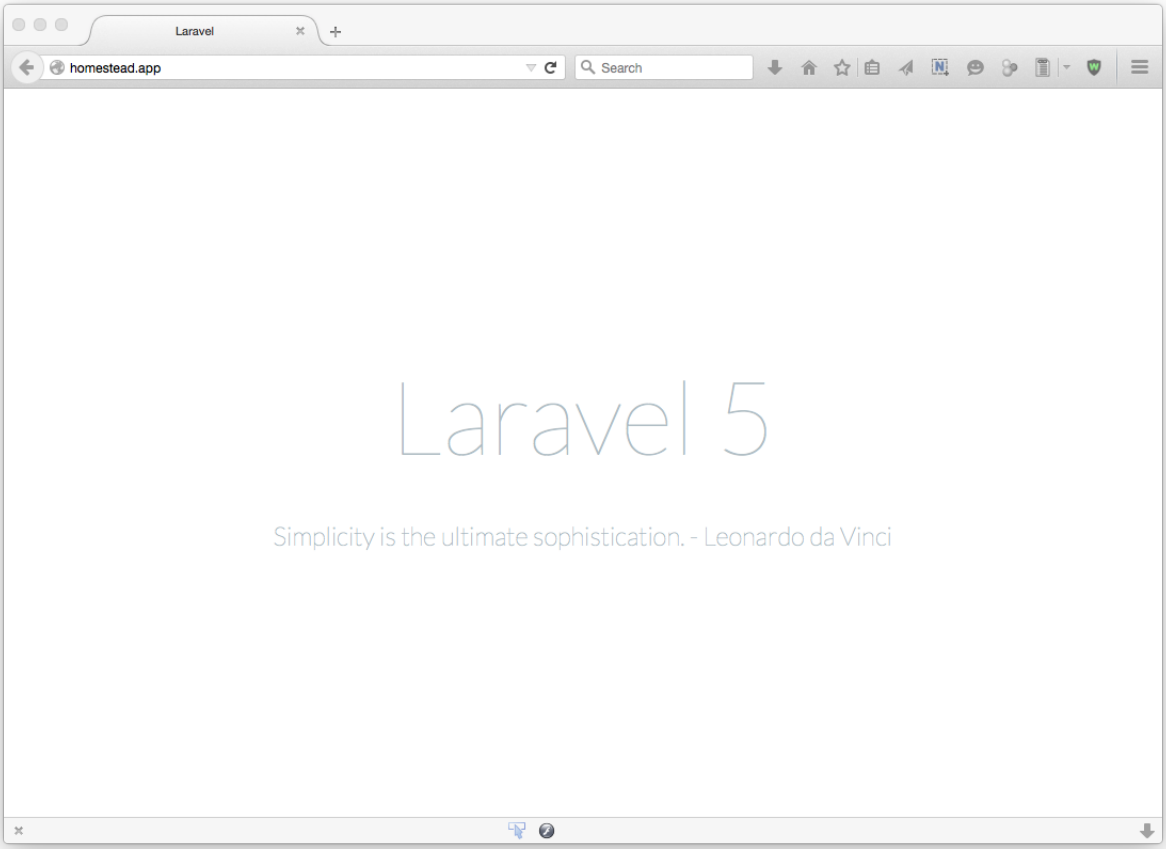
```
mkdir Laravel
cd Laravel
composer create-project laravel/laravel --prefer-dist
```

You should see this:



Create a new Laravel app

Open your web browser, go to: **http://homestead.app**



Laravel is ready!

Note: if you cannot access the site, try to add the port into the URL: http://homestead.app:8000.

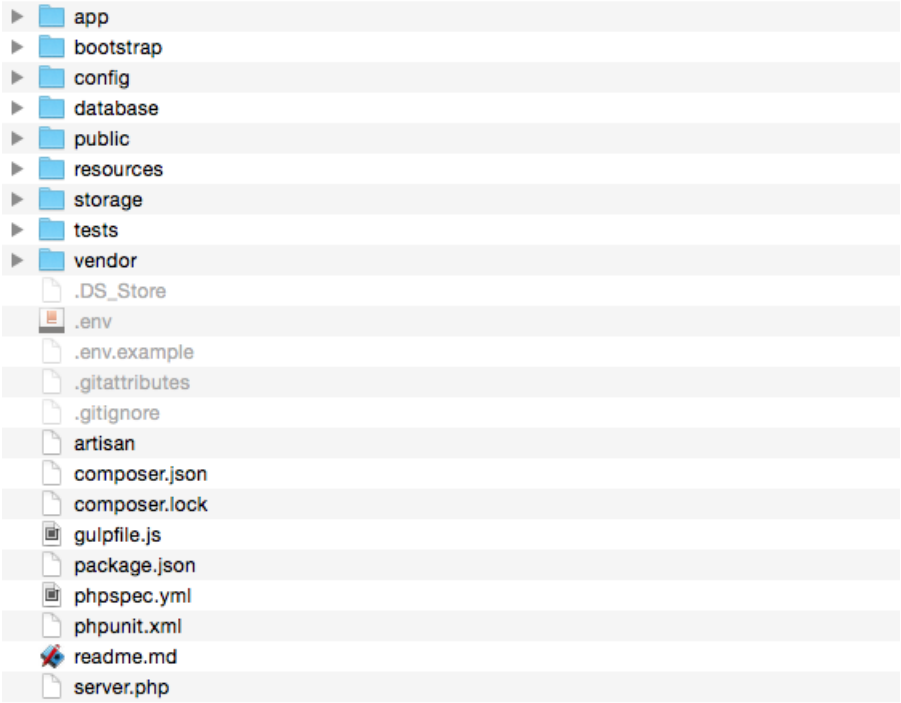
Congratulations! You've installed Laravel! It's time to create something amazing!

Chapter 2: Building Our First Website

Now that we know how to install Laravel, let's start working our way into our first Laravel website. In this chapter, you will learn about Laravel structure, routes, Controllers, Blade templates, Artisan commands, Elixir and many basic features that will come handy when building Laravel applications.

Exploring Laravel structure

I assume that you've installed Laravel at ~/**Code/Laravel**. Let's go there and open the **Laravel** directory.



Laravel structure

To build applications using Laravel, you will need to understand truly Laravel.

Laravel follows **MVC (Model View Controller)** pattern, so if you've already known about MVC, everything will be simple. Don't worry if you don't know what MVC is, you will get to know soon.

As you may have seen, every time you visit a Laravel app, you'll see **nine folders**.

1. app

2. bootstrap
3. config
4. database
5. public
6. resources
7. storage
8. tests
9. vendor

I'm not going to tell you everything about them right now because I know that it's boring.

Trust me.

But we have to take a quick look at them to know what they are, anyway.

App

This directory holds all our application's logic. We will put our controllers, services, filters, commands and many other custom classes here.

Bootstrap

This folder has some files that are used to bootstrap Laravel. The cache folder is also placed here.

Config

When we want to configure our application, check out this folder. We will configure database, mail, session, etc. here.

Database

As the name implies, this folder contains our database migrations and database seeders.

Public

The public folder contains the application's images, CSS, JS and other public files.

Resources

We should put our views (.blade.php files), raw files and other localization files here.

Storage

Laravel will use this folder to store sessions, caches, templates, logs, etc.

Tests

This folder contains the test files, such as PHPUnit files.

Vendor

Composer dependencies (such as Symfony classes, PHPUnit classes, etc.) are placed here.

To understand more about Laravel structure, you can read the official documentation here:

http://laravel.com/docs/master/structure

Understand routes.php

One of the most important files of Laravel is **routes.php**. This file can be found at **Laravel/app/Http/**.

We use this file for "**routing**".

What does it mean?

Routing means that you will tell Laravel to get URL requests and assign them to specific actions that you want. For instance, when someone visits **homestead.app**, which is the home page of our current application, Laravel will "think": "Oh, this guy is going to the home page, I need to display the welcome view and then give the guy some quotes to read!"

We usually register all routes in the **routes.php** file.

Changing Laravel home page

To change Laravel home page, we need to edit the **routes.php** file.

Let's see what's inside the file.

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

We have only one route by default.

This route tells Laravel to return the **welcome view** when someone make a **GET request** to our **root URL** (which represents by the /).

> **Note:** Since Laravel 5.1, the routes.php file has been changed. If you see a different routes.php file, you may have an older Laravel version, please upgrade your application to version **5.1** or newer.
>
> routes.php (version 5.0)

```
Route::get('/', 'WelcomeController@index');
Route::get('home', 'HomeController@index');
Route::controllers([
'auth' => 'Auth\AuthController',
'password' => 'Auth\PasswordController',
]);
```

So if we want to edit the home page, we need to modify the **welcome view**.

What is view and where is the welcome view?

Views contain the HTML served by our application. A simple view may look like a simple HTML file:

```
<html>
    <body>
      <p> A simple view </p>
    </body
</html>
```

All views are stored at the **resources/views** directory.

Go to the **views** folder, find a file called **welcome.blade.php**. It's the **welcome view**.

Open it:

```
<html>
    <head>
        <title>Laravel</title>

        <link href='//fonts.googleapis.com/css?family=Lato:100' rel='stylesheet'\
 type='text/css'>

        <style>
            body {
                margin: 0;
                padding: 0;
                width: 100%;
                height: 100%;
                color: #B0BEC5;
                display: table;
                font-weight: 100;
                font-family: 'Lato';
            }
```

```
            .container {
                text-align: center;
                display: table-cell;
                vertical-align: middle;
            }

            .content {
                text-align: center;
                display: inline-block;
            }

            .title {
                font-size: 96px;
                margin-bottom: 40px;
            }

            .quote {
                font-size: 24px;
            }
        </style>
    </head>
    <body>
        <div class="container">
            <div class="content">
                <div class="title">Laravel 5</div>
                <div class="quote">{{ Inspiring::quote() }}</div>
            </div>
        </div>
    </body>
</html>
```

This welcome view is used to display the home page. It just looks like a basic HTML file!

I assume that you've already known HTML, so you should understand the content of this file clearly.

If you don't, **w3schools** is a good place to learn HTML and PHP.

http://www.w3schools.com/html

It's time to modify the home page!

What should we do?...

How about display our own quote?

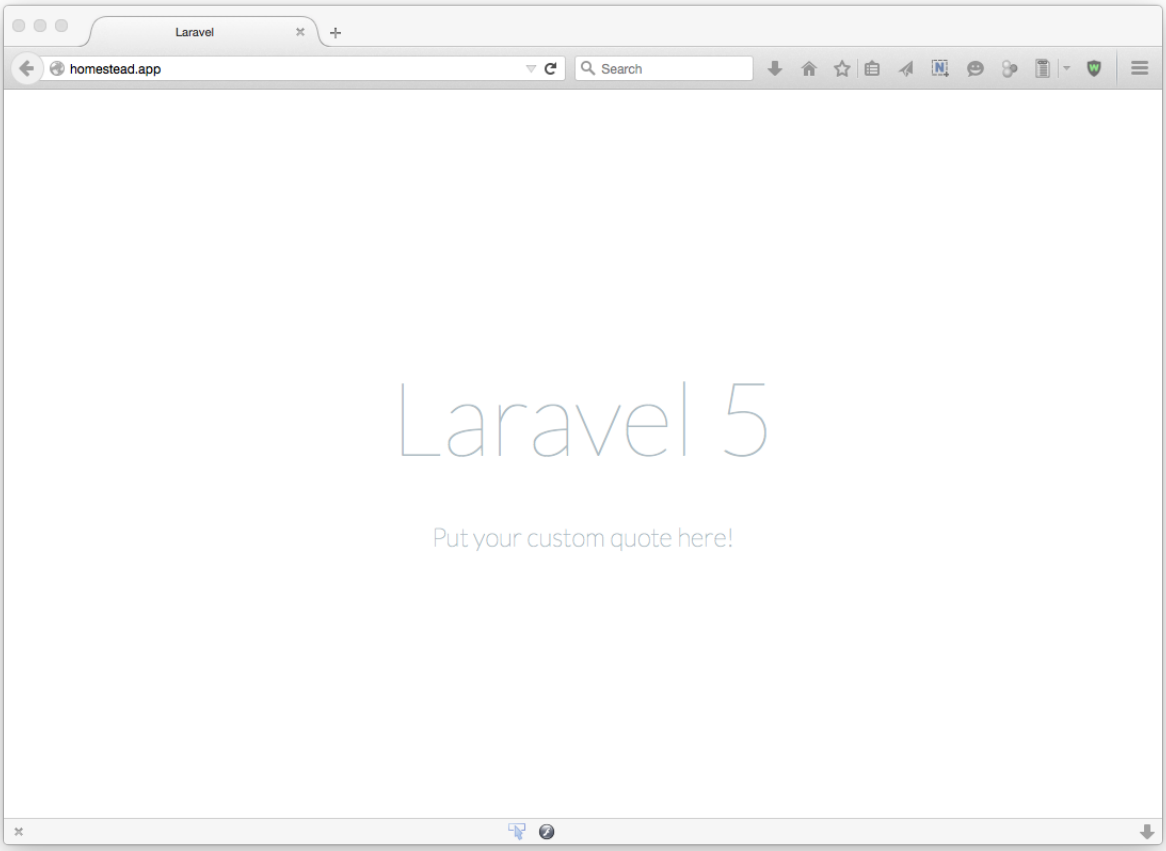
Let's change this line:

```
<div class="quote">{{ Inspiring::quote() }}</div>
```

to

```
<div class="quote">Put your custom quote here!</div>
```

Save the file, head over to your browser and run **homestead.app**.



Our new home page

Awesome! We have just changed our home page by changing the **welcome.blade.php** template!

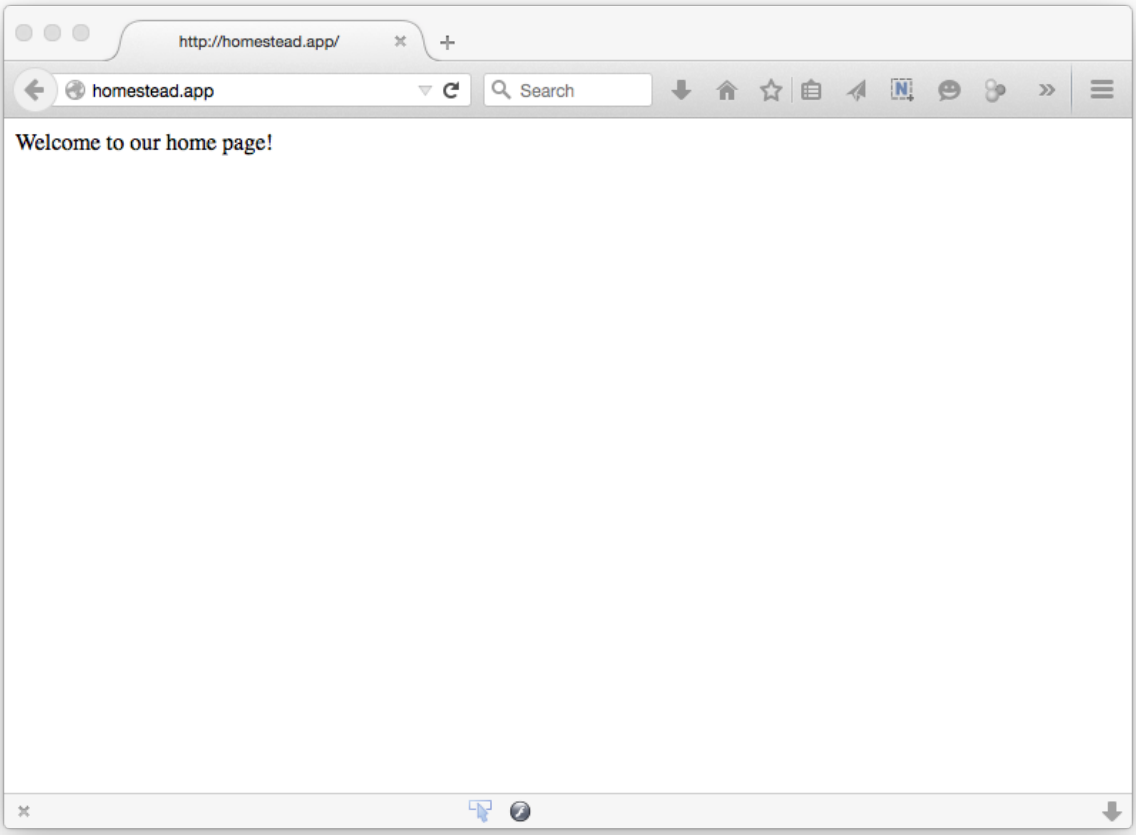
In fact, we can change any page without returning a view:

Modify the first route:

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

to

```
Route::get('/', function()
{
return 'Welcome to our home page!';
});
```



Our new home page

Amazing! Right?

We have just used a **anonymous function** to change our home page. In PHP, we called this function: "**Closure**".

A Closure is a function that doesn't have a name.

We use **Closure** to handle "routing" in small applications. In large applications, we use **Controllers**.

You can find Controllers documentation here:

http://laravel.com/docs/master/controllers

It's recommended that you should always use Controllers because Controllers help to structure your code easier. For instance, you may group all user actions into UserController, all post actions into PostsController.

The disadvantage of Controllers is, you will need to create a file for each Controller, thus it takes a bit more time.

To understand what **Controllers** is, we will be creating some pages using **Controllers**.

Adding more pages to our first website

When we have a basic understanding of how we can edit the home page, adding more pages is easy.

Imagine that we're going to build a website to introduce Learning Laravel book. The website will have three pages:

1. Home page.
2. About page.
3. Contact page.

Because we have many pages, and we might add more pages into our website, we should organize all our pages in **PagesController**.

As you've noticed, we don't have the **PagesController** yet, thus we're going to create it.

Create our first controller

There are two methods to create a controller:

Create a controller manually

To start off with things, create a new file called **PagesController.php** with your favourite text editor (PHPStorm, Sublime Text, etc.).

Place the file in **app/Http/Controllers/** directory.

Update the **PagesController.php** file to look like this:

```
<?php namespace App\Http\Controllers;

class PagesController extends Controller {

    public function home()
    {
        return view('welcome');
    }

}
```

Good job! You now have PagesController with a **home action**.

You may notice that the home action tells Laravel to "**return the welcome view**":

```
return view('welcome');
```

Create a controller by using Artisan

Rather than manually creating a controller, we can use **Artisan** to generate it automatically.

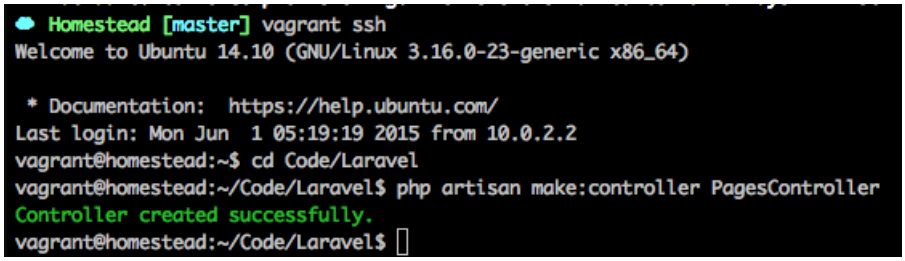
Artisan is Laravel command line tool that helps us to perform tasks that we hate to do manually. Using Artisan, we can create models, views, controllers, migrations and many other things.

You have known how to use Terminal or Git Bash, let's create a controller by running this command:

```
php artisan make:controller PagesController
```

> **Note**: **vagrant ssh** to your VM, **cd** to Laravel folder, and use the command there.

You'll see:



Our first controller

By default, Laravel creates a RESTful controller. Therefore, our **PagesController** class will have these actions:

1. index
2. show
3. create
4. store
5. edit
6. update
7. destroy

We don't need any of these, you can just remove them all and replace them with a home action:

```
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

class PagesController extends Controller {

    public function home()
    {
        return view('welcome');
    }

}
```

Alternatively, you can use this command to generate a new **PagesController** that doesn't have any action:

```
php artisan make:controller PagesController --plain
```

> **Note:** Since Laravel 5.2, the "–plain" flag has been removed. The controllers will always be plain.

Good job! You now have PagesController with the **home action**.

You may notice that the home action tells Laravel to "**return the welcome view**":

```
return view('welcome');
```

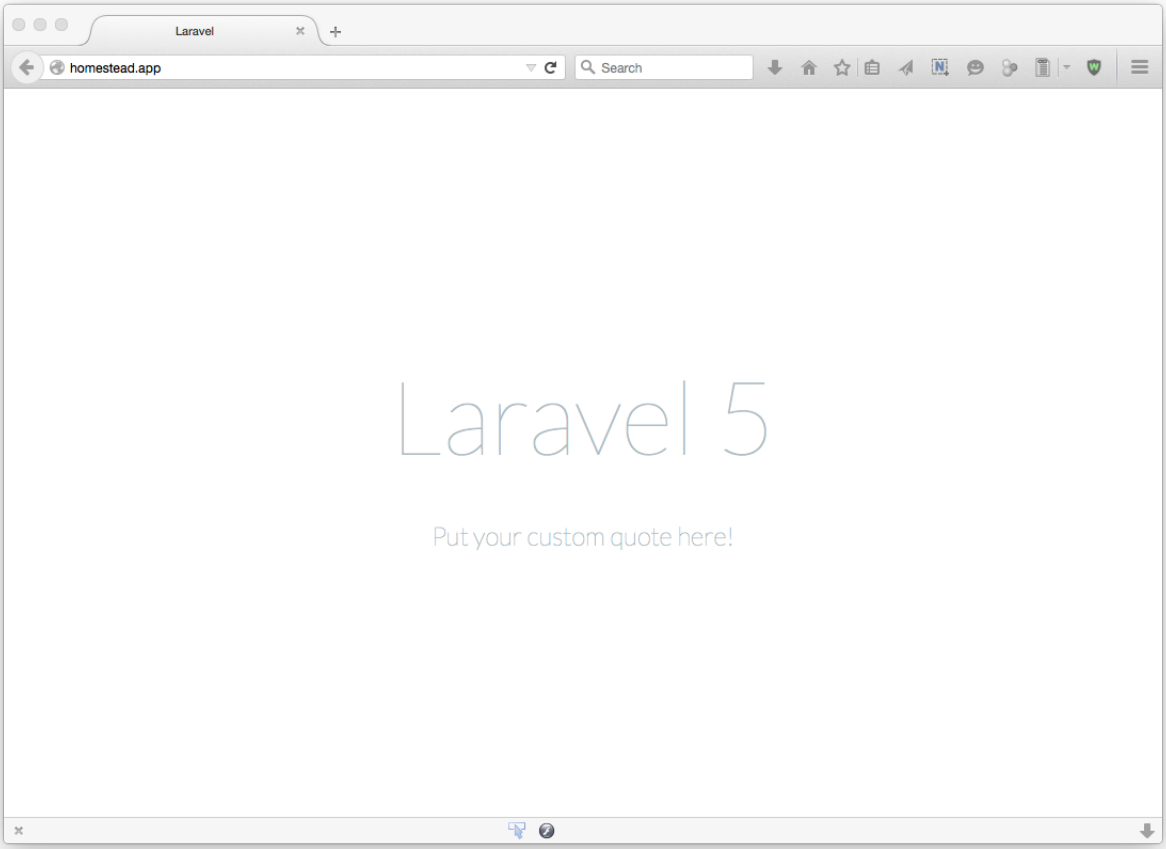
Using our first controller

Cool! When you have **PagesController**, the next thing to do is modifying our **routes**!

Open **routes.php** file. Change the default route to:

```
Route::get('/', 'PagesController@home');
```

This route tells Laravel to execute the home action (which can be found in PagesController) when someone make a GET request to our root URL (which represents by the /).



Display a home page using your first controller

Well done! You've just displayed the front page using your own controller!

Create other pages

Now that we have **PagesController** class. Adding more pages is not difficult.

> How about trying to add the about and contact page yourself?

Edit the **routes.php** file. Add the following lines:

```
Route::get('/about', 'PagesController@about');
Route::get('/contact', 'PagesController@contact');
```

Open **PagesController**, add

```
public function about()
{
    return view('about');
}

public function contact()
{
    return view('contact');
}
```

Next you will want to create **about view** and **contact view**. Create two new files in the **resources/views** directory named **about.blade.php** and **contact.blade.php**.

Finally, add the following contents (copy from the **welcome view**) to each file:

about.blade.php

```
<html>
    <head>
        <title>About Page</title>

        <link href='//fonts.googleapis.com/css?family=Lato:100' rel='stylesheet'\
 type='text/css'>

        <style>
            body {
                margin: 0;
                padding: 0;
                width: 100%;
                height: 100%;
                color: #B0BEC5;
                display: table;
                font-weight: 100;
                font-family: 'Lato';
            }

            .container {
                text-align: center;
```

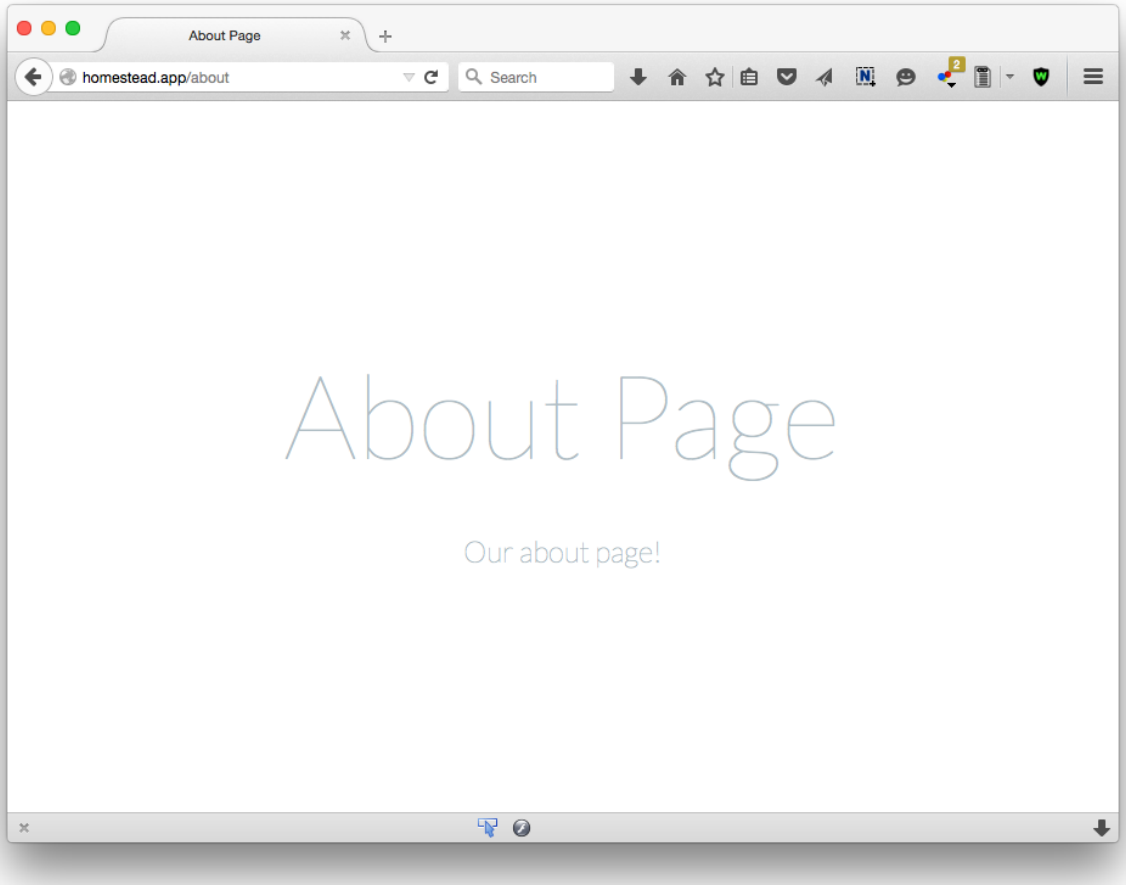
```
                display: table-cell;
                vertical-align: middle;
            }

            .content {
                text-align: center;
                display: inline-block;
            }

            .title {
                font-size: 96px;
                margin-bottom: 40px;
            }

            .quote {
                font-size: 24px;
            }
        </style>
    </head>
    <body>
        <div class="container">
            <div class="content">
                <div class="title">About Page</div>
                <div class="quote">Our about page!</div>
            </div>
        </div>
    </body>
</html>
```

contact.blade.php

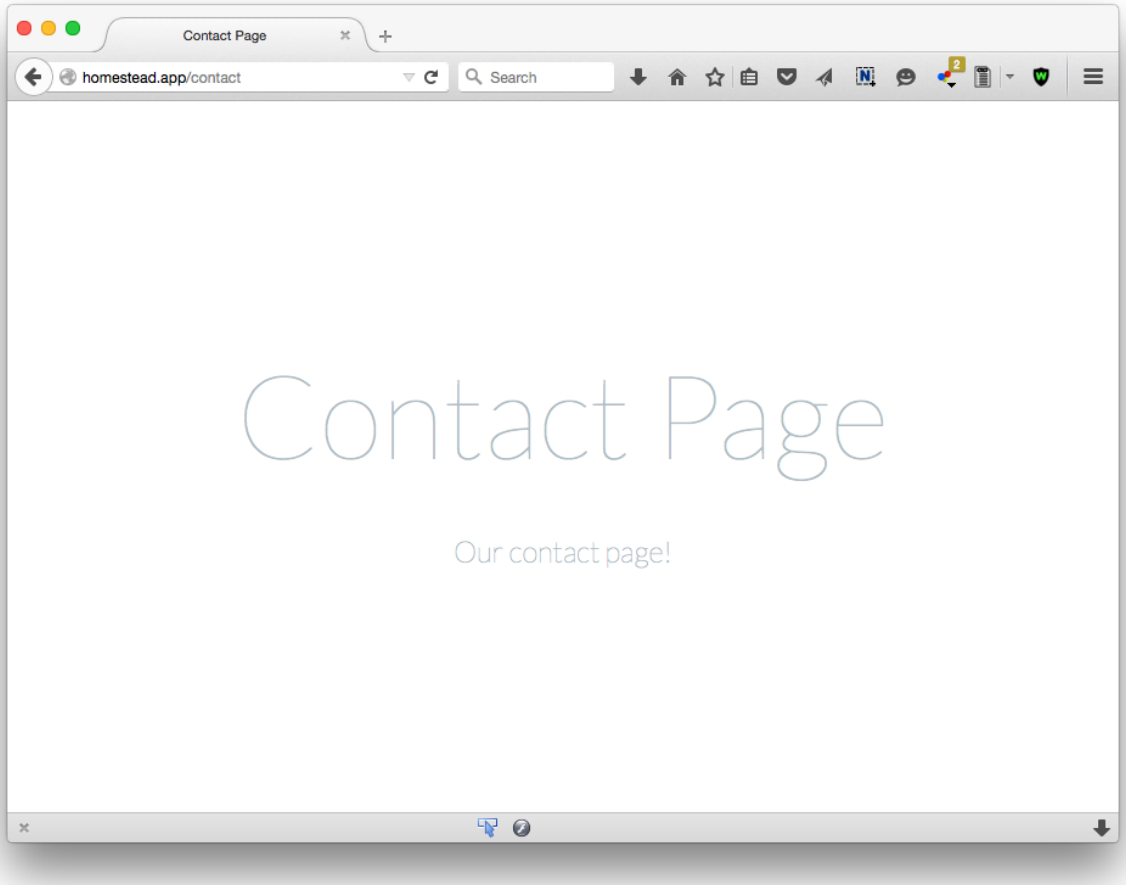
```
<html>
    <head>
        <title>Contact Page</title>

        <link href='//fonts.googleapis.com/css?family=Lato:100' rel='stylesheet'\
 type='text/css'>

        <style>
            body {
                margin: 0;
                padding: 0;
```

```
                width: 100%;
                height: 100%;
                color: #B0BEC5;
                display: table;
                font-weight: 100;
                font-family: 'Lato';
            }

            .container {
                text-align: center;
                display: table-cell;
                vertical-align: middle;
            }

            .content {
                text-align: center;
                display: inline-block;
            }

            .title {
                font-size: 96px;
                margin-bottom: 40px;
            }

            .quote {
                font-size: 24px;
            }
        </style>
    </head>
    <body>
    <div class="container">
        <div class="content">
            <div class="title">Contact Page</div>
            <div class="quote">Our contact page!</div>
        </div>
    </div>
    </body>
</html>
```

Save these changes, go to **homestead.app/about** and **homestead.app/contact**:


About Page
homestead.app/about
Search
About Page
Our about page!

About page



Contact page

Yayyyy! We have created the about and contact page with just a few lines of code!

Now, let's create a **home view** for our homepage, and change the **PagesController** to use it:

*Note: if you see the default home.blade.php file, just delete it and create a new file.

home view:

```
<html>
<head>
    <title>Home Page</title>

    <link href='//fonts.googleapis.com/css?family=Lato:100' rel='stylesheet' typ\
e='text/css'>

    <style>
        body {
            margin: 0;
            padding: 0;
            width: 100%;
            height: 100%;
            color: #B0BEC5;
            display: table;
            font-weight: 100;
            font-family: 'Lato';
        }

        .container {
            text-align: center;
            display: table-cell;
            vertical-align: middle;
        }

        .content {
            text-align: center;
            display: inline-block;
        }

        .title {
            font-size: 96px;
            margin-bottom: 40px;
        }

        .quote {
            font-size: 24px;
        }
    </style>
</head>
<body>
<div class="container">
```

```
    <div class="content">
        <div class="title">Home Page</div>
        <div class="quote">Our Home page!</div>
    </div>
</div>
</body>
</html>
```

PagesControllers

```
public function home()
{
    return view('home');
}
```

Integrating Twitter Bootstrap

Nowadays, the most popular front-end framework is **Twitter Bootstrap**. It's free, open source and has a large active community. I've been using Bootstrap for all projects.

Using Twitter Bootstrap, we can quickly develop responsive, mobile-ready web applications. Millions of beautiful and popular sites across the world are built with Bootstrap.

In this section, we will learn how to integrate Twitter Bootstrap into our Laravel application.

You can get Bootstrap and read its official documentation here:

http://getbootstrap.com

There are many ways to integrate Twitter Bootstrap. You can install it using Bootstrap CDN, Bower, npm, etc.

I'll show you the most three popular methods:

1. Using Bootstrap CDN
2. Using Precompiled Bootstrap Files
3. Using Bootstrap Source Code (Less)

Using Bootstrap CDN

The fastest way to integrate Twitter Bootstrap is using CDN.

Open **home.blade.php**, place these links inside the **head** tag

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css\
/bootstrap.min.css">
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css\
/bootstrap-theme.min.css">

<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.min.js\
"></script>
```

Done! You now have fully integrated Twitter Bootstrap into our website!

You may notice that we've added some CSS styles and Lato font into our **home.blade.php** file before. **Remove** those:

```
<style>
    body {
        margin: 0;
        padding: 0;
        width: 100%;
        height: 100%;
        color: #B0BEC5;
        display: table;
        font-weight: 100;
        font-family: 'Lato';
    }

    .container {
        text-align: center;
        display: table-cell;
        vertical-align: middle;
    }

    .content {
        text-align: center;
        display: inline-block;
    }

    .title {
        font-size: 96px;
        margin-bottom: 40px;
    }

    .quote {
```

```
        font-size: 24px;
    }
</style>
```

and

```
<link href='//fonts.googleapis.com/css?family=Lato:100' rel='stylesheet' type='t\
ext/css'>
```

Here is our new **home.blade.php**:

```
<html>
<head>
    <title>Home Page</title>

    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4\
/css/bootstrap.min.css">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4\
/css/bootstrap-theme.min.css">

    <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.mi\
n.js"></script>

</head>
<body>

<div class="container">
    <div class="content">
        <div class="title">Home Page</div>
        <div class="quote">Our Home page!</div>
    </div>
</div>
</body>
</html>
```

> Note: The bootstrap-theme.min.css is optional

Using Precompiled Bootstrap Files

Twitter Bootstrap is built using Less - a CSS pre-processor.

Less allows us to use variables, mixins, functions and other techniques to enhance CSS. You can learn more about Less here:

http://lesscss.org

Unfortunately, browsers don't understand Less. You must **compile** all Less files using **Less compiler** to produce CSS files. Of course, you have to learn Less.

Luckily, Bootstrap has provided compiled CSS, JS and fonts for us. We can download and use them without worrying about the Less files. Go to:

http://getbootstrap.com/getting-started

Click **Download Bootstrap** to download latest compiled Bootstrap files.

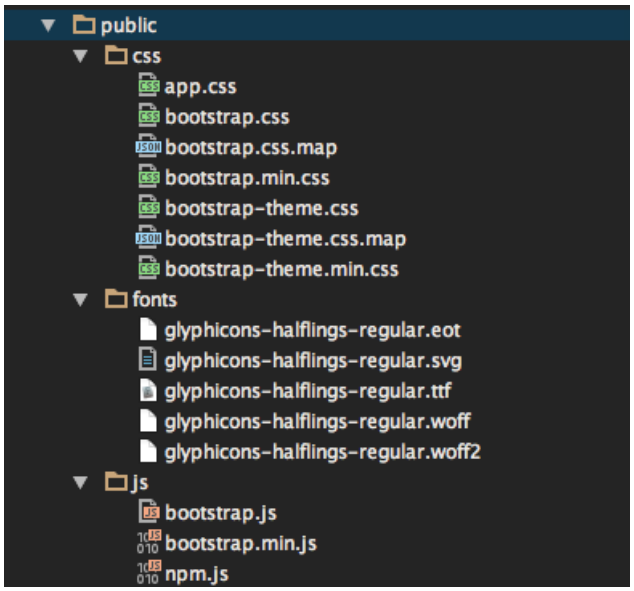
Uncompress the downloaded .zip file. We have three folders:

1. css
2. js
3. fonts

Put them all into the **public** folder of your app. (~/**Code/Laravel/public**).

> Note: By default, Laravel has created **css** and **fonts** folder for us. The **fonts** folder also contains all the **glyphicons** fonts.

When you visit your public folder, it should look like this:



Bootstrap navbar

To load Twitter Bootstrap framework for styling our home page, open **home.blade.php** and place these links inside the **head** tag

```
<link rel="stylesheet" type="text/css" href="{!! asset('css/bootstrap.min.css') \
!!}" >
<link rel="stylesheet" href="{!! asset('css/bootstrap-theme.min.css') !!}">
<script src="{!! asset('js/bootstrap.min.js') !!}"></script>
```

Twitter Bootstrap requires **jQuery** to work properly, you can download **jQuery** here:

https://jquery.com/download

Put the jQuery file into the **public** directory as well, then use the following code to reference it:

```
<script src="{!! asset('js/jquery-1.11.3.min.js') !!}"></script>
```

Or you can just use **jQuery CDN** without downloading the jQuery file:

```
<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
```

> Note: Your jQuery version may be different.

Full code:

```
<link rel="stylesheet" type="text/css" href="{!! asset('css/bootstrap.min.css') \
!!}" >
<link rel="stylesheet" href="{!! asset('css/bootstrap-theme.min.css') !!}">
<script src="{!! asset('js/jquery-1.11.3.min.js') !!}"></script>
<script src="{!! asset('js/bootstrap.min.js') !!}"></script>
```

Done! You now have fully integrated **Twitter Bootstrap** into our website!

We've used **asset** function to link CSS and JS files to our app. You can also use the **asset** function to link images, fonts and other public files. If you don't want to use asset function, you can use relative links instead:

```
<link rel="stylesheet" type="text/css" href="/css/bootstrap.min.css" >
<link rel="stylesheet" href="/css/bootstrap-theme.min.css">

<script src="/js/jquery-1.11.3.min.js"></script>
<script src="/js/bootstrap.min.js"></script>
```

You may notice that we've added some CSS styles and Lato font into our **home.blade.php** file before. **Remove** those:

```
<style>
    body {
        margin: 0;
        padding: 0;
        width: 100%;
        height: 100%;
        color: #B0BEC5;
        display: table;
        font-weight: 100;
        font-family: 'Lato';
    }

    .container {
        text-align: center;
        display: table-cell;
        vertical-align: middle;
    }

    .content {
        text-align: center;
        display: inline-block;
    }

    .title {
        font-size: 96px;
        margin-bottom: 40px;
    }

    .quote {
        font-size: 24px;
    }
</style>
```

and

```
<link href='//fonts.googleapis.com/css?family=Lato:100' rel='stylesheet' type='t\
ext/css'>
```

Here is our new **home.blade.php**:

```
<html>
<head>
    <title>Home Page</title>

    <link rel="stylesheet" type="text/css" href="{!! asset('css/bootstrap.min.cs\
s') !!}" >
    <link rel="stylesheet" href="{!! asset('css/bootstrap-theme.min.css') !!}">
    <script src="{!! asset('js/bootstrap.min.js') !!}"></script>

</head>
<body>

<div class="container">
    <div class="content">
        <div class="title">Home Page</div>
        <div class="quote">Our Home page!</div>
    </div>
</div>
</body>
</html>
```

Note: The bootstrap-theme.min.css is optional.

Using Bootstrap Source Code (Less)

The good news is, Laravel 5 has officially supported **Less**. Less files are placed at **resources/assets/less**.

First, we need to download Twitter Bootstrap source code. Go to:

http://getbootstrap.com/getting-started

Click **Download source** to download latest Bootstrap source files.

Uncompressed the downloaded .zip file. Rename the **less folder** to **bootstrap**. Place the bootstrap folder at **resources/assets/less/**

Open **app.less** file, import the **bootstrap.less** by using the following line:

```
@import "bootstrap/bootstrap";
```

Note: Since Laravel 5.1, Laravel has removed Bootstrap Less source files.

In Laravel 5, we use **Elixir** to automatically compile Less files to **app.css** file.

To load Twitter Bootstrap framework for styling our home page, open **home.blade.php** and place these links inside the **head** tag

```
<link rel="stylesheet" type="text/css" href="{!! asset('app.css') !!}" >
```

Twitter Bootstrap requires jQuery and Twitter Bootstrap JS to work properly, we need to integrate Bootstrap JS and jQuery:

```
<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.min.js\
"></script>
```

Alternatively, you can download the js files to the **js** directory and reference them locally:

```
<script src="{!! asset('js/jquery-1.11.3.min.js') !!}"></script>
<script src="{!! asset('js/bootstrap.min.js') !!}"></script>
```

Full code:

```
<link rel="stylesheet" type="text/css" href="{!! asset('app.css') !!}" >
<script src="{!! asset('js/jquery-1.11.3.min.js') !!}"></script>
<script src="{!! asset('js/bootstrap.min.js') !!}"></script>
```

Done! You now have fully integrated Twitter Bootstrap into your website!

We've used **asset** function to link CSS and JS files to our app. You can also use the **asset** function to link images, fonts and other public files. If you don't want to use **asset** function, you can use **relative links** instead:

```
<link rel="stylesheet" type="text/css" href="/css/app.css" >
<script src="/js/bootstrap.min.js"></script>
```

You may notice that we've added some **CSS styles** and **Lato font** into our **home.blade.php** file before. **Remove** those:

```
<style>
    body {
        margin: 0;
        padding: 0;
        width: 100%;
        height: 100%;
        color: #B0BEC5;
        display: table;
        font-weight: 100;
        font-family: 'Lato';
    }

    .container {
        text-align: center;
        display: table-cell;
        vertical-align: middle;
    }

    .content {
        text-align: center;
        display: inline-block;
    }

    .title {
        font-size: 96px;
        margin-bottom: 40px;
    }

    .quote {
        font-size: 24px;
    }
</style>
```

and

```
<link href='//fonts.googleapis.com/css?family=Lato:100' rel='stylesheet' type='t\
ext/css'>
```

Here is our new **home.blade.php**:

```
<html>
<head>
    <title>Home Page</title>

    <link rel="stylesheet" type="text/css" href="{!! asset('css/app.css') !!}" >
    <script src="{!! asset('js/bootstrap.min.js') !!}"></script>

</head>
<body>

<div class="container">
    <div class="content">
        <div class="title">Home Page</div>
        <div class="quote">Our Home page!</div>
    </div>
</div>
</body>
</html>
```

Oh, I've mentioned **Elixir**, what is it?

Introducting Elixir

One of the best Laravel 5 new features is **Elixir**. We can use Elixir to compile Less files, Coffee Scripts or automate other manual tasks.

Elixir official documentation

Basically, Elixir is Gulp's extension. If you don't know about Gulp yet, you can find more information about it here:

http://gulpjs.com

Gulp is a **Node.js** based task runner, that means if you want to use Elixir, you need to have both Gulp and Node.js installed.

Luckily, Homestead has Gulp and Node.js by default, we can use Gulp right away. If you don't use Homestead, you have to install Node.js and Gulp. To install Node.js, visit:

https://nodejs.org

Follow instructions on the site, you should have installed Node.js easily. After that, you can use **npm** command to install gulp:

```
npm install -g gulp
```

You can check if Gulp is installed by running:

```
gulp -v
```

If Gulp is installed, you should see something like:

```
[14:09:33] CLI version 3.8.11
[14:09:33] Local version 3.8.11
```

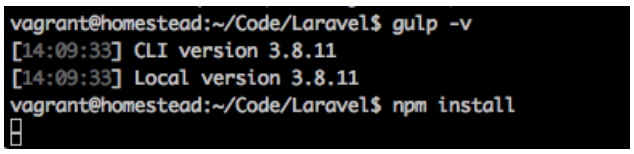
The last step is to install Elixir. Laravel 5 has included a file called **package.json**. You use this file to install Node modules. Open the file, you should see:

```
{
  "private": true,
  "devDependencies": {
    "gulp": "^3.8.8",
    "laravel-elixir": "^1.0.0"
  }
}
```

Do you see the **laravel-elixir** there?

Good! Navigate to our app root (~/Code/Laravel), run this command to install Elixir:

```
npm install
```



Installing Elixir

It may take a while to download. Be patient.

Once complete, there is a new folder called **node_modules** in our app. You can find Gulp, Elixir and other Node.js packages here.

Running first Elixir task

You can write Gulp task (or Elixir task) in **gulpfile.js**. For instance, you can find a Gulp task that compiles app.less file into app.css in the **gulpfile.js**:

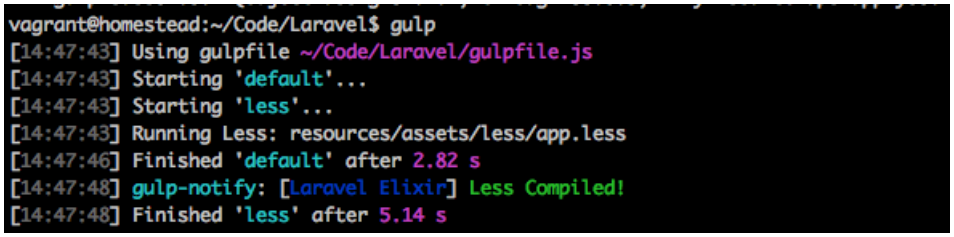
```
elixir(function(mix) {
    mix.less('app.less');
});
```

Feel free to add more tasks by reading the official Elixir documentation:

http://laravel.com/docs/master/elixir

To execute your Elixir tasks, run this command:

```
gulp
```



Executing Elixir task

By running the task, Laravel will automatically compile app.less and save the output to **app.css**. The file can be found in your **public/css** directory.

Adding Twitter Bootstrap components

Cool! Now we can add **Twitter Bootstrap components** into our website.

There are many reusable Bootstrap components built to provide navbars, labels, dropdowns, panels, etc. You can see a full list of components here:

http://getbootstrap.com/components

To use the components, you can copy the example codes, and paste them to your application. For instance, we can add Twitter Bootstrap navbar to our app by adding these codes to **home.blade.php**:

```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <!-- Brand and toggle get grouped for better mobile display -->
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collaps\
e" data-target="#bs-example-navbar-collapse-1">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="#">Brand</a>
    </div>

    <!-- Collect the nav links, forms, and other content for toggling -->
    <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
      <ul class="nav navbar-nav">
        <li class="active"><a href="#">Link <span class="sr-only">(current)</spa\
n></a></li>
        <li><a href="#">Link</a></li>
        <li class="dropdown">
          <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="butto\
n" aria-expanded="false">Dropdown <span class="caret"></span></a>
          <ul class="dropdown-menu" role="menu">
            <li><a href="#">Action</a></li>
            <li><a href="#">Another action</a></li>
            <li><a href="#">Something else here</a></li>
            <li class="divider"></li>
            <li><a href="#">Separated link</a></li>
            <li class="divider"></li>
            <li><a href="#">One more separated link</a></li>
          </ul>
        </li>
      </ul>
      <form class="navbar-form navbar-left" role="search">
        <div class="form-group">
          <input type="text" class="form-control" placeholder="Search">
        </div>
        <button type="submit" class="btn btn-default">Submit</button>
      </form>
      <ul class="nav navbar-nav navbar-right">
        <li><a href="#">Link</a></li>
```

```
        <li class="dropdown">
          <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="butto\
n" aria-expanded="false">Dropdown <span class="caret"></span></a>
          <ul class="dropdown-menu" role="menu">
            <li><a href="#">Action</a></li>
            <li><a href="#">Another action</a></li>
            <li><a href="#">Something else here</a></li>
            <li class="divider"></li>
            <li><a href="#">Separated link</a></li>
          </ul>
        </li>
      </ul>
    </div><!-- /.navbar-collapse -->
  </div><!-- /.container-fluid -->
</nav>
```

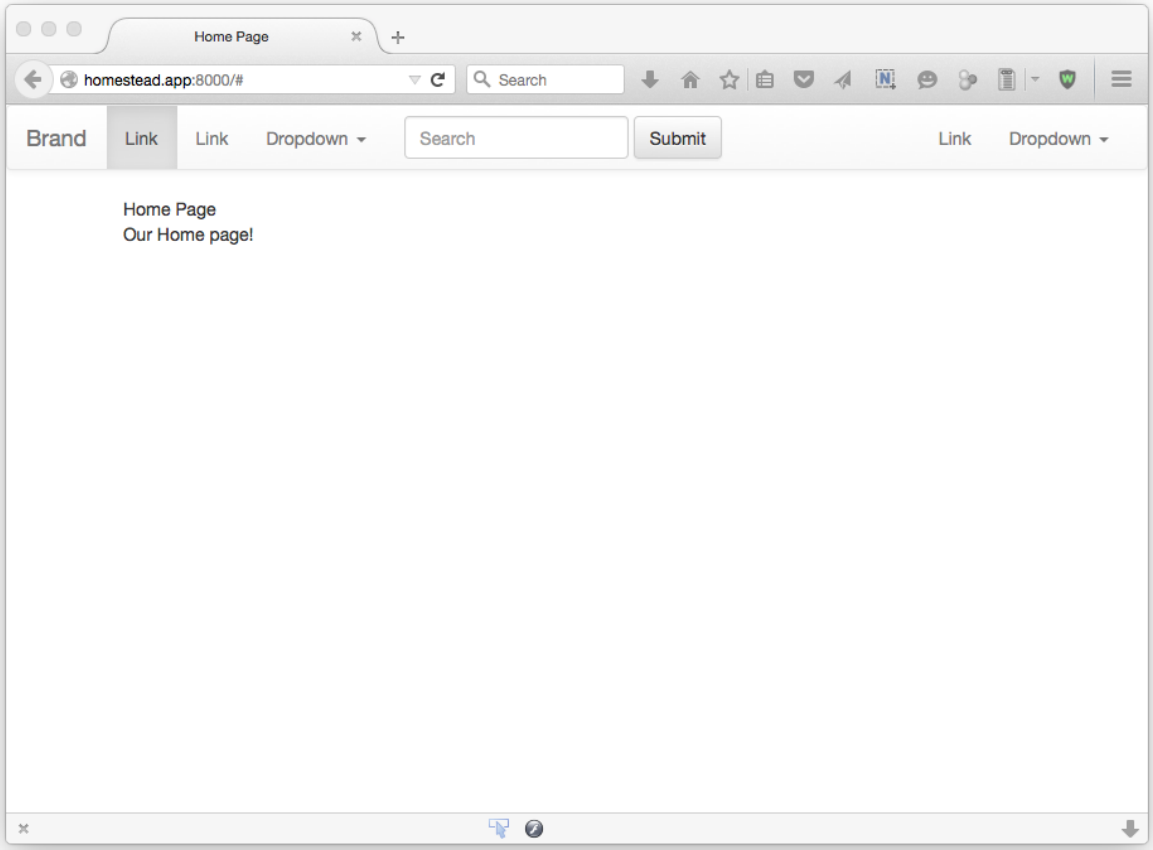
Here is our new home.blade.php after adding the navbar:

```
<html>
<head>
    <title>Home Page</title>

    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4\
/css/bootstrap.min.css">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4\
/css/bootstrap-theme.min.css">
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.mi\
n.js"></script>


</head>
<body>

<nav class="navbar navbar-default">
    <div class="container-fluid">
        <!-- Brand and toggle get grouped for better mobile display -->
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="c\
ollapse" data-target="#bs-example-navbar-collapse-1">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
```

```
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">Brand</a>
        </div>

        <!-- Collect the nav links, forms, and other content for toggling -->
        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
            <ul class="nav navbar-nav">
                <li class="active"><a href="#">Link <span class="sr-only">(curre\
nt)</span></a></li>
                <li><a href="#">Link</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown" r\
ole="button" aria-expanded="false">Dropdown <span class="caret"></span></a>
                    <ul class="dropdown-menu" role="menu">
                        <li><a href="#">Action</a></li>
                        <li><a href="#">Another action</a></li>
                        <li><a href="#">Something else here</a></li>
                        <li class="divider"></li>
                        <li><a href="#">Separated link</a></li>
                        <li class="divider"></li>
                        <li><a href="#">One more separated link</a></li>
                    </ul>
                </li>
            </ul>
            <form class="navbar-form navbar-left" role="search">
                <div class="form-group">
                    <input type="text" class="form-control" placeholder="Search">
                </div>
                <button type="submit" class="btn btn-default">Submit</button>
            </form>
            <ul class="nav navbar-nav navbar-right">
                <li><a href="#">Link</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown" r\
ole="button" aria-expanded="false">Dropdown <span class="caret"></span></a>
                    <ul class="dropdown-menu" role="menu">
                        <li><a href="#">Action</a></li>
                        <li><a href="#">Another action</a></li>
                        <li><a href="#">Something else here</a></li>
                        <li class="divider"></li>
                        <li><a href="#">Separated link</a></li>
```

```
                    </ul>
                </li>
            </ul>
        </div><!-- /.navbar-collapse -->
    </div><!-- /.container-fluid -->
</nav>

<div class="container">
    <div class="content">
        <div class="title">Home Page</div>
        <div class="quote">Our Home page!</div>
    </div>
</div>
</body>
</html>
```

It's time to refresh our browser:



Bootstrap navbar

Learning Blade templates

It's time to learn about **Blade**!

Blade is an official Laravel's templating engine. It's very powerful, but it has very simple syntax. We use Blade to build layouts for our Laravel applications.

Blade view files have **.blade.php** file extension. The **home view** and **other views** that we've been using are Blade templates.

Usually, we put all Blade templates in **resources/views** directory. The great thing is, we can use plain PHP code in a Blade view.

All Blade expressions begin with **@**. For example: @section, @if, @for, etc.

Blade also supports all PHP loops and conditions: **@if**, **@elseif**, **@else**, **@for**, **@ foreach**, **@while**, **etc**. For instance, you can write a if else statement in Blade like this:

```
@if ($product == 1)
 {!! $product->name !!}
@else
There is no product!
@endif
```

Equivalent PHP code:

```
if ($product ==1) {
echo $product->name;
} else {
echo("There is no product!");
}
```

To understand Blade's features, we will use Blade to build our first website's layout.

Creating a master layout

The typical web application has a **master layout**. The layout consists header, footer, sidebar, etc. Using a master layout, we can easily place one element on every view. For instance, we can use the same header and footer for all pages. It helps to make our code look clearer and save our time a lot.

To get started, we will create a master layout called **master.blade.php**

```
<html>
<head>
    <title> @yield('title') </title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4\
/css/bootstrap.min.css">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4\
/css/bootstrap-theme.min.css">

    <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.mi\
n.js"></script>
</head>
<body>

@include('shared.navbar')

@yield('content')

</body>
</html>
```

This view look likes the home view, but we've changed something. Let's see the code line by line.

```
<title> @yield('title') </title>
```

Instead of putting a title here, we use **@yield** directive to get the title from another section.

```
@include('shared.navbar')
```

We use **@include** directive to include other Blade views. You may notice that we've embed a view called **navbar** here.

However, we don't have the navbar view yet, let's create a **shared** directory and put **navbar.blade.php** there.

Copy the Twitter Bootstrap navbar and put it into the **navbar.blade.php**:

```
<nav class="navbar navbar-default">
    <div class="container-fluid">
        <!-- Brand and toggle get grouped for better mobile display -->
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="c\
ollapse" data-target="#bs-example-navbar-collapse-1">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">Brand</a>
        </div>

        <!-- Collect the nav links, forms, and other content for toggling -->
        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
            <ul class="nav navbar-nav">
                <li class="active"><a href="#">Link <span class="sr-only">(curre\
nt)</span></a></li>
                <li><a href="#">Link</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown" r\
ole="button" aria-expanded="false">Dropdown <span class="caret"></span></a>
                    <ul class="dropdown-menu" role="menu">
                        <li><a href="#">Action</a></li>
                        <li><a href="#">Another action</a></li>
                        <li><a href="#">Something else here</a></li>
```

```
                        <li class="divider"></li>
                        <li><a href="#">Separated link</a></li>
                        <li class="divider"></li>
                        <li><a href="#">One more separated link</a></li>
                    </ul>
                </li>
            </ul>
            <form class="navbar-form navbar-left" role="search">
                <div class="form-group">
                    <input type="text" class="form-control" placeholder="Search">
                </div>
                <button type="submit" class="btn btn-default">Submit</button>
            </form>
            <ul class="nav navbar-nav navbar-right">
                <li><a href="#">Link</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown" r\
ole="button" aria-expanded="false">Dropdown <span class="caret"></span></a>
                    <ul class="dropdown-menu" role="menu">
                        <li><a href="#">Action</a></li>
                        <li><a href="#">Another action</a></li>
                        <li><a href="#">Something else here</a></li>
                        <li class="divider"></li>
                        <li><a href="#">Separated link</a></li>
                    </ul>
                </li>
            </ul>
        </div><!-- /.navbar-collapse -->
    </div><!-- /.container-fluid -->
</nav>
```

You may change the shared folder name to partials, embed or whatever you like.

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

As you see it's really convenient for us to not display the content of any pages here. We simply use **@yield** directive to embed a section called **content** from other views.

Great! You've just created a master layout!

Extending the master layout

Now we can change the **home** view to extend our master layout.

```
@extends('master')
@section('title', 'Home')

@section('content')
    <div class="container">
        <div class="content">
            <div class="title">Home Page</div>
            <div class="quote">Our Home page!</div>
        </div>
    </div>
@endsection
```

As you can observe we use **@extends** directive to inherit our **master layout**.

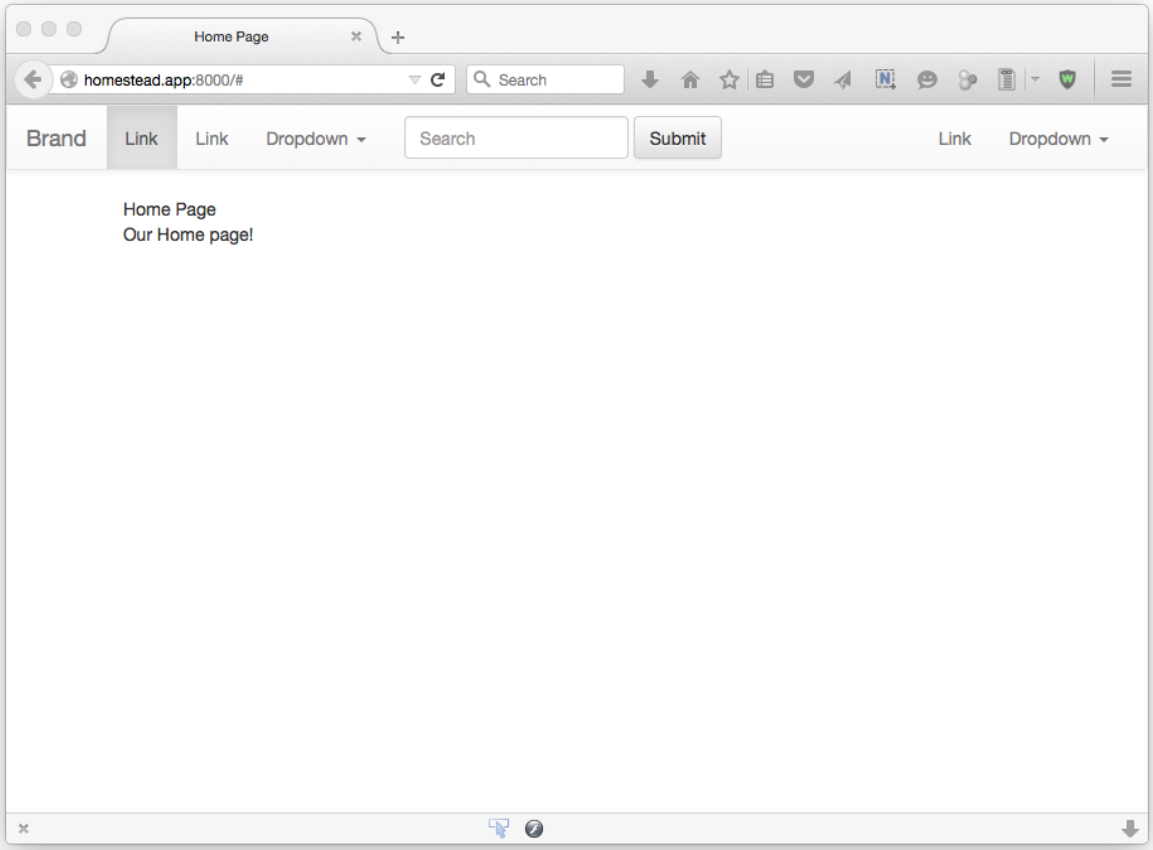
To set the title for our home page, we use **@section** directive.

```
@section('title', 'Home')
```

It's the "short way". If we have a long content, we can use **@section** and **@endsection** to inject our **content** into the master layout.

```
@section('content')
    <div class="container">
        <div class="content">
            <div class="title">Home Page</div>
            <div class="quote">Our Home page!</div>
        </div>
    </div>
@endsection
```

Refresh your browser, you should see the same home page, but this time our code look much cleaner.



The home page using master layout

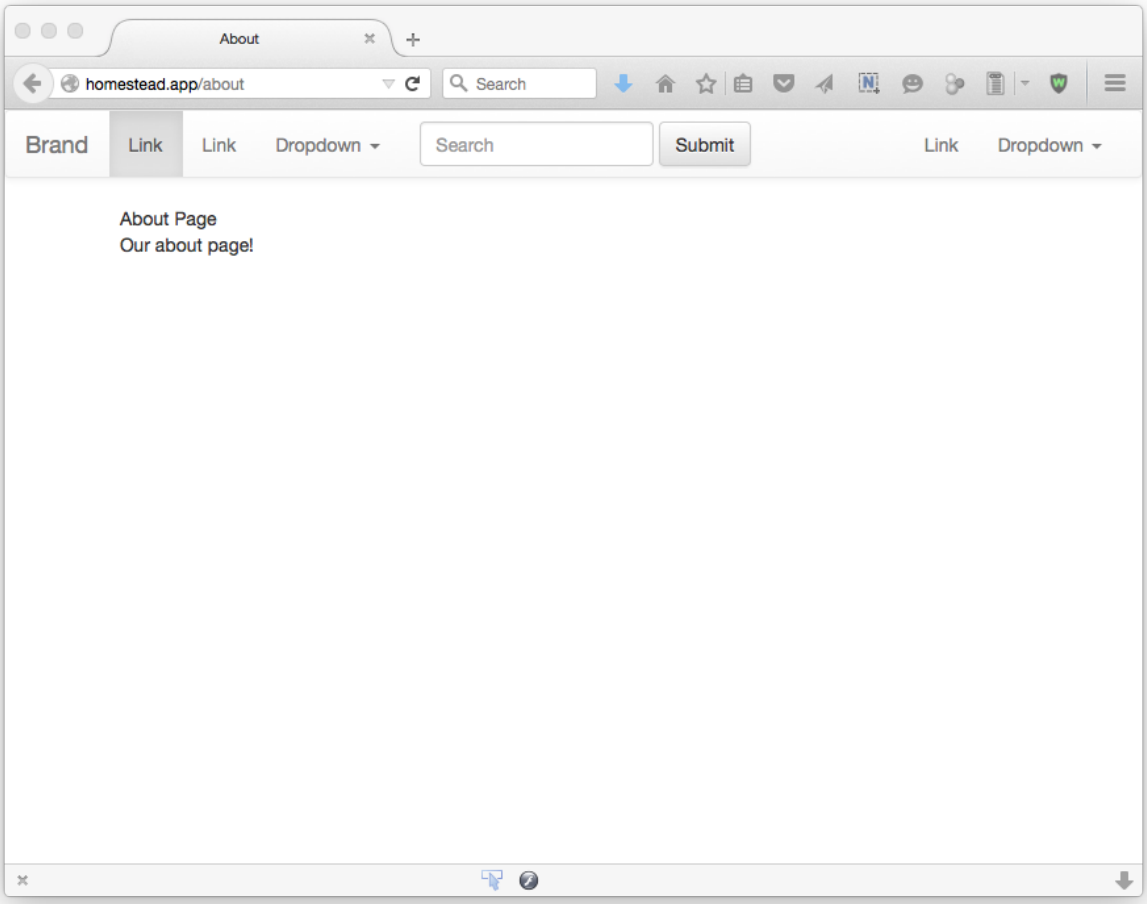
Using the same technique, we can easily change the about and contact page:

about.blade.php

```
@extends('master')
@section('title', 'About')

@section('content')
    <div class="container">
        <div class="content">
            <div class="title">About Page</div>
            <div class="quote">Our about page!</div>
        </div>
    </div>
@endsection
```

contact.blade.php

```
@extends('master')
@section('title', 'Contact')

@section('content')
    <div class="container">
        <div class="content">
            <div class="title">Contact Page</div>
            <div class="quote">Our contact page!</div>
        </div>
    </div>
@endsection
```



new about page

Using other Bootstrap themes

The best thing about Twitter Bootstrap is, it has many themes for you to "switch". If you don't like the default Bootstrap theme, you can easily find another one to use. Here are some popular themes for Bootstrap:

1. http://bootswatch.com
2. http://fezvrasta.github.io/bootstrap-material-design
3. http://designmodo.github.io/Flat-UI

In this section, we're going to apply **Bootstrap Material Design** theme for our website.

This Bootstrap theme will "provide an easy way to use the new Material Design guidelines by Google". If you're using an Android phone, you may have seen Material Design already. You can find out more about Material Design here:

http://www.google.com/design/spec/material-design/introduction.html

This section is designed to test your knowledge of the Laravel 5 structure and views. I'll additionally show you how to manage your assets. Feel free to use other themes if you want. We also build a good template to use for all applications of this book.

First, head over to:

https://github.com/FezVrasta/bootstrap-material-design

Download the zip file. Uncompressed it. Go to the **dist** directory. There are 3 folders:

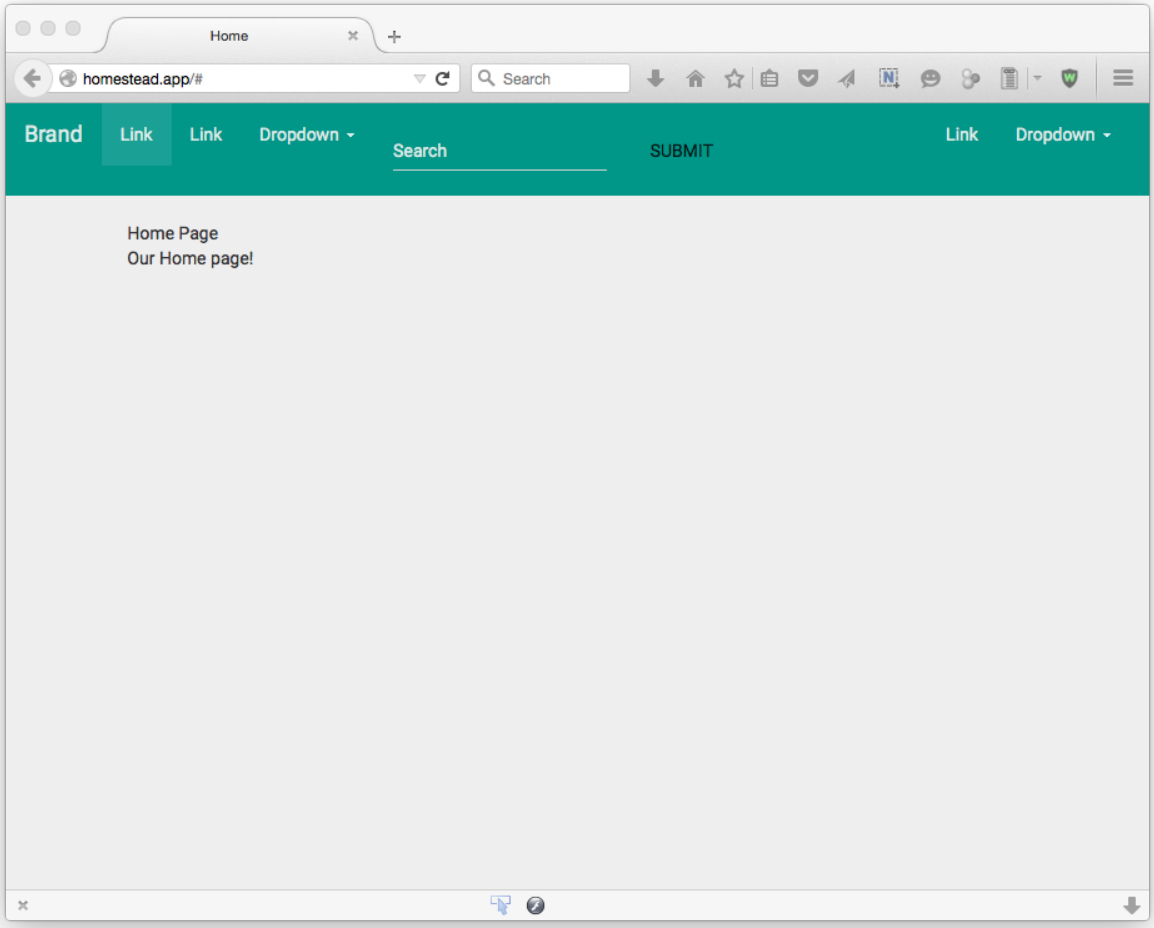
1. css
2. js
3. fonts

Copy **css**, **fonts**, and **js** directory to our **Laravel public** folder. **(~/Code/Laravel/public)**.

Open **master.blade.php**, modify its content to look like this:

```
<html>
<head>
    <title> @yield('title') </title>
    <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css"\
 rel="stylesheet">
    <!-- Include roboto.css to use the Roboto web font, material.css to include \
the theme and ripples.css to style the ripple effect -->
    <link href="/css/roboto.min.css" rel="stylesheet">
    <link href="/css/material.min.css" rel="stylesheet">
    <link href="/css/ripples.min.css" rel="stylesheet">

</head>
<body>

@include('shared.navbar')


@yield('content')


<script src="//code.jquery.com/jquery-1.10.2.min.js"></script>
<script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.2/js/bootstrap.min.js"></sc\
ript>

<script src="/js/ripples.min.js"></script>
<script src="/js/material.min.js"></script>
<script>
    $(document).ready(function() {
        // This command is used to initialize some elements and make them work p\
roperly
        $.material.init();
    });
</script>
</body>


</html>
```

Congratulations! You now have a beautiful Material Design website!



Material Design home page

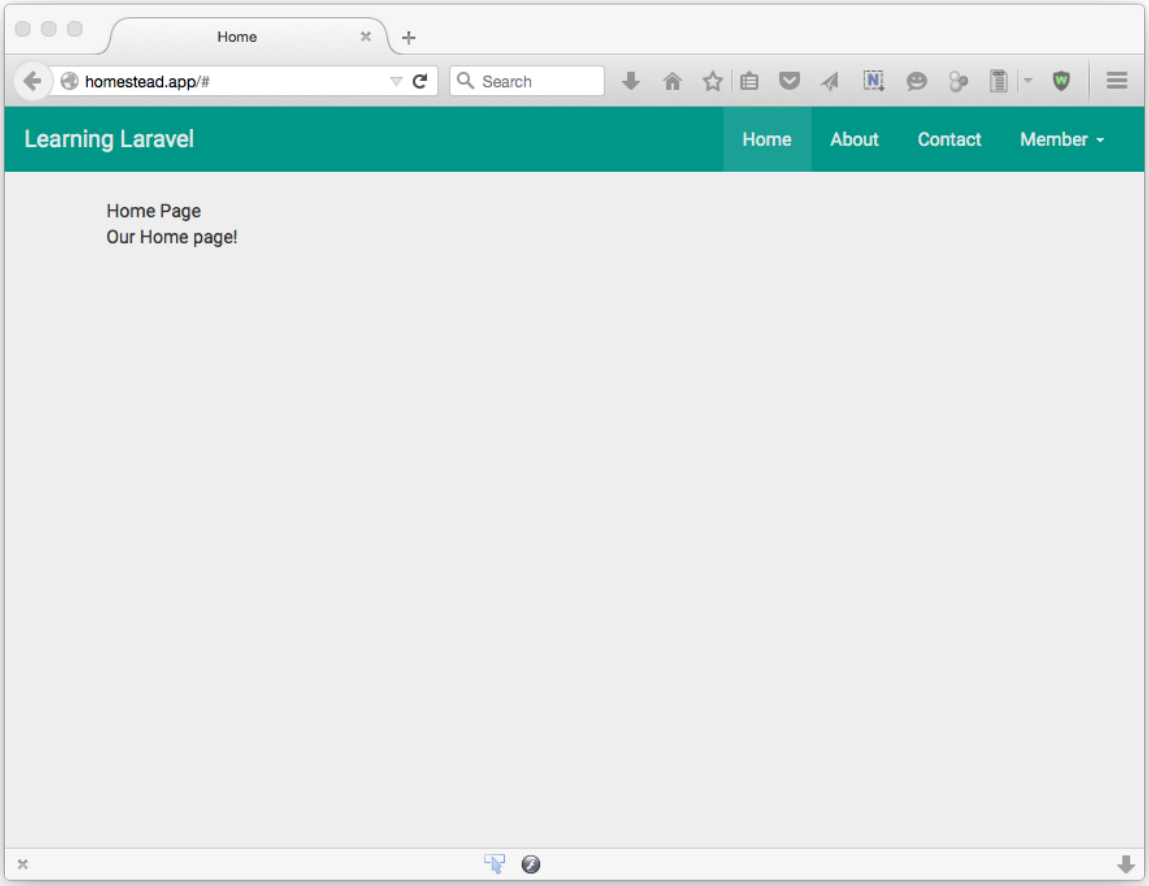
Refine our website layouts

Currently, the site looks messy. We're going to change a few things and re-design all views to make our application look better and professional.

Changing the navbar

Open **navbar.blade.php**, update it:

```
<nav class="navbar navbar-default">
    <div class="container-fluid">
        <!-- Brand and toggle get grouped for better mobile display -->
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="c\
ollapse" data-target="#bs-example-navbar-collapse-1">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">Learning Laravel</a>
        </div>

        <!-- Navbar Right -->
        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
            <ul class="nav navbar-nav navbar-right">
                <li class="active"><a href="/">Home</a></li>
                <li><a href="/about">About</a></li>
                <li><a href="/contact">Contact</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown" r\
ole="button" aria-expanded="false">Member <span class="caret"></span></a>
                    <ul class="dropdown-menu" role="menu">
                        <li><a href="/users/register">Register</a></li>
                        <li><a href="/users/login">Login</a></li>
                    </ul>
                </li>
            </ul>
        </div>
    </div>
</nav>
```



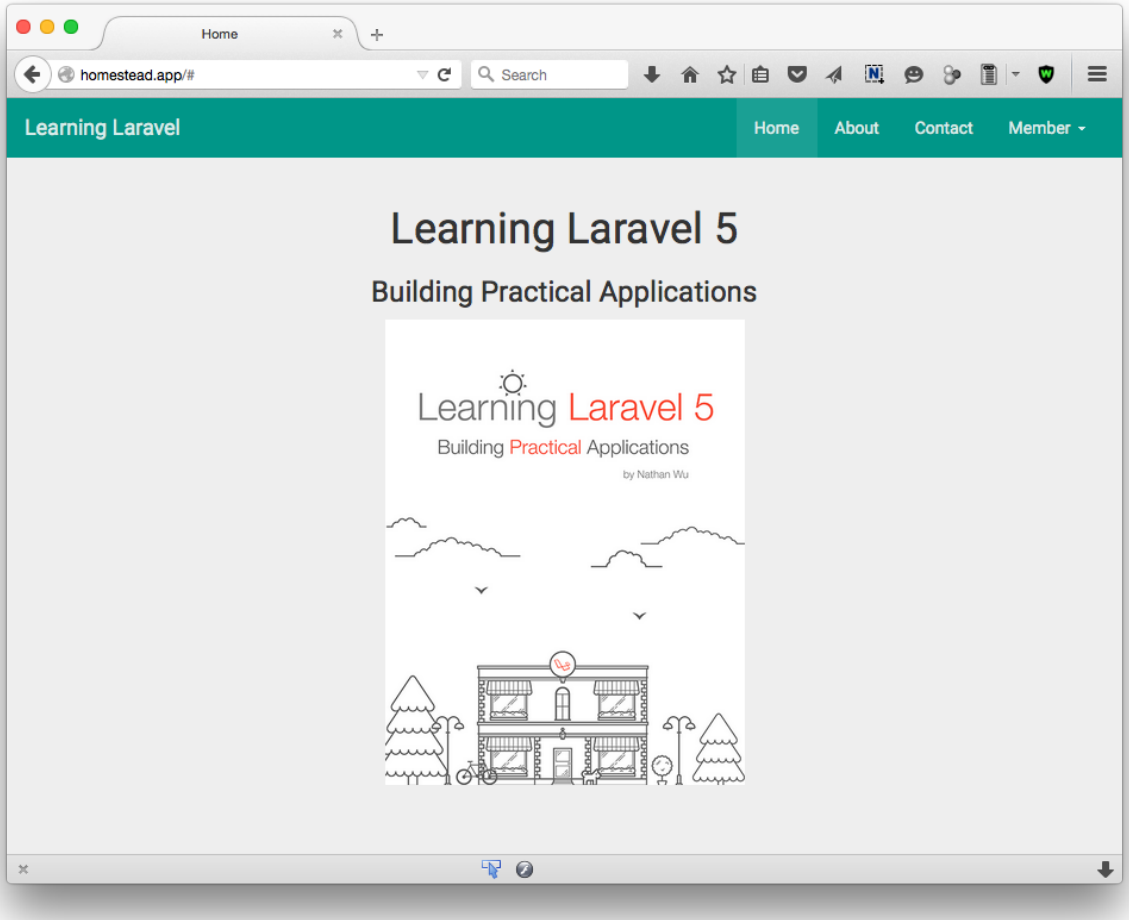
A new navbar

Changing the home page

Open **home.blade.php**, update it:

```
@extends('master')
@section('title', 'Home')

@section('content')

    <div class="container">
        <div class="row banner">

            <div class="col-md-12">
```

```
                <h1 class="text-center margin-top-100 editContent">
                    Learning Laravel 5
                </h1>

                <h3 class="text-center margin-top-100 editContent">Building Prac\
tical Applications</h3>

                <div class="text-center">
                    <img src="http://learninglaravel.net/img/LearningLaravel5_co\
ver0.png" width="302" height="391" alt="">
                </div>

            </div>

        </div>
    </div>

@endsection
```



The new home page

We now have a cool responsive home page.

Responsive website

You may try to navigate to the about and contact page to see if everything is working correctly. Feel free to change minor things like images, contents, colors, and fonts to your liking.

Chapter 2 Summary

Good job! We now have a fully responsive template! We will use this template to build other applications to learn more about Laravel.

In this chapter, you've learned many things:

1. You've known about Laravel structure, how Laravel works and where to put the files.
2. You've learned about Laravel routes.
3. You've learned Controllers. Now you can be able to create web pages using Controllers.
4. You've known what Blade is. It's easy to create Blade templates for your next amazing applications.
5. You've known how to integrate Twitter Bootstrap, CSS, JS and apply different Bootstrap themes.
6. You've known Elixir, how to install Gulp, and how to create a basic Gulp task.

In the next chapter, we will learn how to create a basic CRUD (Create, Read, Update, Delete) application to learn more about Laravel's features.

Chapter 3: Building A Support Ticket System

In this chapter, we will build a support ticket system to learn about Laravel main features, such as Eloquent ORM, Eloquent Relationships, Migrations, Requests, Laravel Collective, sending emails, etc.

While the project design is simple, it provides an excellent way to learn Laravel.

What do we need to get started?

I assume that you have followed the instructions provided in the previous chapter and you've created a basic website. You will need that basic application to start building the support ticket system.

What will we build?

We'll start by laying down the basic principle behind the application's creation, and a summary of how it works.

Our ticket system is simple:

- When users visit the contact page, they will be able to submit a ticket to contact us.
- Once they've created a ticket, the system will send us an email to let us know that there is a new ticket.
- The ticket system automatically generates a unique link to let us access the ticket.
- We can view all the tickets.
- We can be able to reply, edit, change tickets' status or delete them.

Let's start building things!

Laravel Database Configuration

Our application requires a database to work. Laravel supports many database platforms, including:

1. MySQL

2. SQLite
3. PostgreSQL
4. SQL Server

> **Note**: A database is a collection of data. We use database to store, manage and update our data easier.

The great thing is, we can choose any of them to develop our applications. In this book, we will use **MySQL**.

You can configure databases using **database.php** file, which is placed in the **config directory**.

config/database.php

```
<?php

return [

    /*
    |--------------------------------------------------------------------------
    | PDO Fetch Style
    |--------------------------------------------------------------------------
    |
    | By default, database results will be returned as instances of the PHP
    | stdClass object; however, you may desire to retrieve records in an
    | array format for simplicity. Here you can tweak the fetch style.
    |
    */

    'fetch' => PDO::FETCH_CLASS,

    /*
    |--------------------------------------------------------------------------
    | Default Database Connection Name
    |--------------------------------------------------------------------------
    |
    | Here you may specify which of the database connections below you wish
    | to use as your default connection for all database work. Of course
    | you may use many connections at once using the Database library.
    |
    */

    'default' => env('DB_CONNECTION', 'mysql'),
```

```
    /*
    |--------------------------------------------------------------------------
    | Database Connections
    |--------------------------------------------------------------------------
    |
    | Here are each of the database connections setup for your application.
    | Of course, examples of configuring each database platform that is
    | supported by Laravel is shown below to make development simple.
    |
    |
    | All database work in Laravel is done through the PHP PDO facilities
    | so make sure you have the driver for your particular database of
    | choice installed on your machine before you begin development.
    |
    */

    'connections' => [

        'sqlite' => [
            'driver'   => 'sqlite',
            'database' => storage_path('database.sqlite'),
            'prefix'   => '',
        ],

        'mysql' => [
            'driver'    => 'mysql',
            'host'      => env('DB_HOST', 'localhost'),
            'database'  => env('DB_DATABASE', 'forge'),
            'username'  => env('DB_USERNAME', 'forge'),
            'password'  => env('DB_PASSWORD', ''),
            'charset'   => 'utf8',
            'collation' => 'utf8_unicode_ci',
            'prefix'    => '',
            'strict'    => false,
        ],

        'pgsql' => [
            'driver'   => 'pgsql',
            'host'     => env('DB_HOST', 'localhost'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
```

```
            'password' => env('DB_PASSWORD', ''),
            'charset'  => 'utf8',
            'prefix'   => '',
            'schema'   => 'public',
        ],

        'sqlsrv' => [
            'driver'   => 'sqlsrv',
            'host'     => env('DB_HOST', 'localhost'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'prefix'   => '',
        ],

    ],

    /*
    |--------------------------------------------------------------------------
    | Migration Repository Table
    |--------------------------------------------------------------------------
    |
    | This table keeps track of all the migrations that have already run for
    | your application. Using this information, we can determine which of
    | the migrations on disk haven't actually been run in the database.
    |
    */

    'migrations' => 'migrations',

    /*
    |--------------------------------------------------------------------------
    | Redis Databases
    |--------------------------------------------------------------------------
    |
    | Redis is an open source, fast, and advanced key-value store that also
    | provides a richer set of commands than a typical key-value systems
    | such as APC or Memcached. Laravel makes it easy to dig right in.
    |
    */

    'redis' => [
```

```
        'cluster' => false,

        'default' => [
            'host'     => '127.0.0.1',
            'port'     => 6379,
            'database' => 0,
        ],

    ],

];
```

Try to read the comments to understand how to use this file. The most two important settings are:

1. **default**: You can set the type of database you would like to use here. By default, it is mysql. If you want to use a different database, you can set it to: pgsql, sqlite, sqlsrv.
2. **connections**: Fill your database authentication credentials here. The **env() function** is used to retrieve configuration variables **(DB_HOST**, **DB_DATABASE**, **DB_USERNAME**, **DB_PASSWORD)** from **.env file**. If it can't find any variables, it will use the value of the function's second parameter **(localhost**, **forge)**.

If you use Homestead, Laravel has created a **homestead database** for you already. If you don't use Homestead, you will need to create a database manually.

Laravel uses PHP Data Objects (PDO). When we execute a Laravel SQL query, rows are returned in a form of a stdClass object. For instance, we may access our data using:

```
$user->name
```

You can easily change the fetch style to return result in array format by editing this line:

```
'fetch' => PDO::FETCH_CLASS,
```

update to:

```
'fetch' => PDO:FETCH_ASSOC,
```

I don't recommend changing this option at all, to be honest.

Create a database

> **Note**: You need a basic understanding of SQL to develop Laravel applications. At least, you should know how to read, update, modify and delete a database and its tables. If you don't know anything about database, a good place to learn is: http://www.w3schools.com/sql

To develop multiple applications, we will need multiple databases. In this section, we will learn how to create a database.

Default database information

Laravel has created a **homestead** database for us. To connect to **MySQL** or **Postgres** database, you can use these settings:

host: 127.0.0.1

database: homestead

username: homestead

password: secret

port: 33060 (MySQL) or 54320 (Postgres)

Create a database using the CLI

You can easily create a new database via the command line. First, **vagrant ssh** to your Homestead. Use this command to connect to **MySQL**:

```
mysql -uhomestead -p
```

When it asks for the password, use **secret**.

To create a new database, use this command:

```
CREATE DATABASE your_database_name;
```

Feel free to change **your_database_name** to your liking.

To see all databases, run this command:

```
show databases
```

Finally, you may leave MySQL using this command:

```
exit
```

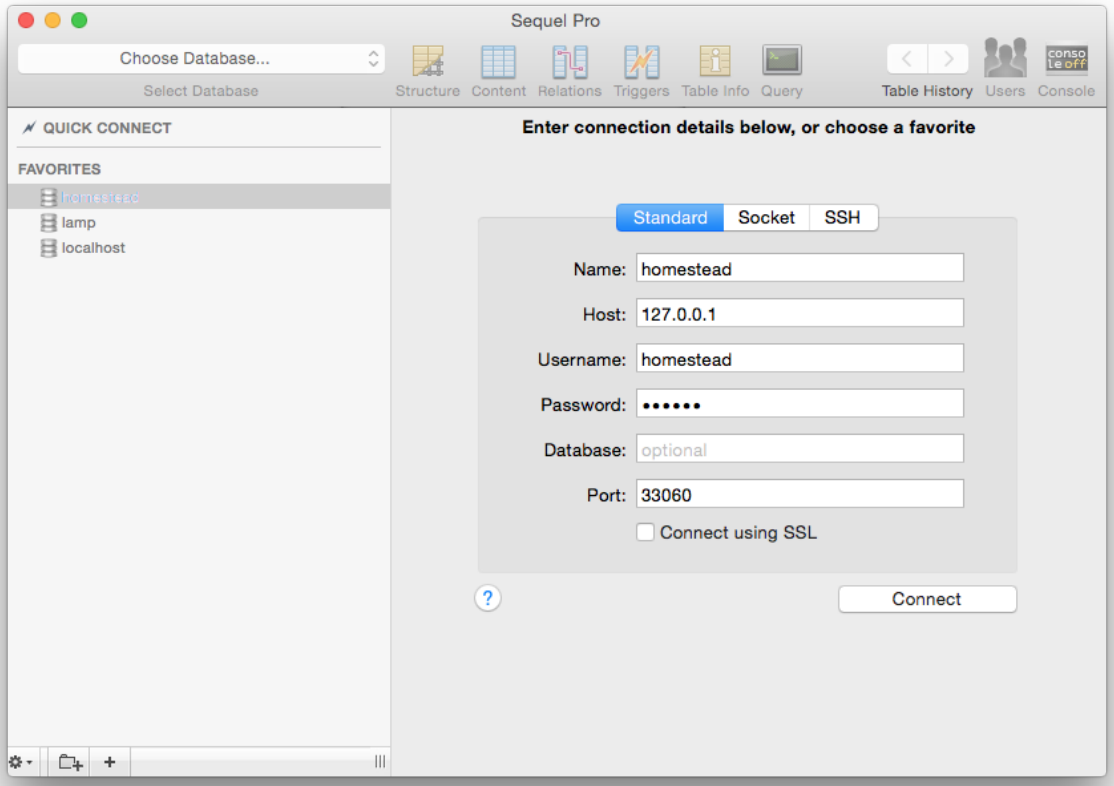
Even though we can easily create a new database via the **CLI**, we should use a **Graphical User Interface (GUI)** to manage databases easily.

Create a database on Mac

On Mac, the most popular GUI to manage databases is **Sequel Pro**. It's free, fast and very easy to use. You can download it here:

www.sequelpro.com

After that, you can connect to MySQL or Postgres database using database credentials in the **Default database information** section.



Connect to the databases using Sequel Pro

Once connected, you can easily create a new database by clicking **Choose Database**... and then **Add Database**.

Alternatively, you may use Navicat.

Create a database on Windows

On Windows, three popular GUIs for managing databases are:

SQLYog (Free)

https://www.webyog.com/product/sqlyog

SQLYog has a free open-source version. You can download it here:

https://github.com/webyog/sqlyog-community

Click the **Download SQLyog Community Version** to download.

HeidiSQL (Free)

http://www.heidisql.com

Navicat

http://www.navicat.com

Feel free to choose to use any GUI that you like. After that, you can connect to MySQL or Postgres database using database credentials in the **Default database information** section.

Using Migrations

One of the best features of Laravel is **Migrations**.

Whether you're working with a team or alone, you may need to find a way to keep track your database schema. **Laravel Migrations** is the right way to go.

Laravel uses migration files to know what we change in our database. The great thing is, you can easily revert or apply changes to your applications by just running a command. For example, we can use this command to reset the database:

```
php artisan migrate:reset
```

It's easy?

This feature is very useful. You should always use Migrations to build your application's database schema.

You can find Migrations documentation at:

http://laravel.com/docs/master/migrations

Meet Laravel Artisan

Artisan is **Laravel's CLI** (Command Line Interface). We often use **Artisan commands** to develop our Laravel applications. For instance, we use Artisan commands to generate migration files, seed our database, see the application namespace, etc.

You've used **Artisan** before! In the last chapter, we've used **Artisan** to generate controllers:

```
php artisan make:controller PagesController
```

Artisan official docs:

http://laravel.com/docs/master/artisan

To see a list of available Artisan commands, go to your application's root, run:

```
php artisan list
```

Create a new migration file

Now, let's try to generate a new migration file!

We're going to create a **tickets table**. Go to your **application root** (~/Code/Laravel), execute this command:

```
php artisan make:migration create_tickets_table
```

You'll see:

```
vagrant@homestead:~/Code/Laravel$ php artisan make:migration create_tickets_table
Created Migration: 2015_06_15_150120_create_tickets_table
```

Laravel will create a new migration template and place it in your **database/migrations** directory.

The name of the new migration template is: **create_tickets_table**. You can name it whatever you want. Check the migrations directory, you'll find a file look like this:

2015_06_15_150120_create_tickets_table.php

You may notice that Laravel put a **timestamp** before the name of the file, it helps to determine the order of the migrations.

Open the file:

```php
<?php

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

class CreateTicketsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        //
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        //
    }
}
```

Basically, a migration is just a standard class. There are two important methods:

1. **up** method: you use this method to add new tables, column to the database.
2. **down** method: well, you might have guessed already, this method is used to reverse what you've created.

It's time to create our tickets table by filling the up method!

But... WAIT!

We have a faster way to create the tickets table: using **–create** option.

Delete the migration file that you've just created. Once deleted, run this command:

```
php artisan make:migration create_tickets_table --create=tickets
```

By using the **–create** option, Laravel automatically generates the codes to create the tickets table for you.

```
public function up()
{
    Schema::create('tickets', function (Blueprint $table) {
        $table->increments('id');
        $table->timestamps();
    });
}
```

Understand Schema to write migrations

Schema is a class that we can use to define and manipulate tables. We use **Schema::create** method to create the **tickets table**:

```
    Schema::create('tickets', function (Blueprint $table) {
```

Schema::create method has two parameters. The first one is the **name of the table**.

The second one is a **Closure**. The Closure has one parameter: **$table**. You can name the parameter whatever you like.

We use the $table parameter to create **database columns**, such as id, name, date, etc.

```
$table->increments('id');
$table->timestamps();
```

increments('id') command is used to create **id column** and defines it to be an auto-increment primary key field in the **tickets** table.

timestamps is a special method of Laravel. It creates **updated_at** and **created_at** column. Laravel uses these columns to know when a row is **created** or **changed**.

To see a full list of Schema methods and column type, check out the **official docs**:

http://laravel.com/docs/master/migrations

You don't need to remember them all. We will learn some of them by creating some migrations in this book.

Our tickets will have these columns:

- id
- title
- content
- slug: URL friendly version of the ticket title
- status: current status of the ticket (answered or pending)
- user_id: who created the ticket

Here's how we can write our first migrations:

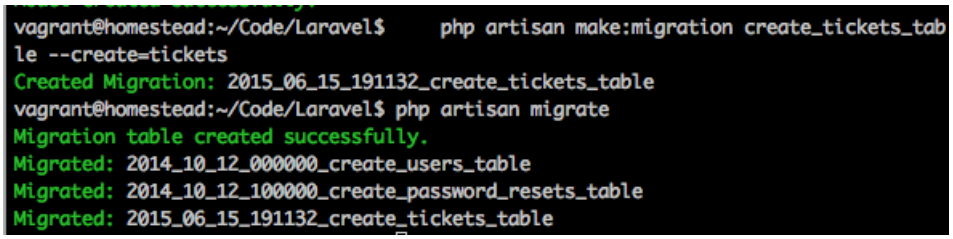
create_tickets_table.php

```
<?php

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

class CreateTicketsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('tickets', function (Blueprint $table) {
            $table->increments('id');
            $table->string('title', 255);
            $table->text('content');
            $table->string('slug')->nullable();
            $table->tinyInteger('status')->default(1);
            $table->integer('user_id');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
```

```
    {
        Schema::drop('tickets');
    }
}
```

Finally, run this command to create the tickets table and its columns:

```
php artisan migrate
```

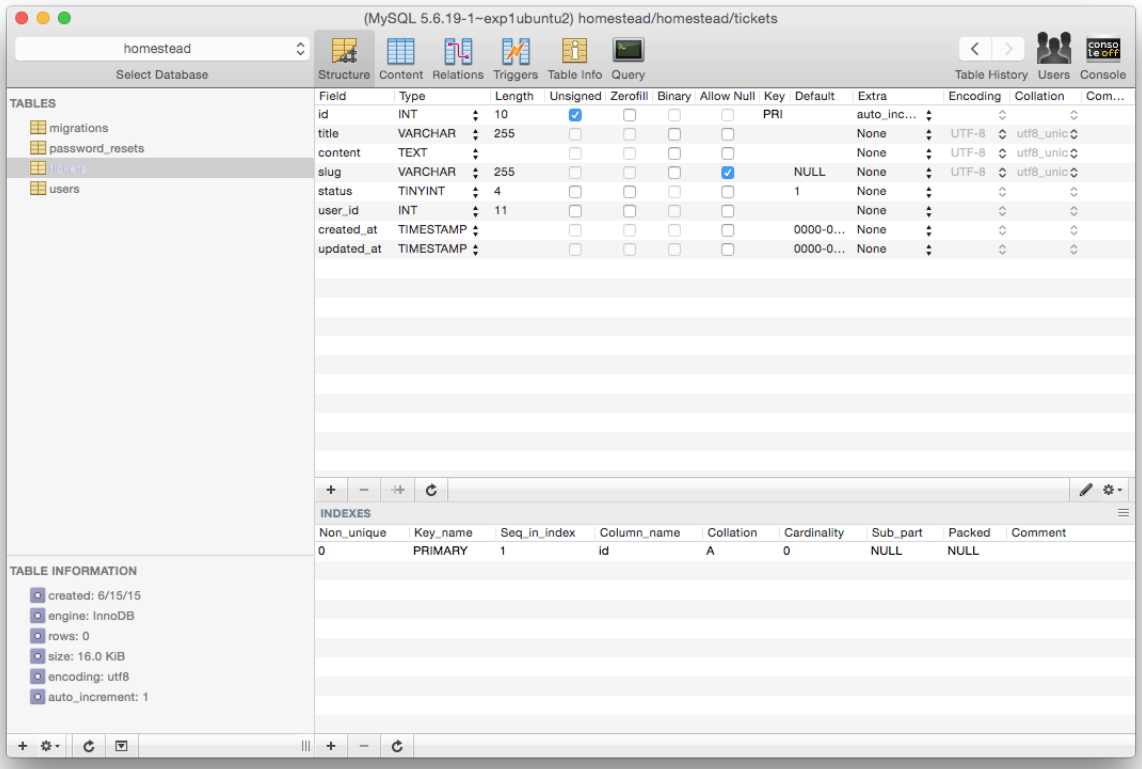


Use php artisan migrate

The first time you run this command, Laravel will create a **migration table** to keep track of what migrations you've created.

By default, Laravel also creates **create_users_table** migration and **create_password_resets** migration for us. These migrations will create **users** and **password_resets** tables. If you want to implement the default authentication, leave the files there. Otherwise, you can just delete them, or run **php artisan fresh** command to completely remove the default authentication feature.

Well, I guess it worked! It looks like the tables have been created. Let's check the **homestead** database:



tickets table and the columns

> **Note**: I use the **homestead** database. If you like to use another one, feel free to change it using the **.env** file.

Well done! You've just created a new **tickets table** to store our data.

Create a new Eloquent model

Laravel has a very nice feature: **Eloquent ORM**.

Eloquent provides a simple way to use ActiveRecord pattern when working with databases. By using this technique, we can wrap our database into objects.

What does it mean?

In **Object Oriented Programming (OOP)**, we usually create multiple objects. Objects can be anything that has properties and actions. For example, a mobile phone can be an object. Each model of a phone has its own blueprint. You can buy a new case for your phone, or you can change its

home screen. But no matter you customize it, it's still based on the blueprint that was created by the manufacturer.

We call that blueprint: **model**. Basically, model's just a class. Each model has its own variables (features of each mobile phone) and methods (actions that you take to customize the phone).

Model is known as the **M** in the **MVC** system (Model-View-Controller).

Now let's get back to our **tickets table**. If we can turn the **tickets** table to be a model, we can then easily access and manage it. Eloquent helps us to do the magic.

We may use Eloquent ORM to create, edit, manipulate, deletes our tickets without writing a single line of SQL!

To get started, let's create our first **Ticket** model by running this Artisan command:

```
php artisan make:model Ticket
```

> **Note**: a model name should be singular, and a table name should be plural.

Yes! It's that simple! You've created a model!

You can find the **Ticket model (Ticket.php)** in the **app** directory.

Here is a new tip. You can also generate the tickets migration at the same time by adding the **-m** option:

```
php artisan make:model Ticket -m
```

Cool?

Ok, let's open our new Ticket model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Ticket extends Model
{
    //
}
```

As you may notice, the Ticket model is just a **PHP** class that extends the **Model** class. Now we can use this file to tell Laravel about its **relationships**. For instance, each ticket is created by a user, we can tell tickets belongs to users by writing like this:

```
public function user()
{
    return $this->belongsTo('App\User');
}
```

We also can be able to use this model to access any tickets' data, such as: title, content, etc.

```
public function getTitle()
{
    return $this->title
}
```

Eloquent is clever. It automatically finds and connects our models with our database tables if you name them correctly (Ticket model and tickets table, in this case).

For some reasons, if you want to use a different name, you can let Eloquent know that by defining a table property like this:

```
protected $table = 'yourCustomTableName';
```

Read more about Eloquent here:

http://laravel.com/docs/master/eloquent

Once we have the Ticket model, we can build a form to let the users create a new ticket!

Create a page to submit tickets

Now as we have the Ticket model, let's write the code for creating new tickets.

To create a new ticket, we will need to use **Controller action** (also known as Controller method) and **view** to display the new ticket form.

Create a view to display the submit ticket form

Go to our **views** directory, create a new directory called **tickets**.

Because we will have many ticket views (such as: create ticket view, edit ticket view, delete ticket view, etc.), we should store all the views in the **tickets** directory.

Next, create a new Blade template called **create.blade.php**

```
@extends('master')
@section('title', 'Contact')

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="well well bs-component">
            <form class="form-horizontal">
                <fieldset>
                    <legend>Submit a new ticket</legend>
                    <div class="form-group">
                        <label for="title" class="col-lg-2 control-label">Title<\
/label>
                        <div class="col-lg-10">
                            <input type="text" class="form-control" id="title" p\
laceholder="Title">
                        </div>
                    </div>
                    <div class="form-group">
                        <label for="content" class="col-lg-2 control-label">Cont\
ent</label>
                        <div class="col-lg-10">
                            <textarea class="form-control" rows="3" id="content"\
></textarea>
                            <span class="help-block">Feel free to ask us any que\
stion.</span>
                        </div>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button class="btn btn-default">Cancel</button>
                            <button type="submit" class="btn btn-primary">Submit\
</button>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
    </div>
@endsection
```

Unfortunately, we can't see this view yet, we have to use a Controller action to display it. Open **PagesController.php**, edit:

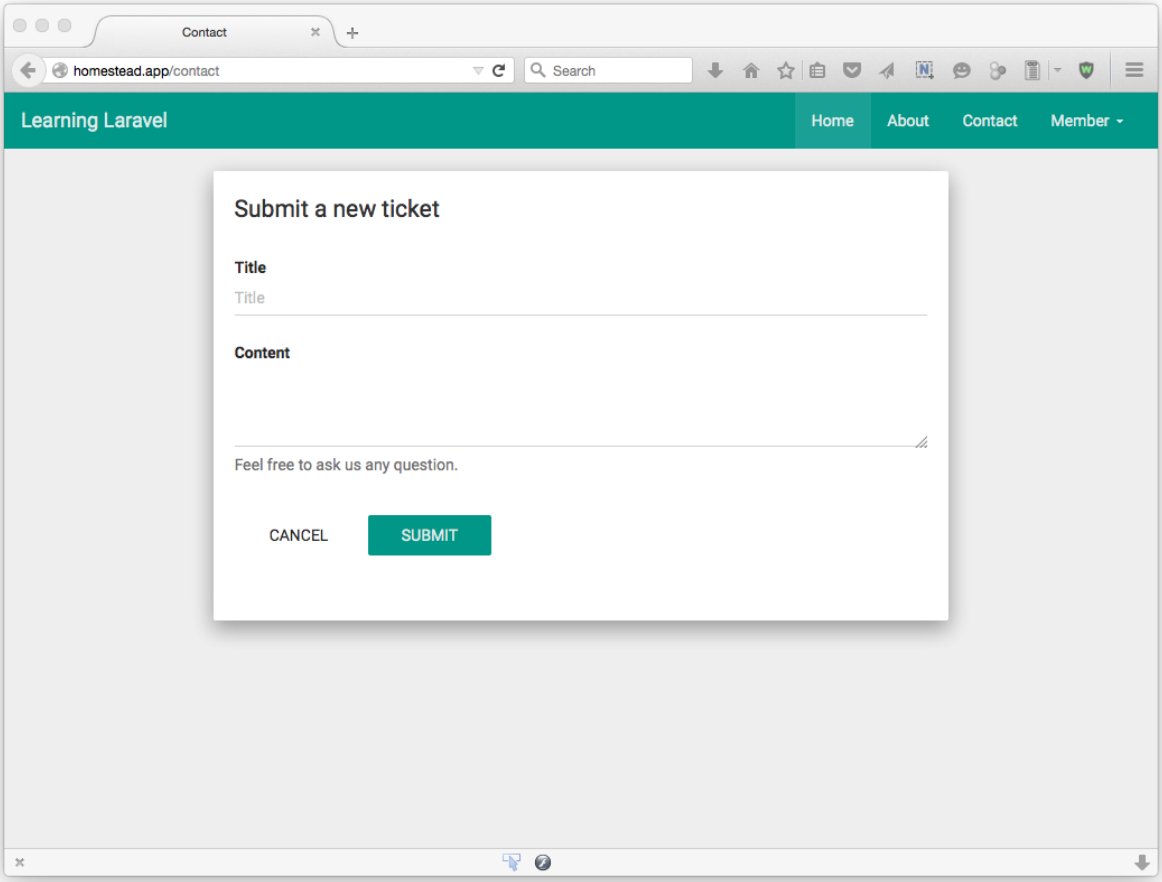
```
public function contact()
{
    return view('contact');
}
```

to

```
public function contact()
{
    return view('tickets.create');
}
```

Instead of displaying the contact view, we redirect users to **tickets.create** view.

After saving these changes, we should see a new responsive **submit ticket** form when visiting the **contact** page:



submit ticket form

Create a new controller for the tickets

Even though we can use **PagesController** to manage all the pages, we should create **TicketsController** to manage our tickets.

TicketsController will be responsible for creating, editing and deleting tickets. It will help to organize and maintain our application much easier.

You have known how to create a controller, let's create the **TicketsController** by running this command:

```
php artisan make:controller TicketsController
```

By default, Laravel has created some RESTful actions (create, edit, update, etc.) for us. We will use some actions, so you don't need to delete them all.

Update the **create** action as follows:

```
public function create()
{
    return view('tickets.create');
}
```

And don't forget to update the **routes.php** file:

```
Route::get('/contact', 'TicketsController@create');
```

Great! You now have the **TicketsController**. Pretty simple, right?

Introducing HTTP Requests

In previous versions of Laravel, developers usually place validation anywhere they want. That is not a good practice.

Luckily, Laravel 5 has a new feature called **Requests** (aka HTTP Requests or Form Requests). When users send a request (submit a ticket, for example), we can use the new **Request** class to define some rules and validate the request. If the validator passes, then everything will be executed as normal. Otherwise, the user will be automatically redirected back to where they are.

As you can see, it's really convenient for us to validate our application's forms.

We will use **Request** to validate the **create ticket** form.

To create a new **Request**, simply run this Artisan command:

```
php artisan make:request TicketFormRequest
```

A new **TicketFormRequest** will be generated! You can find it in the **app/Http/Requests** directory.

Open the file, you can see that there are two methods: **authorize** and **rules**.

authorize() method

```
public function authorize()
{
    return false;
}
```

By default, it returns **false**. That means no one can be able to perform the request. To be able to submit the tickets, we have to turn it to **true**

```
public function authorize()
{
    return true;
}
```

rules() method

```
public function rules()
{
    return [
        //
    ];
}
```

We use this method to define our validation rules.

Currently, our **create ticket** form has two fields: title and content, we can set the following rules:

```
public function rules()
{
    return [
        'title' => 'required|min:3',
        'content'=> 'required|min:10',
    ];
}
```

required|min:3 validation rule means that the users must fill the title field, and the title should have a minimum three character length.

There are many validation rules, you can see a list of available rules at:

http://laravel.com/docs/master/validation#available-validation-rules

Install Laravel Collective packages

Since Laravel 5, some Laravel components have been removed from the core framework. If you've used older versions of Laravel before, you may love these features:

- HTML: HTML helpers for creating common HTML and form elements
- Annotations: route and events annotations.
- Remote: a simple way to SSH into remote servers and run commands.

Fortunately, bringing all these features back is very easy. You just need to install LaravelCollective package!

http://laravelcollective.com

Don't know how to install the package? Let me show you.

Install a package using Composer

First, you need to open your **composer.json** file, which is placed in your application root.

```
{
    "name": "laravel/laravel",
    "description": "The Laravel Framework.",
    "keywords": ["framework", "laravel"],
    "license": "MIT",
    "type": "project",
    "require": {
        "php": ">=5.5.9",
        "laravel/framework": "5.1.*"
    },
    "require-dev": {
        "fzaninotto/faker": "~1.4",
        "mockery/mockery": "0.9.*",
        "phpunit/phpunit": "~4.0",
        "phpspec/phpspec": "~2.1"
    },
    "autoload": {
        "classmap": [
            "database"
        ],
        "psr-4": {
            "App\\": "app/"
```

```
        }
    },
    "autoload-dev": {
        "classmap": [
            "tests/TestCase.php"
        ]
    },
    "scripts": {
        "post-install-cmd": [
            "php artisan clear-compiled",
            "php artisan optimize"
        ],
        "post-update-cmd": [
            "php artisan clear-compiled",
            "php artisan optimize"
        ],
        "post-root-package-install": [
            "php -r \"copy('.env.example', '.env');\""
        ],
        "post-create-project-cmd": [
            "php artisan key:generate"
        ]
    },
    "config": {
        "preferred-install": "dist"
    },
    "minimum-stability": "dev",
    "prefer-stable": true
}
```

This is a **JSON (Javascript Object Notation)** file. We use **JSON** to store and exchange data. JSON is very easy to read. If you can read HTML or XML, I'm sure that you can read JSON.

If you can't read it, learn more about JSON here:

www.w3schools.com/json

In this section, we will add the **HTML** package to our app. The instructions can be found here:

http://laravelcollective.com/docs/5.1/html

To install a Laravel package using Composer, you just need to add the following code:

find:

```
"require": {
    "php": ">=5.5.9",
    "laravel/framework": "5.1.*"
},
```

add:

```
"require": {
    "php": ">=5.5.9",
    "laravel/framework": "5.1.*",
    "laravelcollective/html": "5.1.*"
},
```

> **Note:** If you're using Laravel 5.2 or newer, the version could be different. For example, if you're using Laravel 5.2, the code should be "laravelcollective/html": "5.2.*"

Save the file and run this command at your application root:

```
composer update
```

Done! You've just installed LaravelCollective/HTML package!

Create a service provider and aliases

After installing the HTML package via Composer. You need to follow some extra steps to let Laravel know where to find the package and use it.

To use the package, you have to add a **service providers** to the **providers array** of **config/app.php**.

Find:

```
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    Illuminate\Foundation\Providers\ArtisanServiceProvider::class,
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
    Illuminate\Bus\BusServiceProvider::class,
    Illuminate\Cache\CacheServiceProvider::class,
    Illuminate\Foundation\Providers\ConsoleSupportServiceProvider::class,
```

```
    Illuminate\Routing\ControllerServiceProvider::class,
    Illuminate\Cookie\CookieServiceProvider::class,
    Illuminate\Database\DatabaseServiceProvider::class,
    Illuminate\Encryption\EncryptionServiceProvider::class,
    Illuminate\Filesystem\FilesystemServiceProvider::class,
    Illuminate\Foundation\Providers\FoundationServiceProvider::class,
    Illuminate\Hashing\HashServiceProvider::class,
    Illuminate\Mail\MailServiceProvider::class,
    Illuminate\Pagination\PaginationServiceProvider::class,
    Illuminate\Pipeline\PipelineServiceProvider::class,
    Illuminate\Queue\QueueServiceProvider::class,
    Illuminate\Redis\RedisServiceProvider::class,
    Illuminate\Auth\Passwords\PasswordResetServiceProvider::class,
    Illuminate\Session\SessionServiceProvider::class,
    Illuminate\Translation\TranslationServiceProvider::class,
    Illuminate\Validation\ValidationServiceProvider::class,
    Illuminate\View\ViewServiceProvider::class,

    /*
     * Application Service Providers...
     */
    App\Providers\AppServiceProvider::class,
    App\Providers\EventServiceProvider::class,
    App\Providers\RouteServiceProvider::class,

],
```

Add the following line to the $provider array:

```
Collective\Html\HtmlServiceProvider::class,
```

You should have:

```
    App\Providers\RouteServiceProvider::class,
    Collective\Html\HtmlServiceProvider::class,
],
```

Then find the aliases array:

```
'aliases' => [

    'App'       => Illuminate\Support\Facades\App::class,
    'Artisan'   => Illuminate\Support\Facades\Artisan::class,
    'Auth'      => Illuminate\Support\Facades\Auth::class,
    'Blade'     => Illuminate\Support\Facades\Blade::class,
    'Bus'       => Illuminate\Support\Facades\Bus::class,
    'Cache'     => Illuminate\Support\Facades\Cache::class,
    'Config'    => Illuminate\Support\Facades\Config::class,
    'Cookie'    => Illuminate\Support\Facades\Cookie::class,
    'Crypt'     => Illuminate\Support\Facades\Crypt::class,
    'DB'        => Illuminate\Support\Facades\DB::class,
    'Eloquent'  => Illuminate\Database\Eloquent\Model::class,
    'Event'     => Illuminate\Support\Facades\Event::class,
    'File'      => Illuminate\Support\Facades\File::class,
    'Hash'      => Illuminate\Support\Facades\Hash::class,
    'Input'     => Illuminate\Support\Facades\Input::class,
    'Inspiring' => Illuminate\Foundation\Inspiring::class,
    'Lang'      => Illuminate\Support\Facades\Lang::class,
    'Log'       => Illuminate\Support\Facades\Log::class,
    'Mail'      => Illuminate\Support\Facades\Mail::class,
    'Password'  => Illuminate\Support\Facades\Password::class,
    'Queue'     => Illuminate\Support\Facades\Queue::class,
    'Redirect'  => Illuminate\Support\Facades\Redirect::class,
    'Redis'     => Illuminate\Support\Facades\Redis::class,
    'Request'   => Illuminate\Support\Facades\Request::class,
    'Response'  => Illuminate\Support\Facades\Response::class,
    'Route'     => Illuminate\Support\Facades\Route::class,
    'Schema'    => Illuminate\Support\Facades\Schema::class,
    'Session'   => Illuminate\Support\Facades\Session::class,
    'Storage'   => Illuminate\Support\Facades\Storage::class,
    'URL'       => Illuminate\Support\Facades\URL::class,
    'Validator' => Illuminate\Support\Facades\Validator::class,
    'View'      => Illuminate\Support\Facades\View::class,

],
```

add these two aliases to the aliases array:

```
  'Form' => Collective\Html\FormFacade::class,
  'Html' => Collective\Html\HtmlFacade::class,
```

You should have:

```
    'View'      => Illuminate\Support\Facades\View::class,
    'Form' => Collective\Html\FormFacade::class,
    'Html' => Collective\Html\HtmlFacade::class,
],
```

Now you can use the **LaravelCollective/HTML** package!

How to use HTML package

The **HTML** package helps us to build forms easier and faster. Let's see an example:

Normal HTML code:

```
<form action="contact">
    <label>First name:</label>
    <input type="text" name="firstname" value="Enter your first name">
    <br />
    <label>Last name:</label>
    <input type="text" name="lastname" value="Enter your last name">
    <br />
    <input type="submit" value="Submit">
</form>
```

Using HTML package:

```
{!! Form::open(array ('url' => 'contact')) !!}
{!! Form::label('First name') !!}
{!! Form::text('firstname', 'Enter your first name') !!}
<br />
{!! Form::label('Last name') !!}
{!! Form::text('lastname', 'Enter your last name') !!}
<br />
{!! Form::submit() !!}
{!! Form::close() !!}
```

As you see, we use **Form::open()** to create our opening form tag and **Form::close()** to close the form.

Text fields and labels can be generated using **Form::text** and **Form::label** method.

You can learn more about how to use HTML package by reading Laravel 4 official docs:

http://laravel.com/docs/4.2/html

If you don't like to take advantage of the HTML package to build your forms, you don't have to use it. I just want you to understand its syntax because many Laravel developers are using it these days. It would be better if you know both methods.

A note about Laravel 5.2's changes

> **Important**: If you're using Laravel 5.2 or newer, please read this. **Please note that you should skip this section if you don't get any errors. After Laravel 5.2.30, all routes go into web middleware by default, you may skip this section if you're using Laravel 5.2.30 or newer.**

Since Laravel 5.2, **/app/Http/Kernel.php** and **/app/Http/routes.php** have been changed. You could get some errors like:

```
Undefined variable: errors
```

Solution 1: use web middleware group

In Laravel 5.2, there is a new **web middleware group**.

```
/*
|--------------------------------------------------------------------------
| Application Routes
|--------------------------------------------------------------------------
|
| This route group applies the "web" middleware group to every route
| it contains. The "web" middleware group is defined in your HTTP
| kernel and includes session state, CSRF protection, and more.
|
*/
Route::group(['middleware' => ['web']], function () {
    //
});
```

If you're using **Session** ($errors object, session('status'), etc.) or **CSRF protection**, be sure to put all the related routes inside the web middleware group. For example:

If you see:

```
Route::get('/contact', 'TicketsController@create');
Route::post('/contact', 'TicketsController@store');
```

Be sure to put them inside the web middleware group:

```
Route::group(['middleware' => ['web']], function () {
    Route::get('/contact', 'TicketsController@create');
    Route::post('/contact', 'TicketsController@store');
});
```

Solution 1: fix the $errors object's error

Because the **$errors** object is only available when you put the related route inside the web middleware group, you may get this error:

```
Undefined variable: errors
```

To fix this, find:

```
@foreach ($errors->all() as $error)
    // content
@endforeach
```

Then add an if statement to check if the **$errors** object is available:

```
@if (isset($errors) && $errors->any())
    @foreach ($errors->all() as $error)
        // content
    @endforeach
@endif
```

Best solution: update your Kernel.php file

If you don't want to put all your routes into the web middleware group, update your **Kernel.php** file as follows:

Find:

```
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
];
```

Change to:

```
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \App\Http\Middleware\VerifyCsrfToken::class,
];
```

Now everything should be working fine.

Submit the form data

Having learned about **Requests**, working with **Controllers** and **View** to build the **create ticket** form, now you're ready to process the submitted data.

if you click the submit button now, nothing happens. We need to use other **HTTP method** to submit the form.

Two commonly used HTTP methods for a client to communicate with a server are: **GET** and **POST**.

We've used **GET** to display the form:

```
Route::get('/contact', 'TicketsController@create');
```

But we won't use **GET** to submit data. **GET** requests should only be used to retrieve data.

We always use **POST** method to handle the form submissions endpoints. When we use **POST**, requests are never cached, parameters are not saved in users' browser history. Therefore, POST is safer than GET.

Let's open the **routes.php** file, add:

```
Route::post('/contact', 'TicketsController@store');
```

Good! Now when users make a **POST** request to the contact page, this route tells Laravel to execute the TicketsController's **store** action.

The store action is still empty. You can update it to display the form data:

TicketsController.php

```
public function store(TicketFormRequest $request)
{
    return $request->all();
}
```

We use the **TicketFormRequest** as **a parameter** of the **store action** here to tell Laravel that we want to apply validation to the store action.

Laravel requires us to **type-hint** the **IlluminateHttpRequest** class on our controller constructor to obtain an instance of the current **HTTP request**. Simply put, we need to add this line at the top of the **TicketsController.php** file:

```
use App\Http\Requests\TicketFormRequest;
```

find:

```
class TicketsController extends Controller
{
```

add above

```
use App\Http\Requests\TicketFormRequest;
class TicketsController extends Controller
{
```

One more step, you need to update the ticket form to send POST requests. Open **tickets/create.blade.php**.

find:

```
<form class="form-horizontal">
```

update to:

```
<form class="form-horizontal" method="post">
```

You also need to tell Laravel the name of the fields:

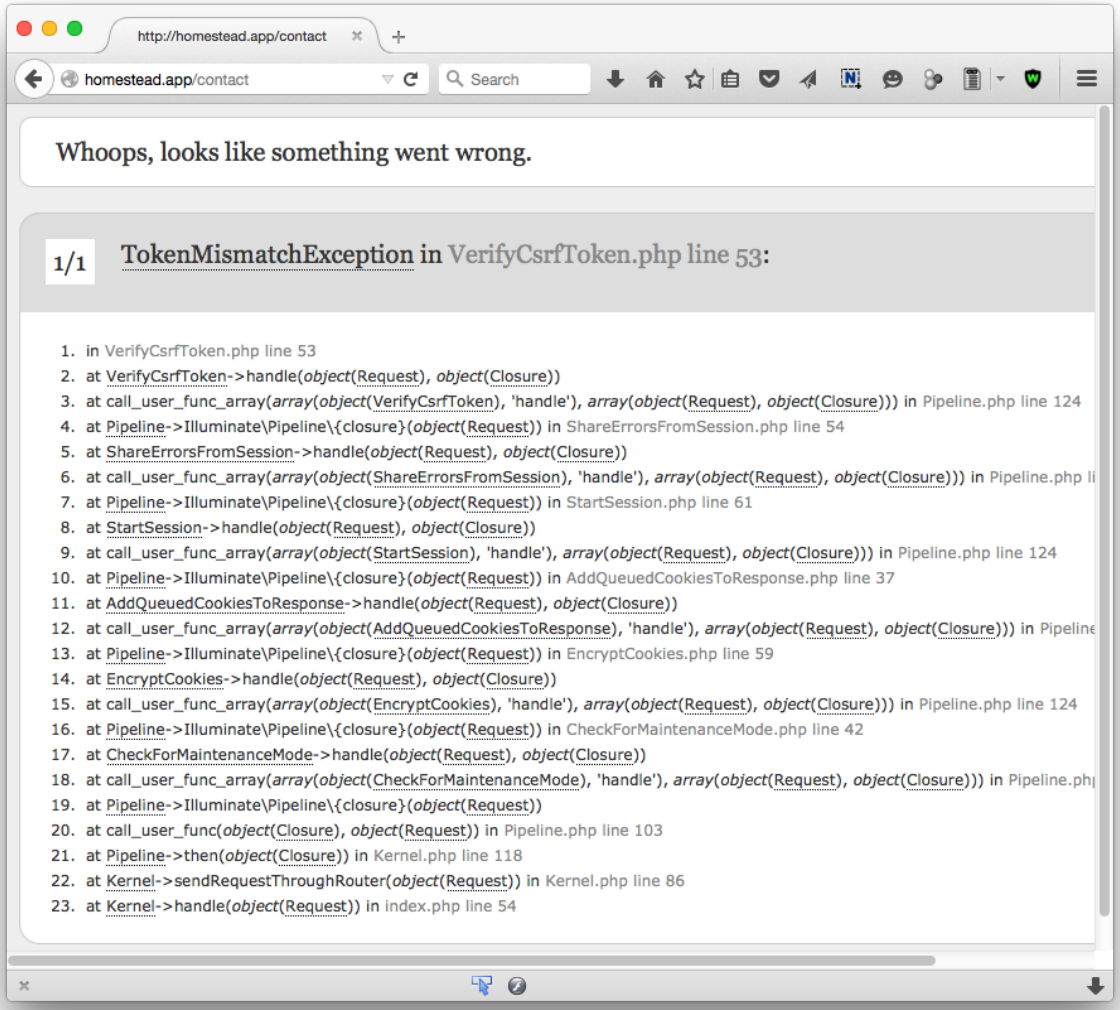
find:

```
<input type="text" class="form-control" id="title" placeholder="Title">
...
<textarea class="form-control" rows="3" id="content"></textarea>
```

update to:

```
<input type="text" class="form-control" id="title" placeholder="Title" name="tit\
le">
...
<textarea class="form-control" rows="3" id="content" name="content"></textarea>
```

Now, try to click the submit button!



Ticket form error

Oops! There is an error.

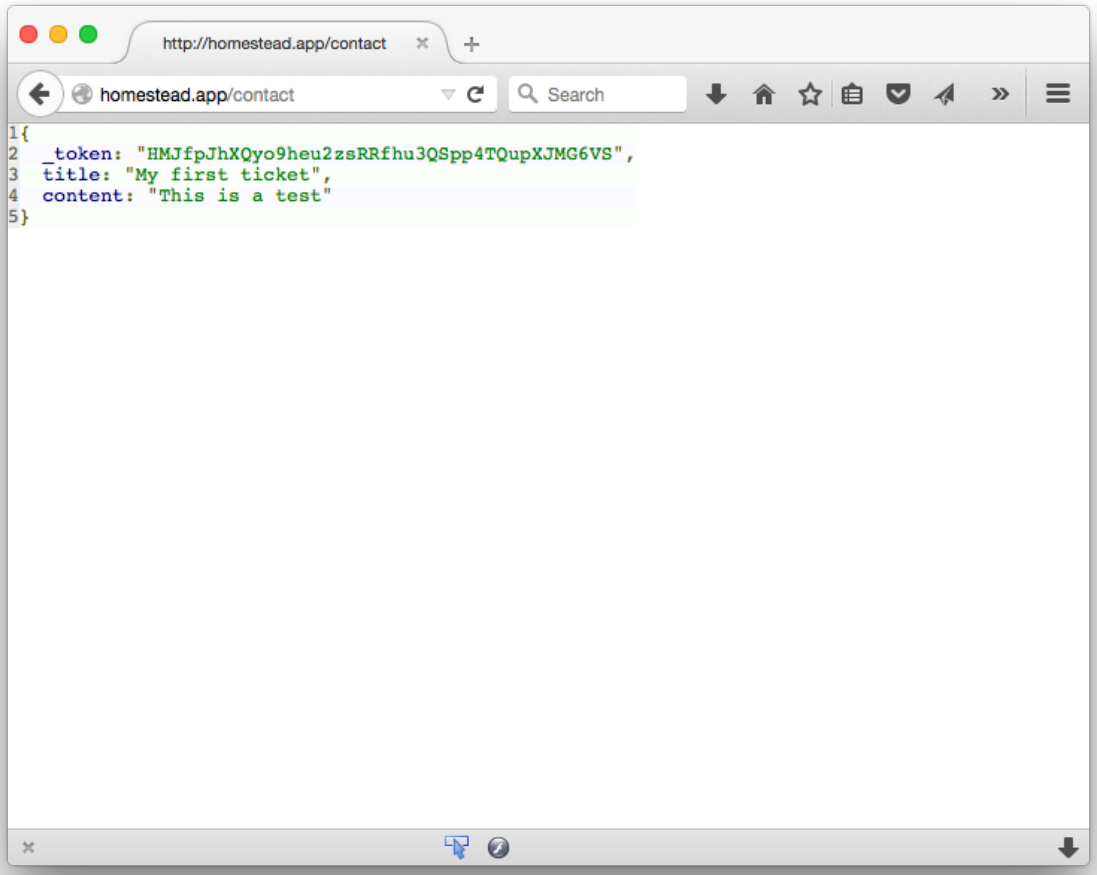
What is TokenMismatchException?

For security purposes, Laravel now requires a token to be sent when using the POST method. If you don't send any token, it will throw an error.

To fix this, you need to add a **hidden token field** below your form opening tag:

```
<form class="form-horizontal" method="post">
    <input type="hidden" name="_token" value="{!! csrf_token() !!}">
```

Good! Refresh your browser, fill the form and hit submit again, you'll see:



Display requests successfully

Yayyy! We can see the ticket data! Everything is working!

One last step is to display the errors when the users don't fill the form or the form is not valid.

Find:

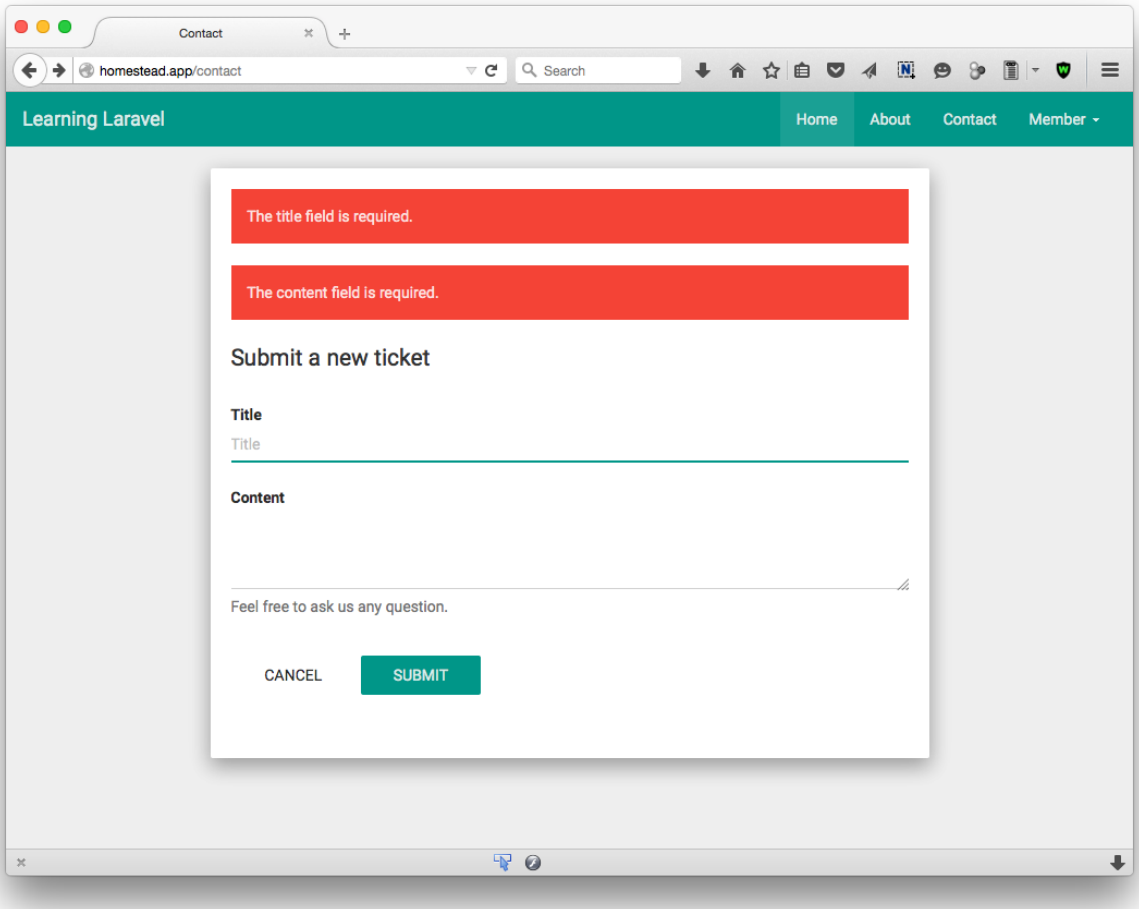
```
<form class="form-horizontal" method="post">
```

add below:

```
@foreach ($errors->all() as $error)
    <p class="alert alert-danger">{{ $error }}</p>
@endforeach
```

Basically, if the validator fails, Laravel will store all errors in the session. We can easily access the errors via **$errors** object.

Now, let's go back to the form, don't fill anything and hit the submit button:



Display error

Here is the new **tickets/create.blade.php** file:

```
@extends('master')
@section('title', 'Create a new ticket')

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="well well bs-component">

            <form class="form-horizontal" method="post">

                @foreach ($errors->all() as $error)
                    <p class="alert alert-danger">{{ $error }}</p>
                @endforeach

                <input type="hidden" name="_token" value="{!! csrf_token() !!}">

                <fieldset>
                    <legend>Submit a new ticket</legend>
                    <div class="form-group">
                        <label for="title" class="col-lg-2 control-label">Title<\
/label>
                        <div class="col-lg-10">
                            <input type="title" class="form-control" id="title" \
placeholder="Title" name="title">
                        </div>
                    </div>
                    <div class="form-group">
                        <label for="content" class="col-lg-2 control-label">Cont\
ent</label>
                        <div class="col-lg-10">
                            <textarea class="form-control" rows="3" id="content"\
 name="content"></textarea>
                            <span class="help-block">Feel free to ask us any que\
stion.</span>
                        </div>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button class="btn btn-default">Cancel</button>
                            <button type="submit" class="btn btn-primary">Submit\
</button>
                        </div>
```

```
                    </div>
                </fieldset>
            </form>
        </div>
    </div>
@endsection
```

Using .env file

In the next section, we will learn how to insert data into the database. Before working with databases, you should understand **.env** file first.

What is the .env file?

Our applications usually run in different environments. For example, we develop our apps on a local server, and deploy it on a production server. The database settings and server credentials of each environment might be different. Laravel 5 provides us an easy way to handle different configuration settings by simply editing the **.env** file.

The .env file helps us to load custom configurations variables without editing .htaccess files or virtual hosts, and keep our sensitive credentials more secure.

Learn more about .env file here:

https://github.com/vlucas/phpdotenv

How to edit it?

The .env file is very easy to configure. Let's open it:

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=o8UTlUkakeVkSUlFbGjSXaCcmAAkU0xB

DB_HOST=localhost
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync
```

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

As you see, the file is very clear. Let's try to edit a few settings.

Currently, you're using the default **homestead** database. If you've created a different database and you want to use it instead, edit this line:

```
DB_DATABASE=homestead
```

to

```
DB_DATABASE=yourCustomDatabaseName
```

If you don't want to display full error messages, turn the **APP_DEBUG** to false.

If you're using **sendgrid** to send emails, replace these lines with your **sendgrid credentials**:

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

For example:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.sendgrid.net
MAIL_PORT=587
MAIL_USERNAME=learninglaravel
MAIL_PASSWORD=secret
```

Now, let's move to the fun part!

Insert data into the database

In the previous section, you've learned how to receive and validate the users' requests.

Once you have the submitted form data, inserting the data into the database is pretty easy.

First, let's begin by putting this line at the top of your **TicketsController** file:

```
use App\Ticket;
```

Be sure to put it above the class name:

```
use App\Ticket;
class TicketsController extends Controller
{
```

This tells Laravel that you want to use your **Ticket model** in this class.

Now you can use Ticket model to store the form data. Update the store action:

```
public function store(TicketFormRequest $request)
{
    $slug = uniqid();
    $ticket = new Ticket(array(
        'title' => $request->get('title'),
        'content' => $request->get('content'),
        'slug' => $slug
    ));

    $ticket->save();

    return redirect('/contact')->with('status', 'Your ticket has been created! I\
ts unique id is: '.$slug);

}
```

Let's see the code line by line:

```
$slug = uniqid();
```

We use the uniqid() function to generate a unique ID based on the microtime. You may use **md5()** function to generate the slugs or create your custom slugs.

This is the ticket's unique ID.

```
    $ticket = new Ticket(array(
        'title' => $request->get('title'),
        'content' => $request->get('content'),
        'slug' => $slug
    ));
```

Next, we create a new **Ticket model** instance, set attributes on the model.

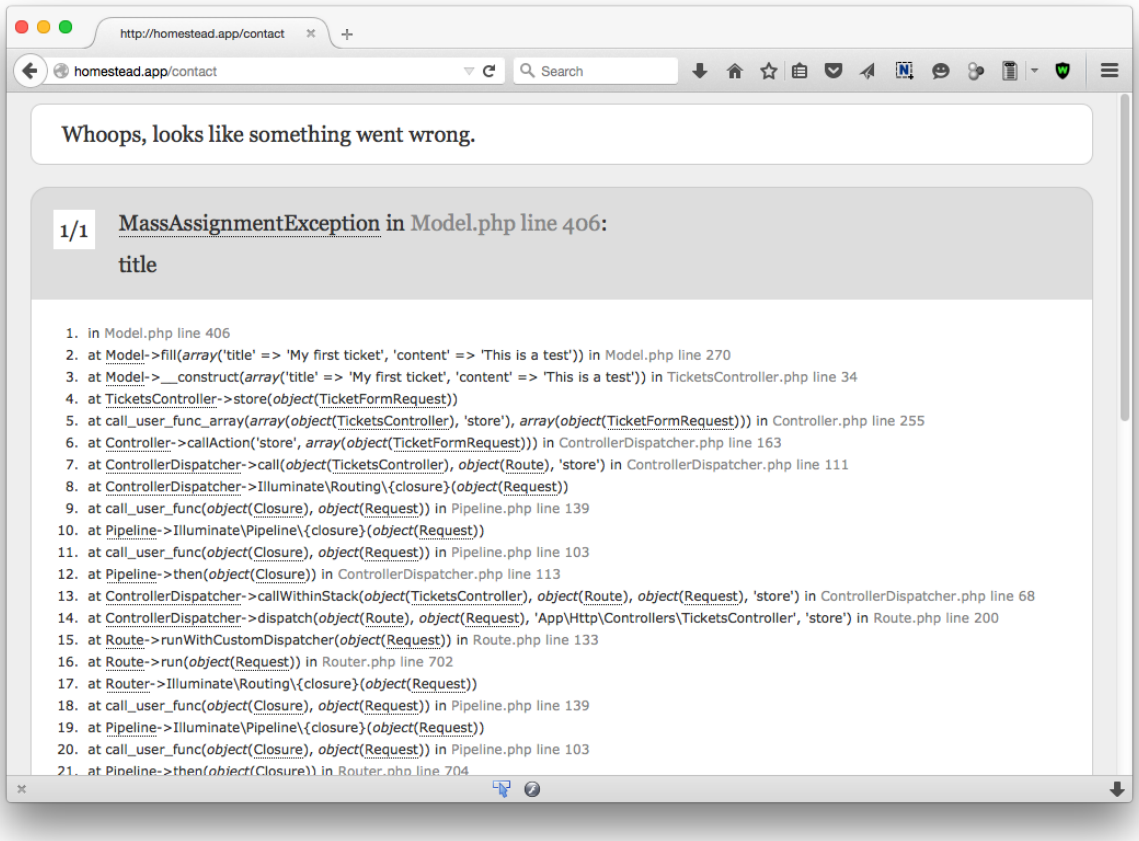
```
$ticket->save();
```

Then we call the **save** method to save the data to our database.

```
return redirect('/contact')->with('status', 'Your ticket has been created! Its u\
nique id is: '.$slug);
```

Once the ticket has been saved, we redirect users to the contact page with a **message**.

Finally, try to create a new ticket and submit it.



Display error

Oh no!

There is an error: "**MassAssignmentException**"

Don't worry, it's a Laravel feature that protect against **mass-assignment**.

What is mass-assignment?

According to the Laravel official docs:

"mass-assignment vulnerability occurs when user's pass unexpected HTTP parameters through a request, and then that parameter changes a column in your database you did not expect. For example, a malicious user might send an is_admin parameter through an HTTP request, which is then mapped onto your model's create method, allowing the user to escalate themselves to an administrator"

Read more about it here:

http://laravel.com/docs/master/eloquent#mass-assignment

To save the ticket, open the **Ticket model**. (Ticket.php file)

Then place the following contents into the Ticket Model:

```
class Ticket extends Model
{
    protected $fillable = ['title', 'content', 'slug', 'status', 'user_id'];
}
```

The **$fillable** property make the columns **mass assignable**.

Alternatively, you may use the **$guarded** property to make all attributes **mass assignable** except for your chosen attributes. For example, I use the **id** column here:

```
protected $guarded = ['id'];
```

> Note: You must use either $fillable or $guarded.

One more thing to do, we need to update the **tickets/create.blade.php** view to display the status message:

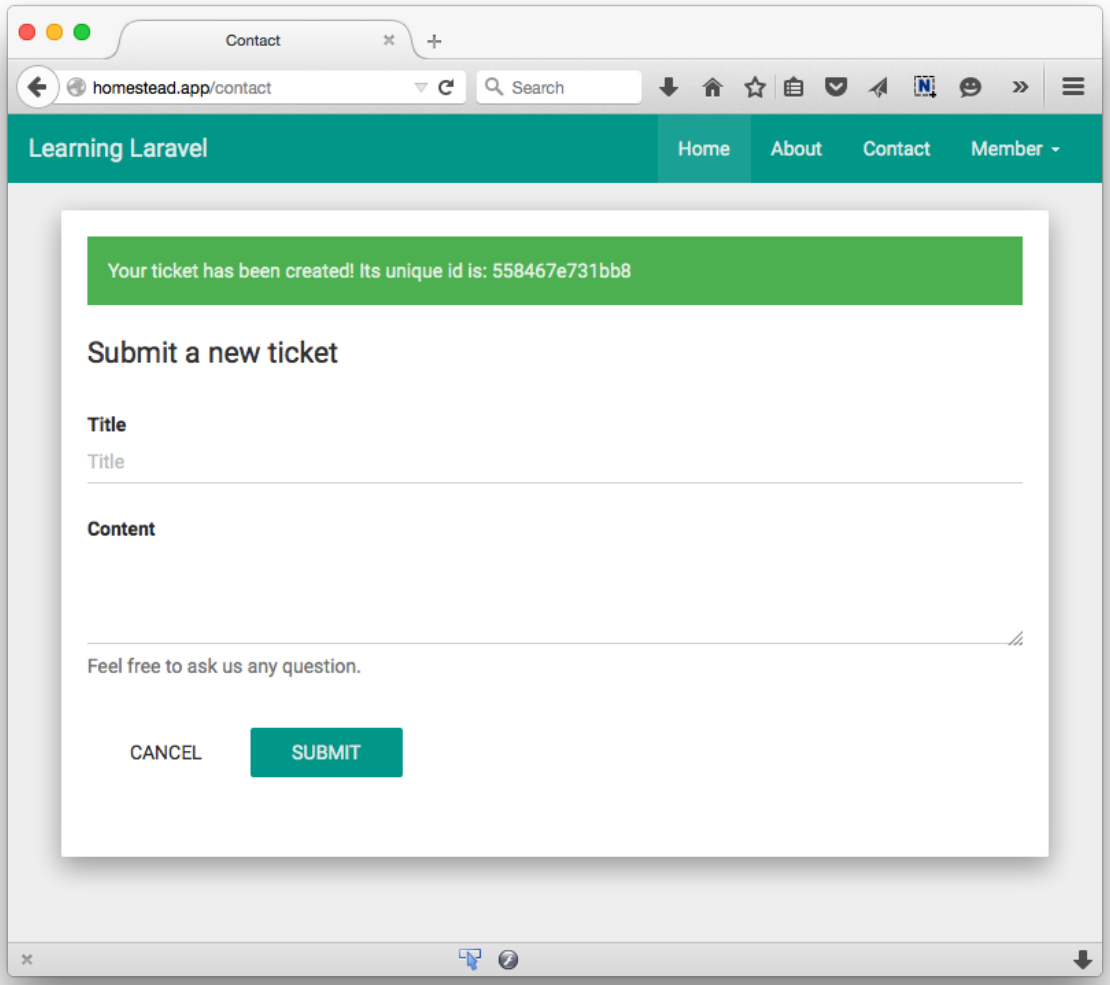
Find:

```
<form class="form-horizontal" method="post">

    @foreach ($errors->all() as $error)
        <p class="alert alert-danger">{{ $error }}</p>
    @endforeach
```

Add Below:

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

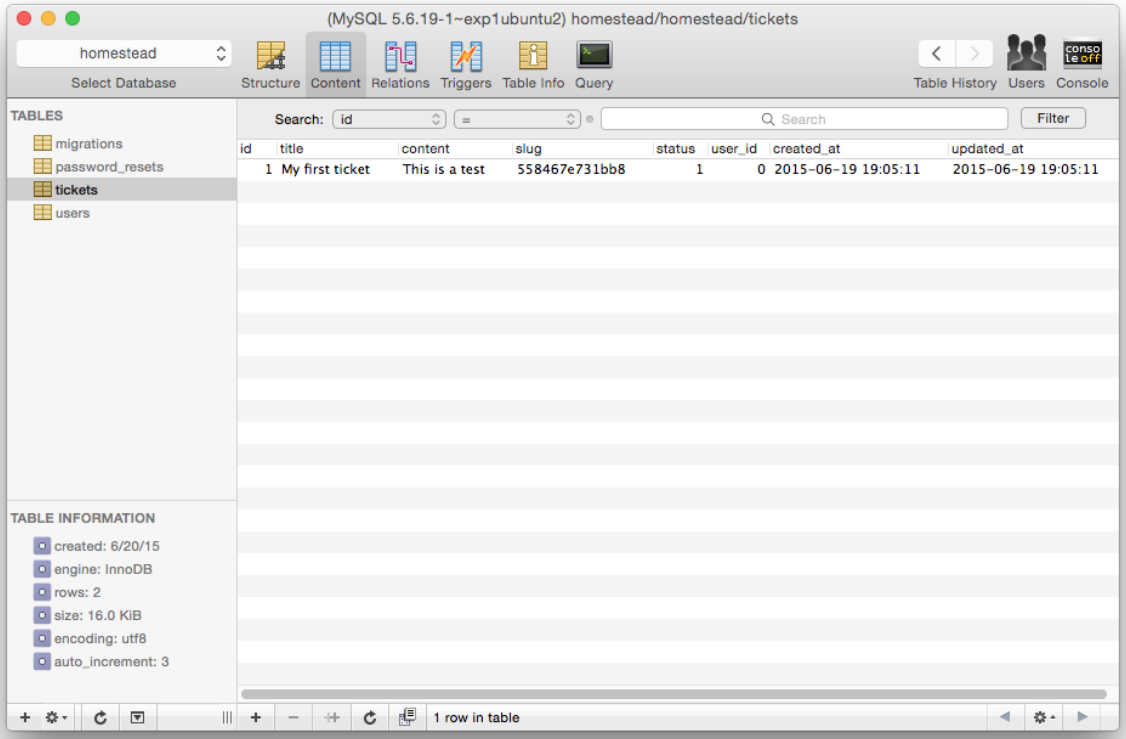
Good! Try to create a new ticket again.



Success message

Well done! You've just saved a new ticket to the database!

Be sure to check your application database, you should see some new records in the tickets table:



Success message

View all tickets

As you continue developing the app, you'll find that you need a way to display all tickets, so you can be able to view, modify or delete them easily.

The following are the steps to list all tickets:

First, we'll modify the **routes.php** file:

```
Route::get('/tickets', 'TicketsController@index');
```

When users access **homestead.app/tickets**, we use **TicketsController** to execute the index action. Feel free to change the link path or the action's name to whatever you like.

Then, open the **TicketsController** file, update the **index** action

```
public function index()
{
    $tickets = Ticket::all();
    return view('tickets.index', compact('tickets'));
}
```

We use **Ticket::all()** to get all tickets in our database and store them in the **$tickets** variable.

Before we return the **tickets.index** view, we use the **compact()** method to convert the result to an array, and pass it to the view.

Alternatively, you can use:

```
return view('tickets.index')->with('tickets', $tickets);
```

or

```
return view('tickets.index', ['tickets'=> $tickets]);
```

Those three methods are the same.

Finally, create a new view called **index.blade.php** and place it in the **tickets** directory:

```
@extends('master')
@section('title', 'View all tickets')
@section('content')

    <div class="container col-md-8 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">
                    <h2> Tickets </h2>
                </div>
                @if ($tickets->isEmpty())
                    <p> There is no ticket.</p>
                @else
                    <table class="table">
                        <thead>
                            <tr>
                                <th>ID</th>
                                <th>Title</th>
                                <th>Status</th>
                            </tr>
                        </thead>
```

```
                        <tbody>
                            @foreach($tickets as $ticket)
                                <tr>
                                    <td>{!! $ticket->id !!} </td>
                                    <td>{!! $ticket->title !!}</td>
                                    <td>{!! $ticket->status ? 'Pending' : 'Answe\
red' !!}</td>
                                </tr>
                            @endforeach
                        </tbody>
                    </table>
                @endif
            </div>
    </div>

@endsection
```

We perform the following steps to load the tickets:

```
@if ($tickets->isEmpty())
    <p> There is no ticket.</p>
@else
```

First, we check if the $tickets variable is empty or not. If it's empty, then we display a message to our users.

```
@else
    <table class="table">
        <thead>
            <tr>
                <th>ID</th>
                <th>Title</th>
                <th>Status</th>
            </tr>
        </thead>
        <tbody>
                @foreach($tickets as $ticket)
                    <tr>
                        <td>{!! $ticket->id !!} </td>
                        <td>{!! $ticket->title !!}</td>
                        <td>{!! $ticket->status ? 'Pending' : 'Answered' !!}</td>
                    </tr>
```

```
                @endforeach
            </tbody>
        </table>
@endif
```

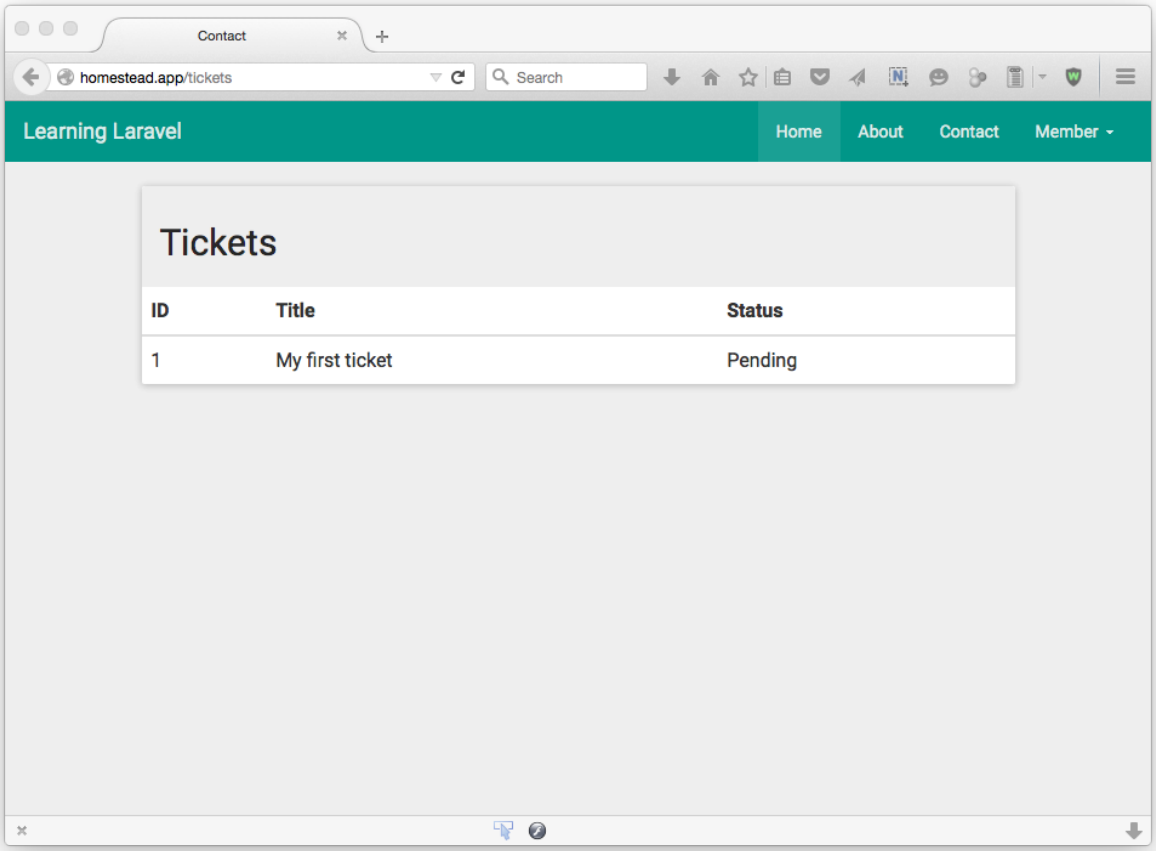
If the $tickets is not empty, we use **foreach() loop** to display all tickets.

```
<td>{{ $ticket->status ? 'Pending' : 'Answered' }}</td>
```

Here is how we can write the **if else** statement in a short way. If the ticket's status is **1**, we say that it's **pending**. If the ticket's status is **0**, we say that it's **answered**.

Feel free to change the name of the status to your liking.

Go to **homestead.app/tickets**, you should be able to view all tickets that you've created!



All tickets

View a single ticket

At this point, viewing a ticket is much easier. When we click on the title of the ticket, we want to display its content and status.

As usual, open **routes.php** file, add:

```
Route::get('/ticket/{slug?}', 'TicketsController@show');
```

You should notice that we use a special route (**/ticket/{slug?}**) here. By doing this, we tell Laravel that any **route parameter** named **slug** will be bound to the **show** action of our **TicketsController**. Simply put, when we visit **ticket/558467e731bb8**, Laravel automatically detects the **slug** (which is **558467e731bb8**) and pass it to the action.

> Note: you can change {slug?} to {slug} or whatever you like. Be sure to put your custom name in the **{ }** brackets.

Next, open **TicketsController**, update the **show action** as follows:

```
public function show($slug)
{
    $ticket = Ticket::whereSlug($slug)->firstOrFail();
    return view('tickets.show', compact('ticket'));
}
```

We pass the **slug** of the ticket we want to display in the **show** action. Then we can use this slug to find the correct ticket via our **Ticket** model's **firstOrFail** method.

The **firstOrFail** method will retrieve the first result of the query. If there is no result, it will throw a **ModelNotFoundException**.

If you don't want to throw an exception, you can use the **first()** method.

```
$ticket = Ticket::whereSlug($slug)->first();
```

Finally, we return the **tickets.show** view with the ticket.

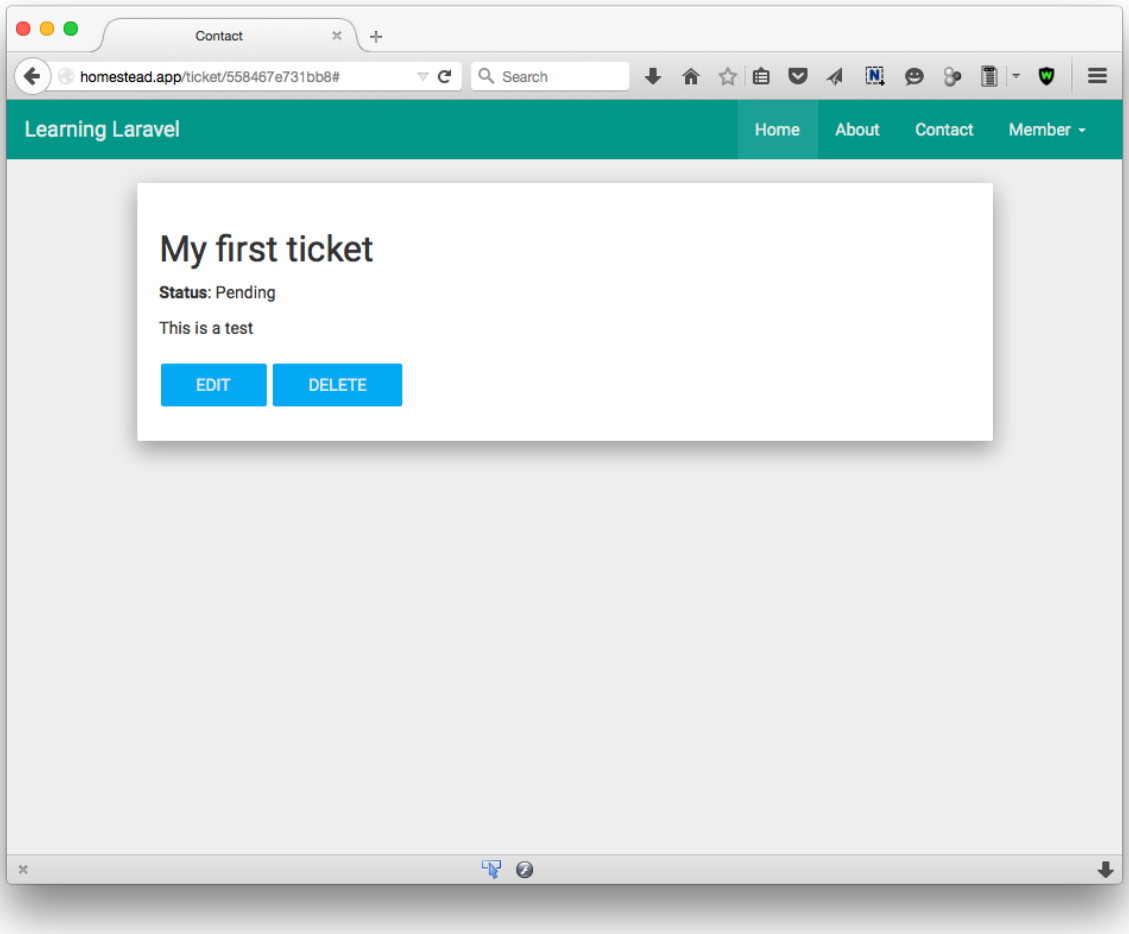
We don't have the **show** view yet. Let's create it:

```
@extends('master')
@section('title', 'View a ticket')
@section('content')

    <div class="container col-md-8 col-md-offset-2">
            <div class="well well bs-component">
                <div class="content">
                    <h2 class="header">{!! $ticket->title !!}</h2>
                    <p> <strong>Status</strong>: {!! $ticket->status ? 'Pending'\
 : 'Answered' !!}</p>
                    <p> {!! $ticket->content !!} </p>
                </div>
                <a href="#" class="btn btn-info">Edit</a>
                <a href="#" class="btn btn-info">Delete</a>
            </div>
    </div>

@endsection
```

Pretty simple, right?

We just display the ticket's title, status and content. We also add the **edit** and **delete** button here to easily edit and remove the ticket.

Now if you access http://homestead.app/ticket/yourSlug, you'll see:



A single ticket

Note: your ticket's slug may be different. Be sure to use a correct slug to view the ticket.

Using a helper function

Laravel has many **helper** functions. These PHP functions are really useful. We can use helper functions to manage paths, modify strings, configure our application, etc.

You can find them here:

http://laravel.com/docs/master/helpers

Now, as we have a view to display all the tickets, let's explore how we can link the **title of the ticket** to the **TicketController's show** action.

Open **tickets/index.blade.php**.

Find:

```
    @foreach($tickets as $ticket)
         <tr>
             <td>{!! $ticket->id !!} </td>
             <td>{!! $ticket->title !!}</td>
             <td>{!! $ticket->status ? 'Pending' : 'Answered' !!}</td>
         </tr>
     @endforeach
```

Update to:

```
@foreach($tickets as $ticket)
    <tr>
        <td>{!! $ticket->id !!}</td>
        <td>
            <a href="{!! action('TicketsController@show', $ticket->slug) !!}">{!\
! $ticket->title !!} </a>
        </td>
        <td>{!! $ticket->status ? 'Pending' : 'Answered' !!}</td>
    </tr>
@endforeach
```

Here, we use **action** function to generate a URL for the **TicketsController's show** action:

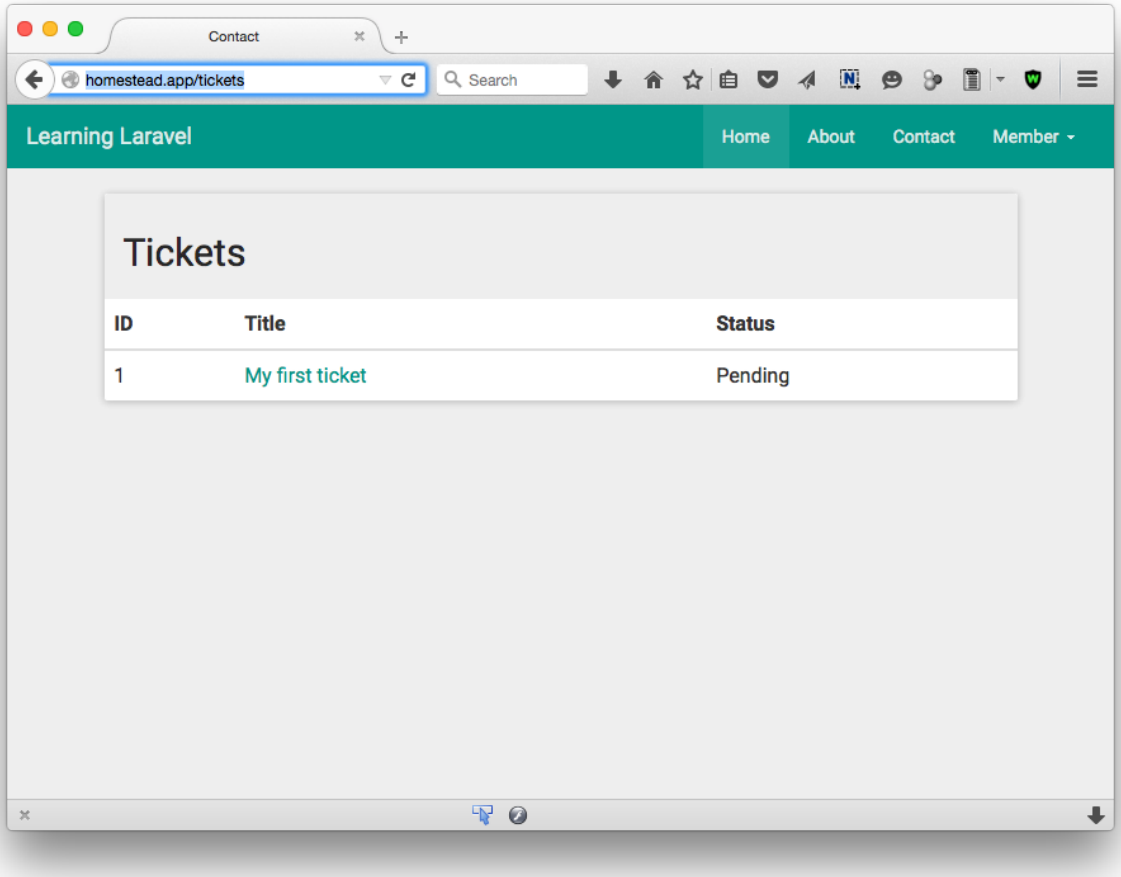
```
action('TicketsController@show', $ticket->slug)
```

The second argument is a **route parameter**. We use **slug** to find the ticket, so we put the ticket's slug here.

Alternatively, you can write the code like this:

```
action('TicketsController@show', ['slug' => $ticket->slug])
```

Now, when you access **homestead.app/tickets**, you can click on the title to view the ticket.



View a single ticket on the all tickets page

Edit a ticket

It's time to move on to create our ticket edit form.

Open **routes.php**, add this route:

```
Route::get('/ticket/{slug?}/edit','TicketsController@edit');
```

When users go to **/ticket/{slug?}/edit**, we redirect them to the **TicketsController's edit action**.

Let's modify the **edit** action:

```
public function edit($slug)
{
    $ticket = Ticket::whereSlug($slug)->firstOrFail();
    return view('tickets.edit', compact('ticket'));
}
```

We find the ticket using its slug, then we use the **tickets.edit** view to display the edit form.

Let's create our **edit view** at **resouces/views/tickets/edit.blade.php**:

```
@extends('master')
@section('title', 'Edit a ticket')

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="well well bs-component">

            <form class="form-horizontal" method="post">

                @foreach ($errors->all() as $error)
                    <p class="alert alert-danger">{{ $error }}</p>
                @endforeach

                @if (session('status'))
                    <div class="alert alert-success">
                        {{ session('status') }}
                    </div>
                @endif

                <input type="hidden" name="_token" value="{!! csrf_token() !!}">

                <fieldset>
                    <legend>Edit ticket</legend>
                    <div class="form-group">
                        <label for="title" class="col-lg-2 control-label">Title<\
/label>
                        <div class="col-lg-10">
                            <input type="text" class="form-control" id="title" n\
ame="title" value="{!! $ticket->title !!}">
                        </div>
                    </div>
                    <div class="form-group">
                        <label for="content" class="col-lg-2 control-label">Cont\
```

```
ent</label>
                        <div class="col-lg-10">
                            <textarea class="form-control" rows="3" id="content"\
 name="content">{!! $ticket->content !!}</textarea>
                        </div>
                    </div>

                    <div class="form-group">
                        <label>
                            <input type="checkbox" name="status" {!! $ticket->st\
atus?"":"checked"!!} > Close this ticket?
                        </label>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button class="btn btn-default">Cancel</button>
                            <button type="submit" class="btn btn-primary">Update\
</button>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
    </div>
@endsection
```

This view is very similar to the **create** view, but we add a new checkbox to modify **ticket's status**.

```
<div class="form-group">
    <label>
        <input type="checkbox" name="status" {!! $ticket->status?"":"checked"!!}\
 > Close this ticket?
    </label>
</div>
```

Try to understand this line:

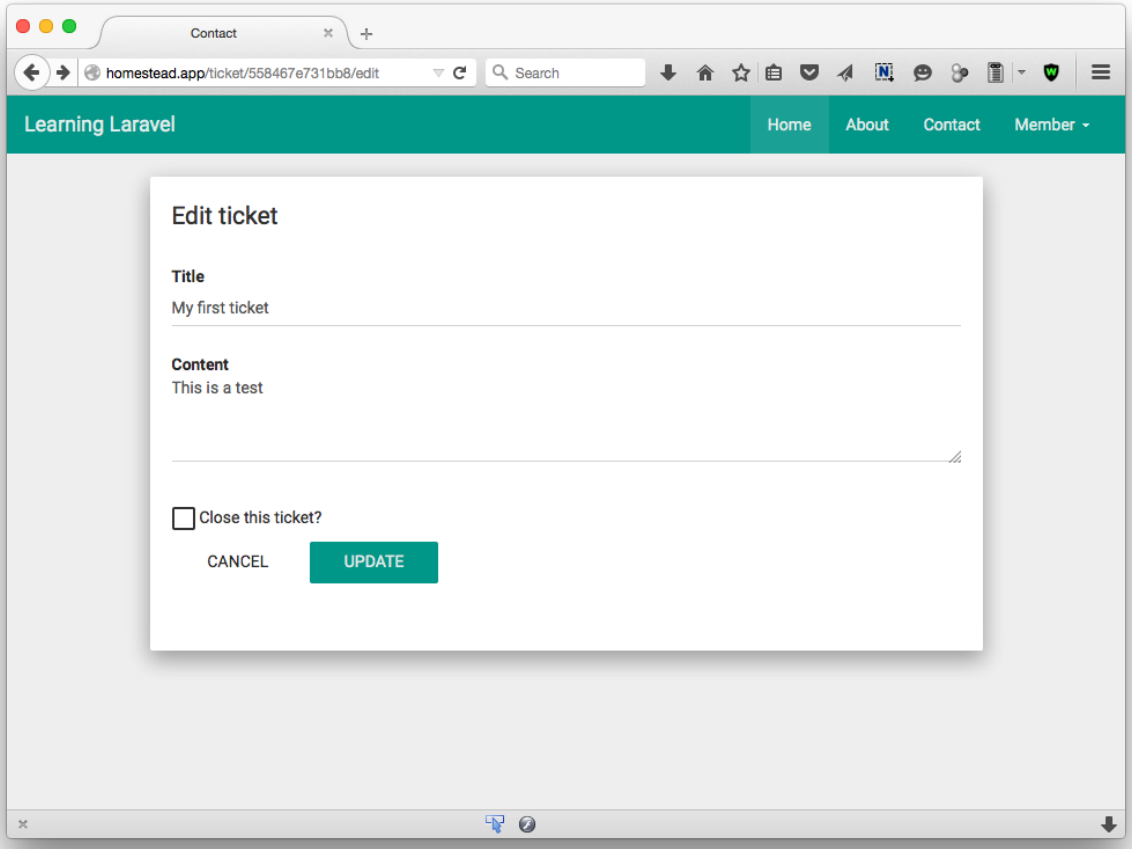
```
{!! $ticket->status?"":"checked"!!}
```

If the status is **1 (pending)**, we display **nothing**, the checkbox is **not checked**. If the status is **0 (answered)** , we display a **checked attribute**, the checkbox is **checked**.

Good! Now, let's open the **show** view, update the **edit button** as follows:

```
<a href="{!! action('TicketsController@edit', $ticket->slug) !!}" class="btn btn\
-info">Edit</a>
```

We use the **action helper** again! When you click on the edit button, you should be able to access the edit form:



Edit form

Unfortunately, we can't update the ticket yet. Remember what you've done to create a new ticket?

We need to use **POST method** to submit the form.

Open **routes.php**, add:

```
Route::post('/ticket/{slug?}/edit','TicketsController@update');
```

Then use **update action** to handle the submission and store the changes.

```
public function update($slug, TicketFormRequest $request)
{
    $ticket = Ticket::whereSlug($slug)->firstOrFail();
    $ticket->title = $request->get('title');
    $ticket->content = $request->get('content');
    if($request->get('status') != null) {
        $ticket->status = 0;
    } else {
        $ticket->status = 1;
    }
    $ticket->save();
    return redirect(action('TicketsController@edit', $ticket->slug))->with('stat\
us', 'The ticket '.$slug.' has been updated!');

}
```

As you notice, you can save the ticket by using the following code:

```
    $ticket = Ticket::whereSlug($slug)->firstOrFail();
    $ticket->title = $request->get('title');
    $ticket->content = $request->get('content');
    if($request->get('status') != null) {
        $ticket->status = 0;
    } else {
        $ticket->status = 1;
    }
    $ticket->save();
```

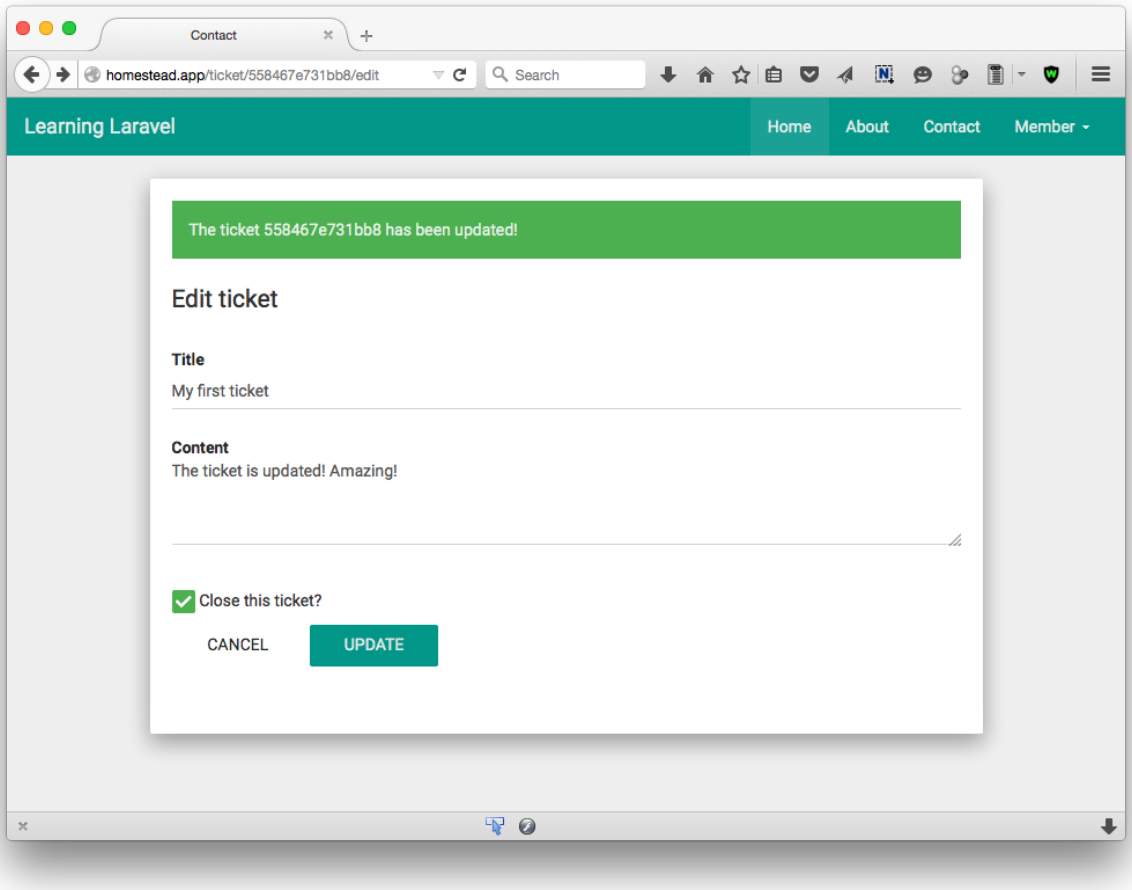
This is how we can check if the users select the status checkbox or not:

```
if($request->get('status') != null) {
    $ticket->status = 0;
} else {
    $ticket->status = 1;
}
```

Finally, we redirect users to the ticket page with a status message:

```
return redirect(action('TicketsController@edit', $ticket->slug))->with('status',\
 'The ticket '.$slug.' has been updated!');
```

Try to edit the ticket now and hit the **update button**!



The ticket has been updated!

Amazing! The ticket has been updated!

Delete a ticket

You've learned how to create, read and update a ticket. Next, you will learn how to delete it. By the end of this section, you'll have a nice **CRUD** application!

First step, let's open the **routes.php** file:

```
Route::post('/ticket/{slug?}/delete','TicketsController@destroy');
```

When we send a POST request to this route, Laravel will take the slug and execute the **TicketsController's destroy** action.

Note: You may use the GET method here.

Open **TicketsController** and update the destroy action:

```
public function destroy($slug)
{
    $ticket = Ticket::whereSlug($slug)->firstOrFail();
    $ticket->delete();
    return redirect('/tickets')->with('status', 'The ticket '.$slug.' has been d\
eleted!');
}
```

We find the ticket using the provided slug. After that, we use **$ticket->delete()** method to delete the ticket.

As always, we then redirect users to the **all tickets** page. Let's update the **index.blade.php** to display the status message:

Find:

```
<div class="panel-heading">
    <h2> Tickets </h2>
</div>
```

Add below:

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

Finally, in order to remove the ticket, all we need to do is create a form to submit a delete request.

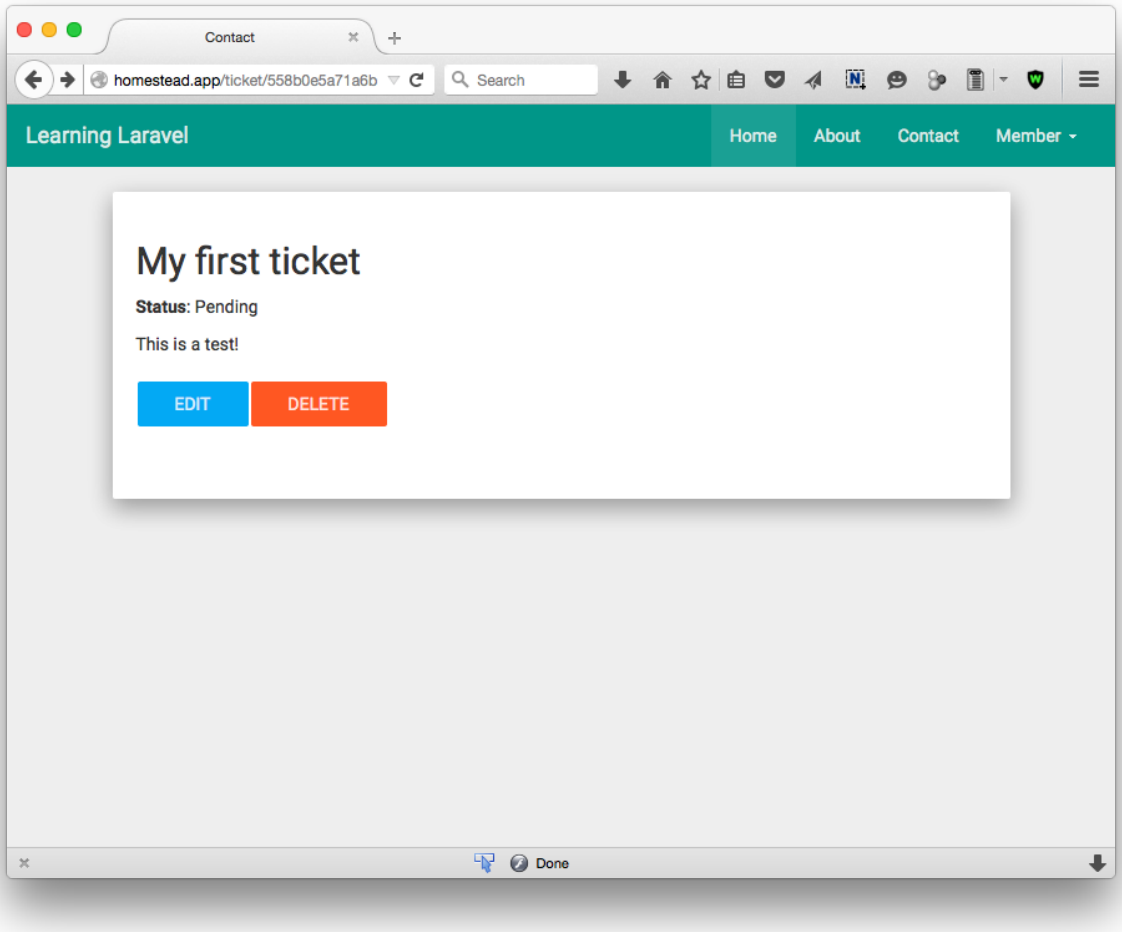
Open **show.blade.php** and find:

```
<a href="{!! action('TicketsController@edit', $ticket->slug) !!}" class="btn btn\
-info">Edit</a>
<a href="#" class="btn btn-info">Delete</a>
```

Update to:

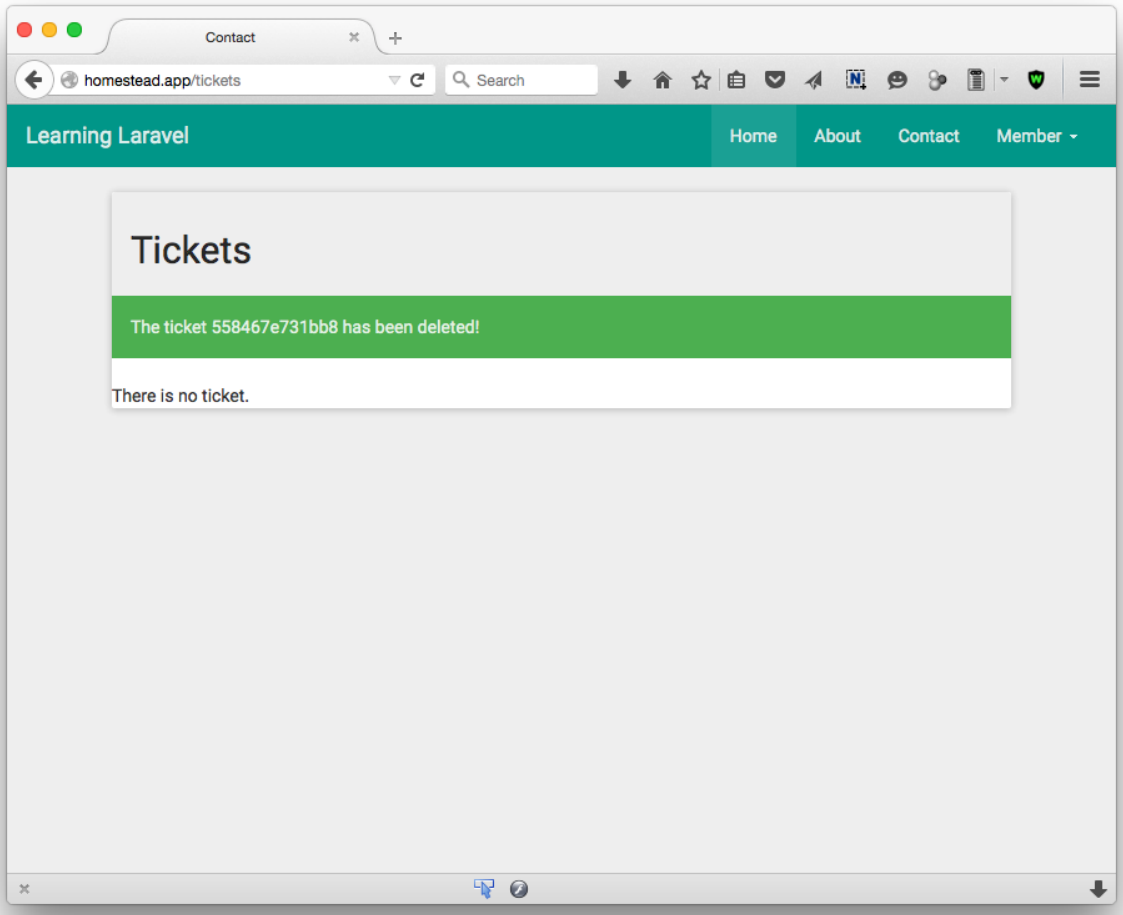
```
<a href="{!! action('TicketsController@edit', $ticket->slug) !!}" class="btn btn\
-info pull-left">Edit</a>

<form method="post" action="{!! action('TicketsController@destroy', $ticket->slu\
g) !!}" class="pull-left">
<input type="hidden" name="_token" value="{!! csrf_token() !!}">
        <div class="form-group">
            <div>
                <button type="submit" class="btn btn-warning">Delete</button>
            </div>
        </div>
</form>
<div class="clearfix"></div>
```

The code above creates a nice delete form for you. When you view a ticket, you should see a different delete button:



New delete button

Now, click the **delete button**, you should be able to remove the ticket!



Delete a ticket successfully

You've just deleted a ticket!

Congratulations! You now have a **CRUD** (Create, Read, Update, Delete) application!

Sending an email

When users submit a ticket, we may want to receive an email to get notified. In this section, you will learn how to send emails using Laravel.

Laravel provides many methods to send emails. You may use a plain PHP method to send emails, or you may use some email service providers such as Mailgun, Sendgrid, Mandrill, Amazon SES, etc.

To send emails on a production server, simply edit the **mail.php** configuration file, which is placed in the **config** directory.

Here is the file without comments:

```
return [

    'driver' => env('MAIL_DRIVER', 'smtp'),

    'host' => env('MAIL_HOST', 'smtp.mailgun.org'),

    'port' => env('MAIL_PORT', 587),

    'from' => ['address' => null, 'name' => null],

    'encryption' => env('MAIL_ENCRYPTION', 'tls'),

    'username' => env('MAIL_USERNAME'),

    'password' => env('MAIL_PASSWORD'),

    'sendmail' => '/usr/sbin/sendmail -bs',

    'pretend' => false,
];
```

To send emails on a local development server (Homestead), simply edit the **.env** file.

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

As usual, you may learn how to use Mailgun, Mandrill and SES drivers at the official docs:

http://laravel.com/docs/master/mail

Because we're working on Homestead, we will learn how to send emails on Homestead using **Gmail** and Sendgrid for FREE!!!!!

Sending emails using Gmail

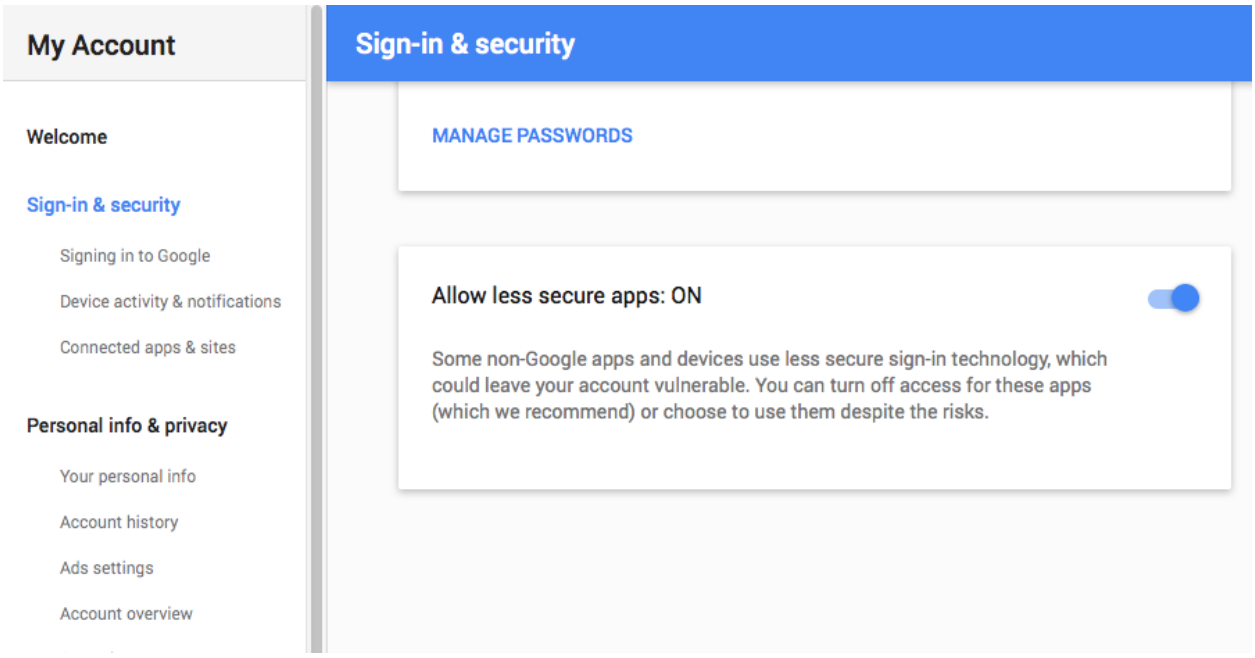
If you have a gmail account, it's very easy to send emails using Laravel 5!

First, go to:

https://myaccount.google.com/security#connectedapps

Take a look at the **Sign-in & security -> Connected apps & sites -> Allow less secure apps** settings.

You must turn the option "Allow less secure apps" **ON**.



Configure Gmail

Once complete, edit the **.env** file:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=yourEmail
MAIL_PASSWORD=yourPassword
MAIL_ENCRYPTION=tls
```

Well done! You're now ready to send emails using Gmail!

If you get this error when sending email: "**Failed to authenticate on SMTP server with username "youremail@gmail.com" using 3 possible authenticators**"

You may try one of these methods:

- Go to https://accounts.google.com/UnlockCaptcha, click continue and unlock your account for access through other media/sites.
- Using a double quote password: "**your password**"
- Try to use only your Gmail username: yourGmailUsername

Sending emails using Sendgrid

Go to **SendGrid**, register a new account:

https://sendgrid.com

When your account is activated, edit the **.env** file:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.sendgrid.net
MAIL_PORT=587
MAIL_USERNAME=yourSendgridUsername
MAIL_PASSWORD=yourPassword
```

Good job! You're now ready to send emails using Sendgrid!

Sending a test email

To send a test email, open **routes.php** file and add this route:

```
Route::get('sendemail', function () {

    $data = array(
        'name' => "Learning Laravel",
    );

    Mail::send('emails.welcome', $data, function ($message) {

        $message->from('yourEmail@domain.com', 'Learning Laravel');

        $message->to('yourEmail@domain.com')->subject('Learning Laravel test ema\
il');

    });

    return "Your email has been sent successfully";

});
```

As you see, we use the **send** method on the **Mail** facade. There are three arguments:

1. The name of the view that we use to send emails.

2. An array of data that we want to pass to the email.
3. A closure that we can use to customize our email subjects, sender, recipients, etc.

When you visit http://homestead.app/sendemail, Laravel will try to send an email. If the email is sent successfully, Laravel will display a message.

> **Note**: Be sure to replace **yourEmail@domain.com** with your **real email address**

Finally, we don't have the **welcome.blade.php** view yet, let's create it and put it in the **emails** directory.

```
<!DOCTYPE html>
<html lang="en-US">
<head>
    <meta charset="utf-8">
</head>
<body>
<h2>Learning Laravel!</h2>

<div>
    Welcome to {!! $name !!} website!
</div>

</body>
</html>
```

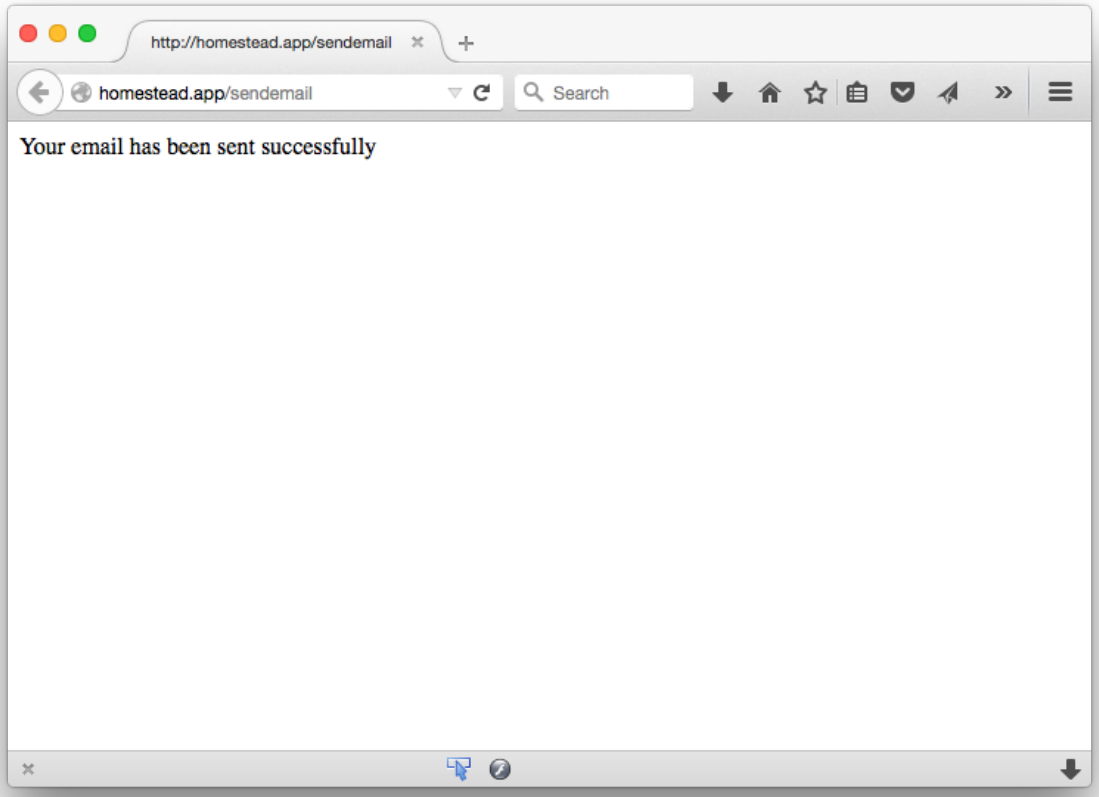
Because we've passed an array containing the **$name** key in the above route, we could display the **name** within this **welcome view** using:

```
{!! $name !!}
```

or

```
<?php echo $name ?>
```

Done! Now go to http://homestead.app/sendemail, you should see:



Sent email successfully

Check your inbox, you should receive a new email!

Feel free to customize your email address, recipients, subjects, etc.

Sending an email when there is a new ticket

Now that we have set up everything, let's send an email when users create a new ticket!

Open **TicketsController.php** and update the **store** action.

Find:

```
return redirect('/contact')->with('status', 'Your ticket has been created! Its u\
nique id is: '.$slug);
```

Add above:

```
    $data = array(
        'ticket' => $slug,
    );

    Mail::send('emails.ticket', $data, function ($message) {
        $message->from('yourEmail@domain.com', 'Learning Laravel');

        $message->to('yourEmail@domain.com')->subject('There is a new ticket!');
    });
```

And don't forget to tell Laravel that you want to use the **Mail** facade here:

Find:

```
class TicketsController extends Controller
```

Add above:

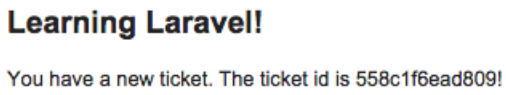
```
use Illuminate\Support\Facades\Mail;
```

As you may notice, we don't have the **emails/ticket.blade.php** view yet. Let's create it!

```
<!DOCTYPE html>
<html lang="en-US">
<head>
    <meta charset="utf-8">
</head>
<body>
<h2>Learning Laravel!</h2>

<div>
    You have a new ticket. The ticket id is {!! $ticket !!}!
</div>

</body>
</html>
```

It's time to **create a new ticket**!

If everything works out well, you should see a new email in your inbox! Here is the email's content:



Got a new email

Reply to a ticket

Welcome to the last section!

In this section, we will learn how to create a form for users to post a reply.

Create a new comments table

First, we need a table to store all the ticket responses. I name this table **comments**.

Let's run this migration command:

```
php artisan make:migration create_comments_table --create=comments
```

Then, open **yourTimestamps_create_comments_table.php**, use **Schema** to create some columns:

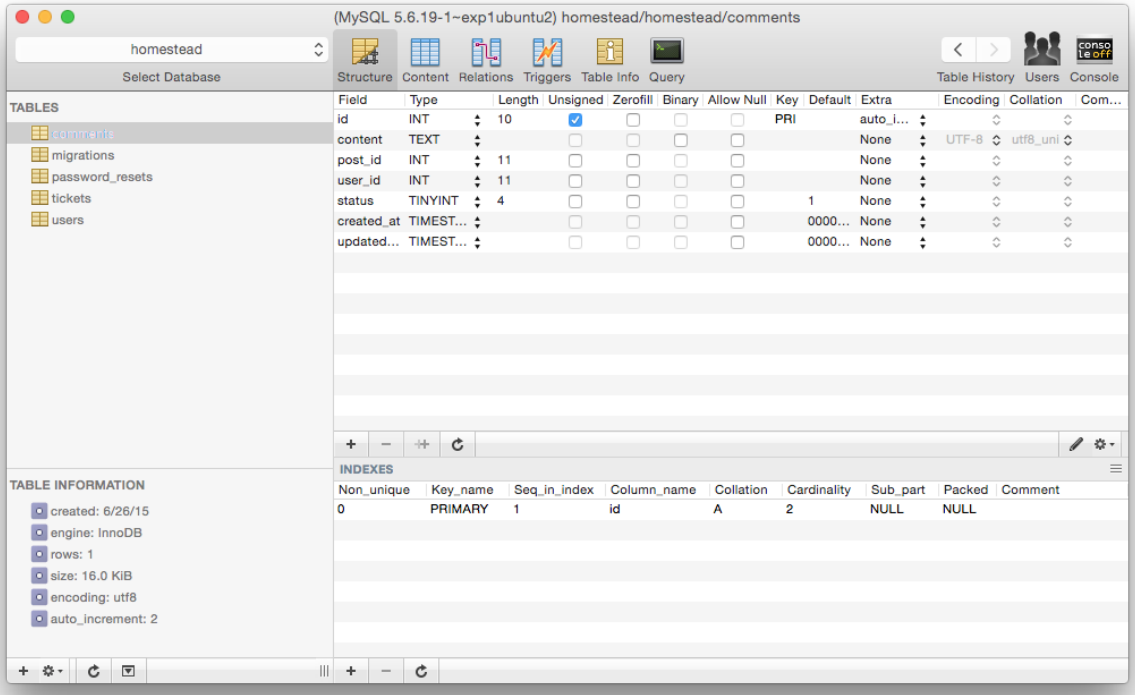
```
<?php

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

class CreateCommentsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('comments', function (Blueprint $table) {
            $table->increments('id');
            $table->text('content');
            $table->integer('post_id');
            $table->integer('user_id');
```

```
            $table->tinyInteger('status')->default(1);
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('comments');
    }
}
```

You should know how to read this file by now. Let's run the **migrate** command to create the comments table and its columns:

```
php artisan migrate
```

Great! Check your database now to make sure that you have created the **comments** table.



Comments table

Introducing Eloquent: Relationships

In Laravel, you can maintain a relationship between tables easily using Eloquent. Here are the relationships that Eloquent supports:

- One to One
- One to Many
- Many to Many
- Has Many Through
- Polymorphic Relations
- Many To Many Polymorphic Relations

What is a relationship?

Usually, tables are related to each other. For instance, our tickets may have many comments (ticket responses). That is **One to Many** relationship.

Once we've defined a **One to Many** relationship between tickets and comments table, we can easily access and list all comments or any related records.

Learn more about Eloquent relationships here:

http://laravel.com/docs/master/eloquent-relationships

Create a new Comment model

As you know, we need a Comment model! Run this command to create it:

```
php artisan make:model Comment
```

Once completed, open it and add this relationship:

```
    public function ticket()
    {
        return $this->belongsTo('App\Ticket');
    }
```

In addition, we may want to make all columns mass assignable except the id column:

```
protected $guarded = ['id'];
```

Instead of using the **$fillable** property, we use the **$guarded** property here.

You now have:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    protected $guarded = ['id'];

    public function ticket()
    {
        return $this->belongsTo('App\Ticket');
    }
}
```

By doing this, we let Eloquent know that this **Comment model** belongs to the **Ticket model**.

Next, open the **Ticket model** and add:

```
public function comments()
{
    return $this->hasMany('App\Comment', 'post_id');
}
```

As you may have guessed, we tell that the Ticket model has many comments and Eloquent can use the **post_id** (ticket id) to find all related comments.

That's it! You've defined a **One to Many** relationship between two tables.

Create a new comments controller

With the relation defined, we will create a new **CommentsController** to handle form submissions and save comments to our database.

Before doing that, let's modify our **routes.php** first:

```
Route::post('/comment', 'CommentsController@newComment');
```

When we send a **POST** request to this route, Laravel will execute the **CommetsController's newComment** action.

It's time to run this command to generate our controller:

```
php artisan make:controller CommentsController
```

Open the new **CommentsController**. Remove all default actions and create a new one:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;
use App\Http\Requests\CommentFormRequest;
use App\Comment;

class CommentsController extends Controller
{
    public function newComment(CommentFormRequest $request)
    {
```

```
        $comment = new Comment(array(
            'post_id' => $request->get('post_id'),
            'content' => $request->get('content')
        ));

        $comment->save();

        return redirect()->back()->with('status', 'Your comment has been created\
!');
    }
}
```

Here is a little tip, you can use **redirect()->back()** to redirect users back to the previous page!

As you see, we still use **Request** here for the validation.

Create a new CommentFormRequest

We don't have the CommentFormRequest yet, so let's create it as well:

```
php artisan make:request CommentFormRequest
```

Create CommentFormRequest

Now, define our rules:

```php
<?php

namespace App\Http\Requests;

use App\Http\Requests\Request;

class CommentFormRequest extends Request
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'content'=> 'required|min:3',
        ];
    }
}
```

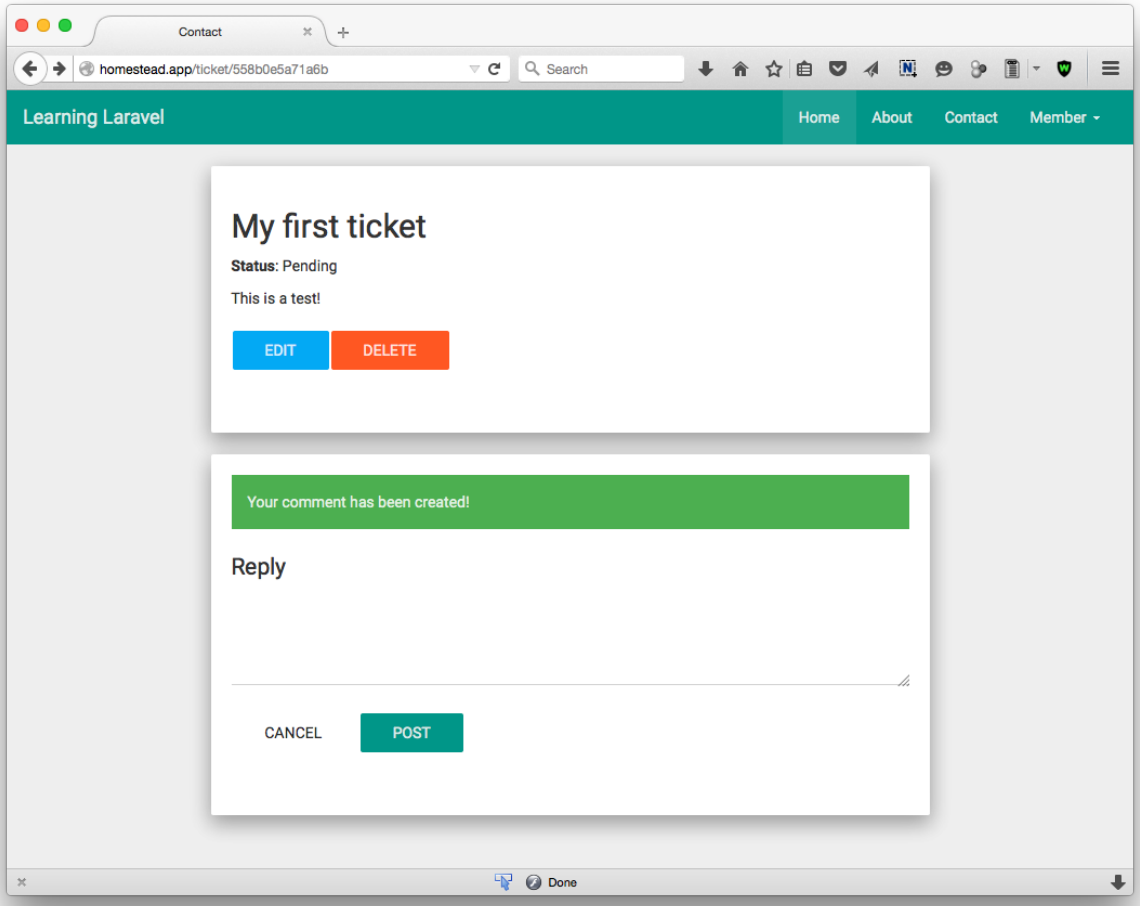
Create a new reply form

Good job! Now open the **tickets.show** view and add this form:

```
        <div class="well well bs-component">
            <form class="form-horizontal" method="post" action="/comment">

                @foreach($errors->all() as $error)
                    <p class="alert alert-danger">{{ $error }}</p>
                @endforeach

                @if(session('status'))
                    <div class="alert alert-success">
                        {{ session('status') }}
                    </div>
                @endif

                <input type="hidden" name="_token" value="{!! csrf_token() !!}">
                <input type="hidden" name="post_id" value="{!! $ticket->id !!}">

                <fieldset>
                    <legend>Reply</legend>
                    <div class="form-group">
                        <div class="col-lg-12">
                            <textarea class="form-control" rows="3" id="content"\
 name="content"></textarea>
                        </div>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button type="reset" class="btn btn-default">Cancel<\
/button>
                            <button type="submit" class="btn btn-primary">Post</\

button>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
```

This form is very similar to the **create ticket form**, we just need to add a new **hidden input** to submit the **ticket id** (post_id) as well.

When you have the form, let's try to reply to a ticket.



Create CommentFormRequest

Yes! You've created a new response!

Display the comments

One last step, we're going to modify the **show action** of our **TicketsController** to list all ticket's comments and pass them to the view.

Open **TicketsController**. The changes in the **show action** are listed as follows:

```
public function show($slug)
{
    $ticket = Ticket::whereSlug($slug)->firstOrFail();
    $comments = $ticket->comments()->get();
    return view('tickets.show', compact('ticket', 'comments'));
}
```

As you may see in the code above, we just need to use this line to list all comments:

```
 $comments = $ticket->comments()->get();
```

Amazing, right? You don't even use a single SQL code!

Now, open the **show.blade.php** view, add this code above the reply form:

```
@foreach($comments as $comment)
    <div class="well well bs-component">
        <div class="content">
            {!! $comment->content !!}
        </div>
    </div>
@endforeach
```

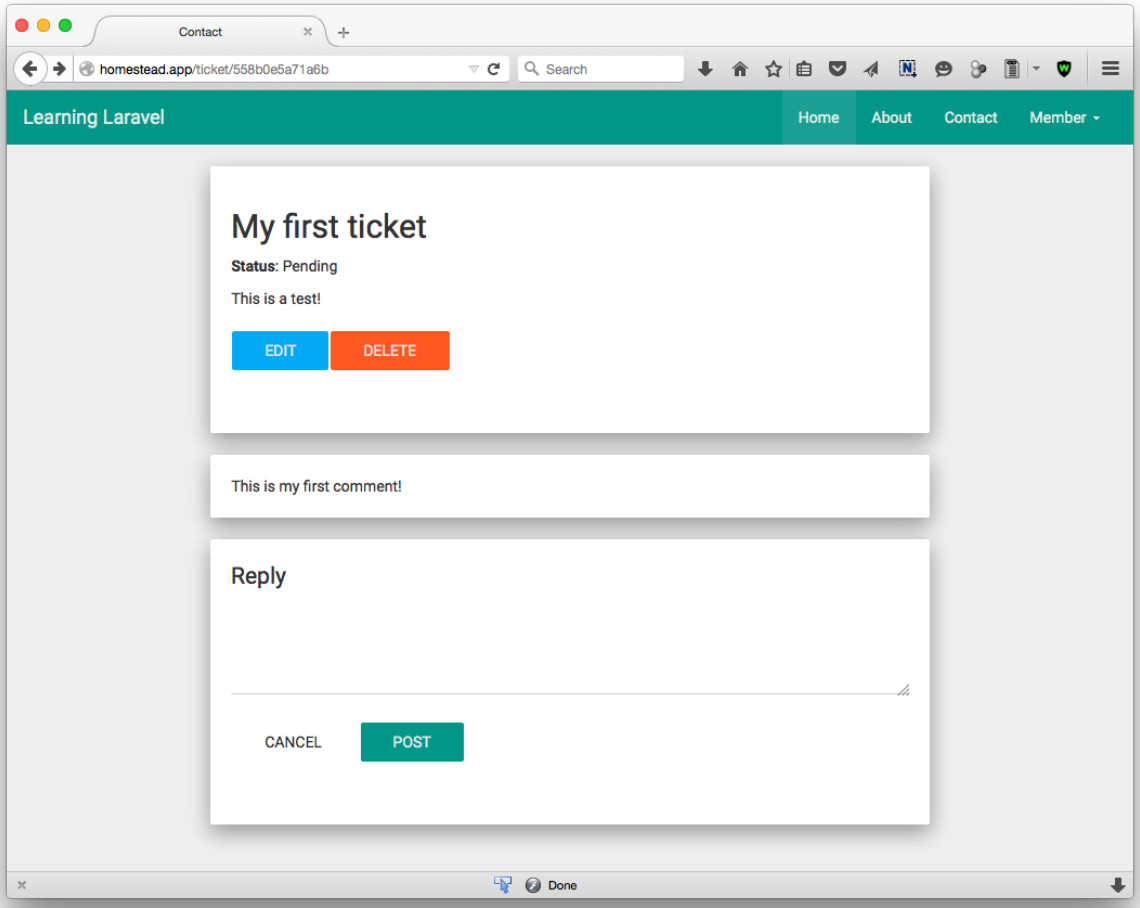
Here is the new **show.blade.php**:

```
@extends('master')
@section('title', 'View a ticket')
@section('content')

    <div class="container col-md-8 col-md-offset-2">
            <div class="well well bs-component">
                <div class="content">
                    <h2 class="header">{!! $ticket->title !!}</h2>
                    <p> <strong>Status</strong>: {!! $ticket->status ? 'Pending'\
 : 'Answered' !!}</p>
                    <p> {!! $ticket->content !!} </p>
                </div>
                <a href="{!! action('TicketsController@edit', $ticket->slug) !!}\
" class="btn btn-info pull-left">Edit</a>

                <form method="post" action={!! action('TicketsController@destroy\
', $ticket->slug) !!} class="pull-left">
```

```
                <input type="hidden" name="_token" value="{!! csrf_token() !!}">
                        <div class="form-group">
                            <div>
                                <button type="submit" class="btn btn-warning">De\
lete</button>
                            </div>
                        </div>
                </form>
                <div class="clearfix"></div>
            </div>

            @foreach($comments as $comment)
                <div class="well well bs-component">
                    <div class="content">
                        {!! $comment->content !!}
                    </div>
                </div>
            @endforeach

            <div class="well well bs-component">
                <form class="form-horizontal" method="post" action="/comment">

                    @foreach($errors->all() as $error)
                        <p class="alert alert-danger">{{ $error }}</p>
                    @endforeach

                    @if(session('status'))
                        <div class="alert alert-success">
                            {{ session('status') }}
                        </div>
                    @endif

                    <input type="hidden" name="_token" value="{!! csrf_token() !\
!}">
                    <input type="hidden" name="post_id" value="{!! $ticket->id !\
!}">

                    <fieldset>
                        <legend>Reply</legend>
                        <div class="form-group">
                            <div class="col-lg-12">
                                <textarea class="form-control" rows="3" id="cont\
```

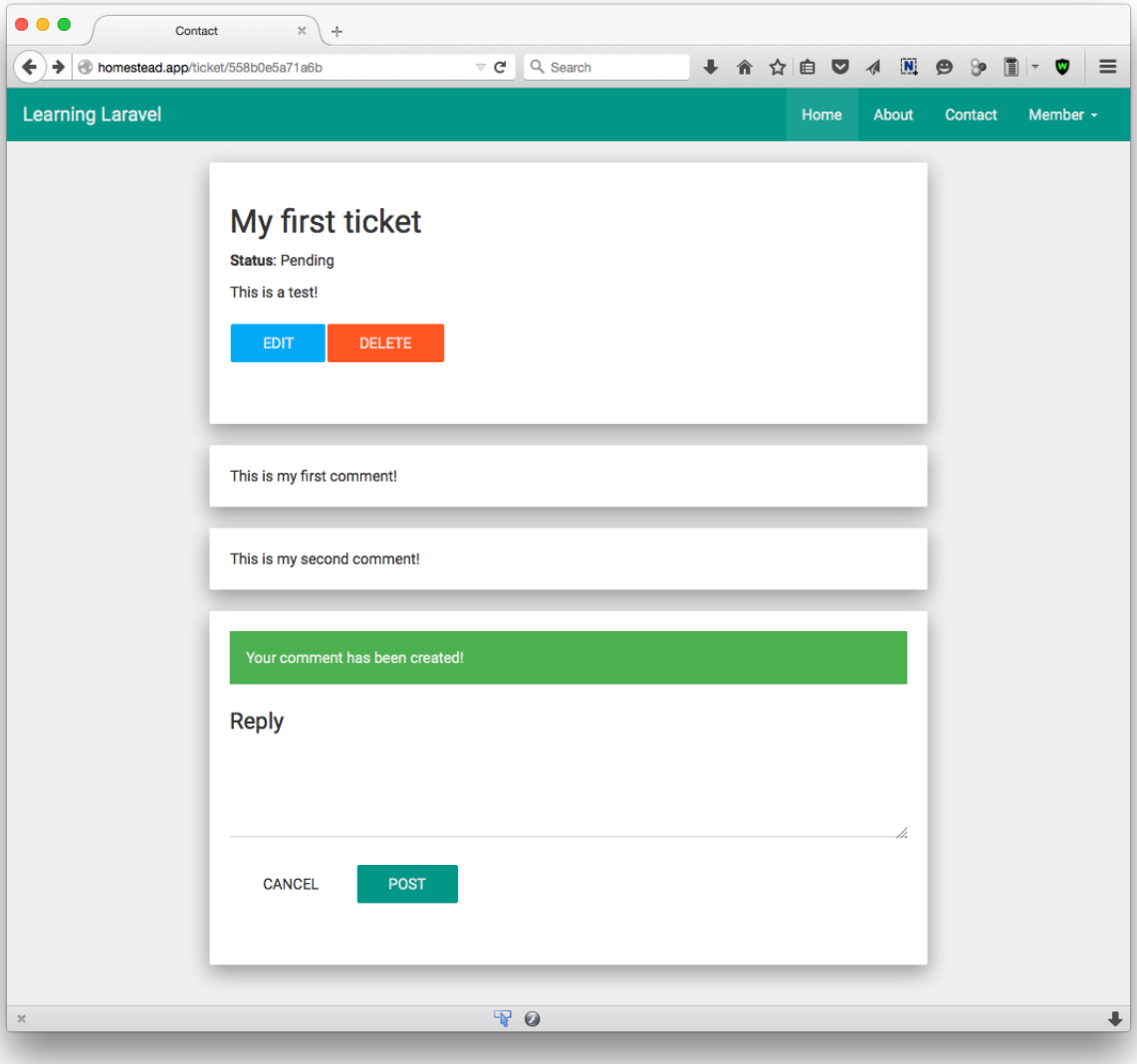
```
ent" name="content"></textarea>
                                </div>
                            </div>

                            <div class="form-group">
                                <div class="col-lg-10 col-lg-offset-2">
                                    <button type="reset" class="btn btn-default">Can\
cel</button>
                                    <button type="submit" class="btn btn-primary">Po\
st</button>
                                </div>
                            </div>
                        </fieldset>
                    </form>
                </div>
            </div>

@endsection
```

Refresh your browser now!



A new reply

To make sure that everything is working, reply again!



The second reply

Congratulations! You now have a fully working support ticket system!

Chapter 3 Summary

In this chapter, you've gone through the different steps involved in creating a ticket support system. Even though the app is simple, it provides us many things to learn:

- You've known how to create databases.
- You've learned one of the most important Laravel features: migrations. Now you can create any database structures that you want.

- You've understood how to use Request to validate forms.
- If you want to use different packages, you've known how to install them.
- You've learned about Laravel's helper functions.
- Sending emails using Gmail and Sendgrid is easy, right?
- You've known how to define Eloquent Relationships and work with those relationships easily.

Basically, you may now be able to create a simple blog system. Feel free to build a different application or customize this application to meet your needs.

The next chapter is where all the fun begin! You will learn to create a complete blog application that has an admin control panel. You may use this application to write your blog posts or you may use it as a starter template for all your amazing applications.

Chapter 4: Building A Blog Application

Up to this point, we have used many Laravel features to build our applications. In this chapter, we're going to build a blog application. By doing this, we will learn about Laravel Authentication, Seeding, Localization, Middleware and many useful features that can help us to have a solid understanding about Laravel 5.

For our purposes, it's always best to think about how our blog application works first.

What do we need to get started?

I assume that you have followed the instructions provided in the previous chapter and you've created a support ticket system. We will use the previous application as our template.

What will we build?

Our simple blog application will have these features:

- Users can be able to register and login.
- Admin can write posts.
- Users can be able to comment on the posts.
- There is an admin control panel to manage users, posts (create, update, delete)
- Permissions and roles system.
- Admin can create/remove/edit categories.

Let's get started!

Building a user registration page

Since Laravel 5, implementing authentication has become very easy, Laravel has provided almost everything for us.

In this section, you will learn how to create a simple registration page.

First, we need to create some routes. Open **routes.php** and add these routes:

```
Route::get('users/register', 'Auth\AuthController@getRegister');
Route::post('users/register', 'Auth\AuthController@postRegister');
```

The first route will provide the **registration form**. The second route will **process the form**. Both routes are handled by the **AuthController**'s actions: **getRegister** and **postRegister**.

By default, Laravel has created the **AuthController** for us. You can find it in the **Auth** directory.

Let's open the file and take a look at:

```
protected function validator(array $data)
{
    return Validator::make($data, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255|unique:users',
        'password' => 'required|confirmed|min:6',
    ]);
}
```

As you may notice, there are **three fields** here: **name**, **email** and **password**.

When users visit **users/register**, this AuthController will render a registration view, which contains a registration form.

Unfortunately, Laravel doesn't create the registration **view** for us, we have to create it manually. The registration views should be placed at **resources/views/auth/register.blade.php**.

Here is the code:

```
@extends('master')
@section('name', 'Register')

@section('content')
    <div class="container col-md-6 col-md-offset-3">
        <div class="well well bs-component">

            <form class="form-horizontal" method="post">

                @foreach ($errors->all() as $error)
                    <p class="alert alert-danger">{{ $error }}</p>
                @endforeach

                 {!! csrf_field() !!}

                <fieldset>
```

```
                    <legend>Register an account</legend>
                    <div class="form-group">
                        <label for="name" class="col-lg-2 control-label">Name</l\
abel>
                        <div class="col-lg-10">
                            <input type="text" class="form-control" id="name" pl\
aceholder="Name" name="name" value="{{ old('name') }}">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="email" class="col-lg-2 control-label">Email<\
/label>
                        <div class="col-lg-10">
                            <input type="email" class="form-control" id="email" \
placeholder="Email" name="email" value="{{ old('email') }}">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="password" class="col-lg-2 control-label">Pas\
sword</label>
                        <div class="col-lg-10">
                            <input type="password" class="form-control"  name="p\
assword">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="password" class="col-lg-2 control-label">Con\
firm password</label>
                        <div class="col-lg-10">
                            <input type="password" class="form-control"  name="p\
assword_confirmation">
                        </div>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button type="reset" class="btn btn-default">Cancel<\
/button>
                            <button type="submit" class="btn btn-primary">Submit\
```

```
</button>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
    </div>
@endsection
```

You've created many forms in the previous chapters, so I guess you should understand this file clearly.

Here is a new tip. Instead of using this line to generate a new CSRF token:

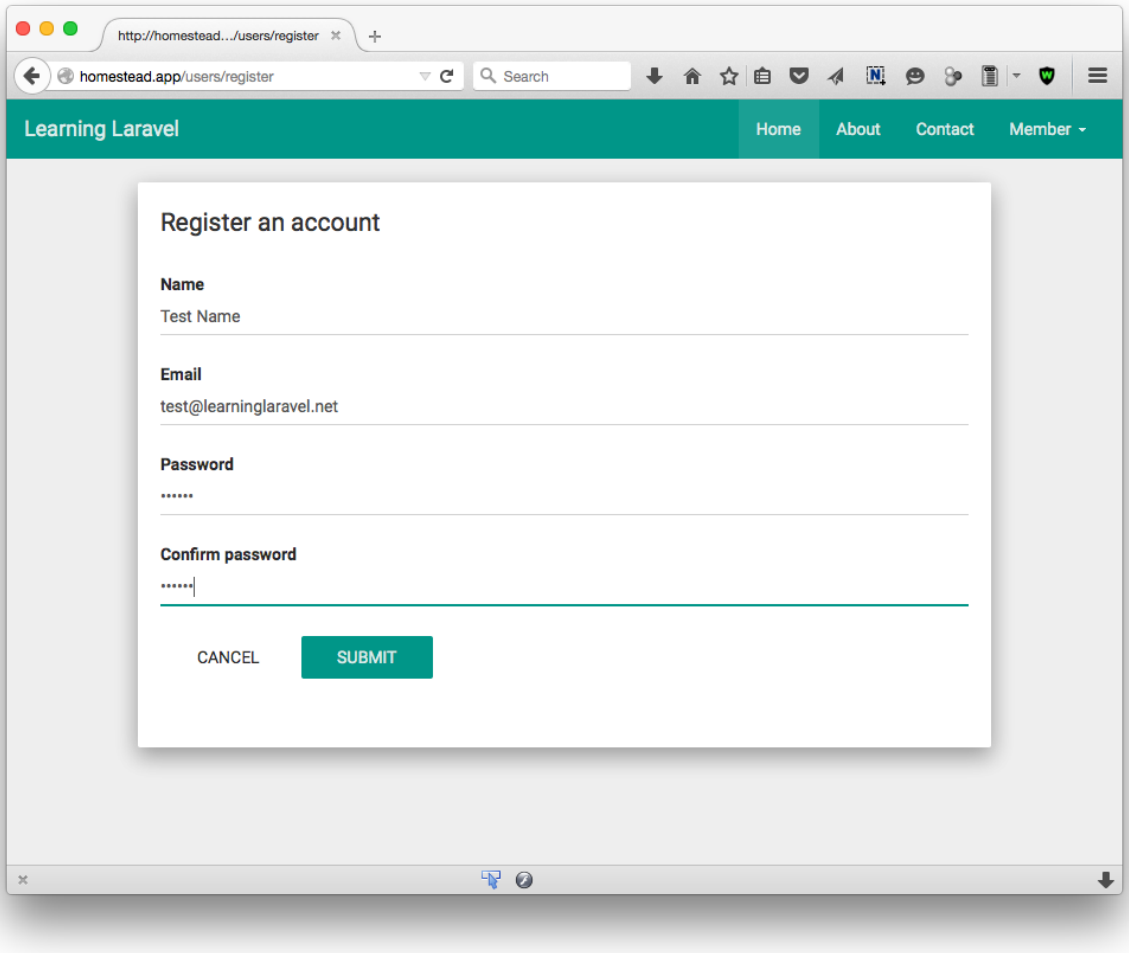
```
<input type="hidden" name="_token" value="{!! csrf_token() !!}">
```

You can simply use this helper function to generate the token!

```
{!! csrf_field() !!}
```

You also may notice that there is a new **old()** method. When the form's validation fails, the users will be redirected back to the form. We use this method to display the old users' input, so they don't have to fill in all the fields again.

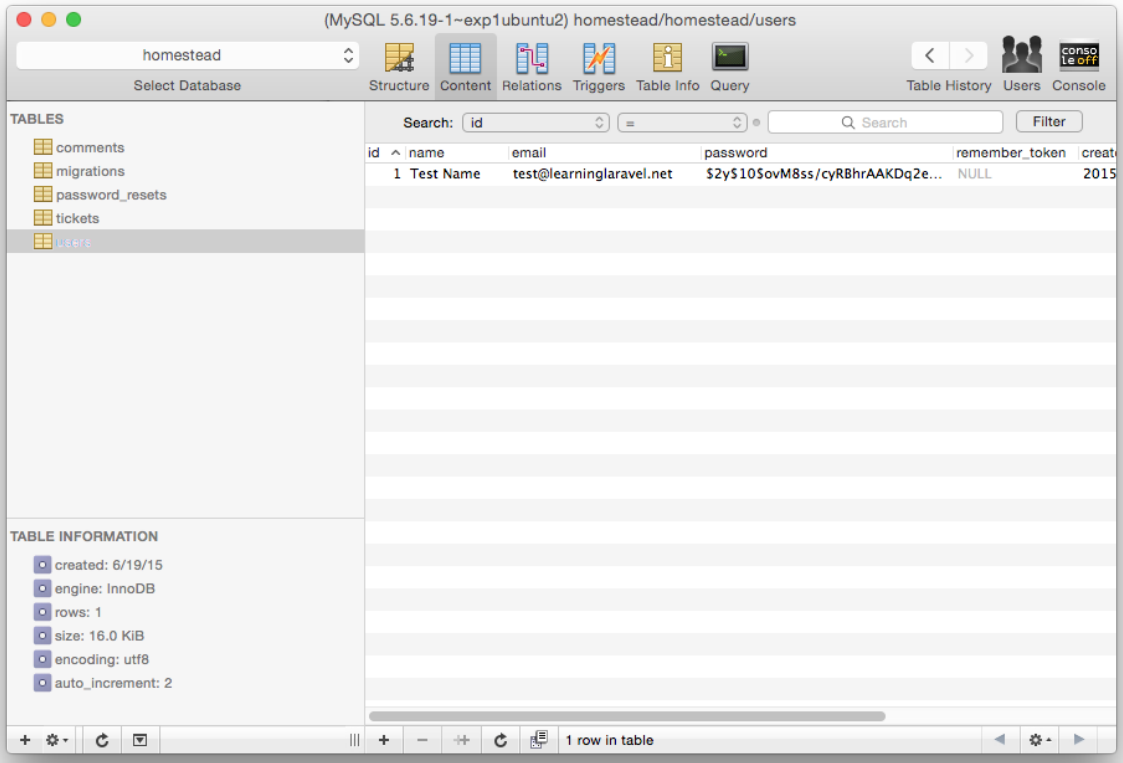
Now, go to http://homestead.app/users/register, you should see a nice user registration form.



Register a new user account

Fill in all the fields, and hit submit! You've registered a new user!

Check your database now, you should see:



A new user

By default, Laravel automatically redirects you to the **/home** URI. If you see this error when you're at **http://homestead.app/home**:

```
NotFoundHttpException in RouteCollection.php line 161:
```

Then that means your **routes.php** file doesn't have the **home** route:

```
Route::get('home', 'PagesController@home');
```

You can fix this error by adding the **home route** into your app or you may simply ignore it. You'll learn how to redirect users to other locations in the next section.

Creating a logout functionality

When users register for a new account, Laravel will validate the registration form. If the validation rules pass, Laravel will save data to the database, log the users in and redirect them to the **/home** URI of our application (home page).

To redirect users to other locations, open **AuthController.php**, find:

```
use AuthenticatesAndRegistersUsers;
```

and add this line below:

```
protected $redirectPath = '/yourPath';
```

You may notice that when you go to the registration page, Laravel will automatically redirect you back to the home page because you're now logged in.

> **Note**: In Laravel 5.2, the **$redirectPath** variable has been changed to **$redirectTo**.

We don't have the **logout** functionality yet, but don't worry, it's very easy to implement.

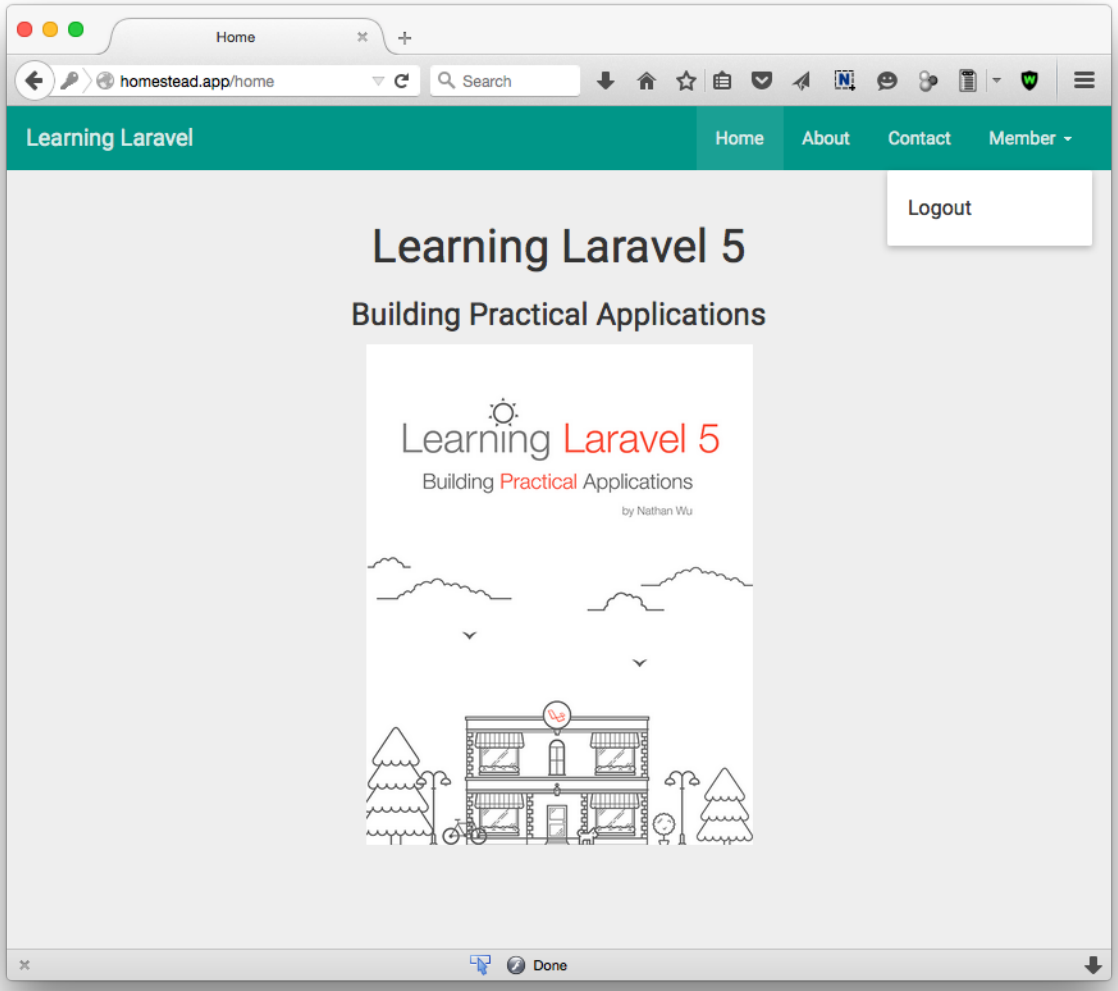
Open the **shared/navbar.blade.php** view, find:

```
<li><a href="/users/register">Register</a></li>
<li><a href="/users/login">Login</a></li>
```

Replace with the following code:

```
@if (Auth::check())
    <li><a href="/users/logout">Logout</a></li>
@else
    <li><a href="/users/register">Register</a></li>
    <li><a href="/users/login">Login</a></li>
@endif
```

To check whether a user is logged in, we can use the **Auth::check()** method. In the code above, if users are logged in, we will display a logout link.



A logout link

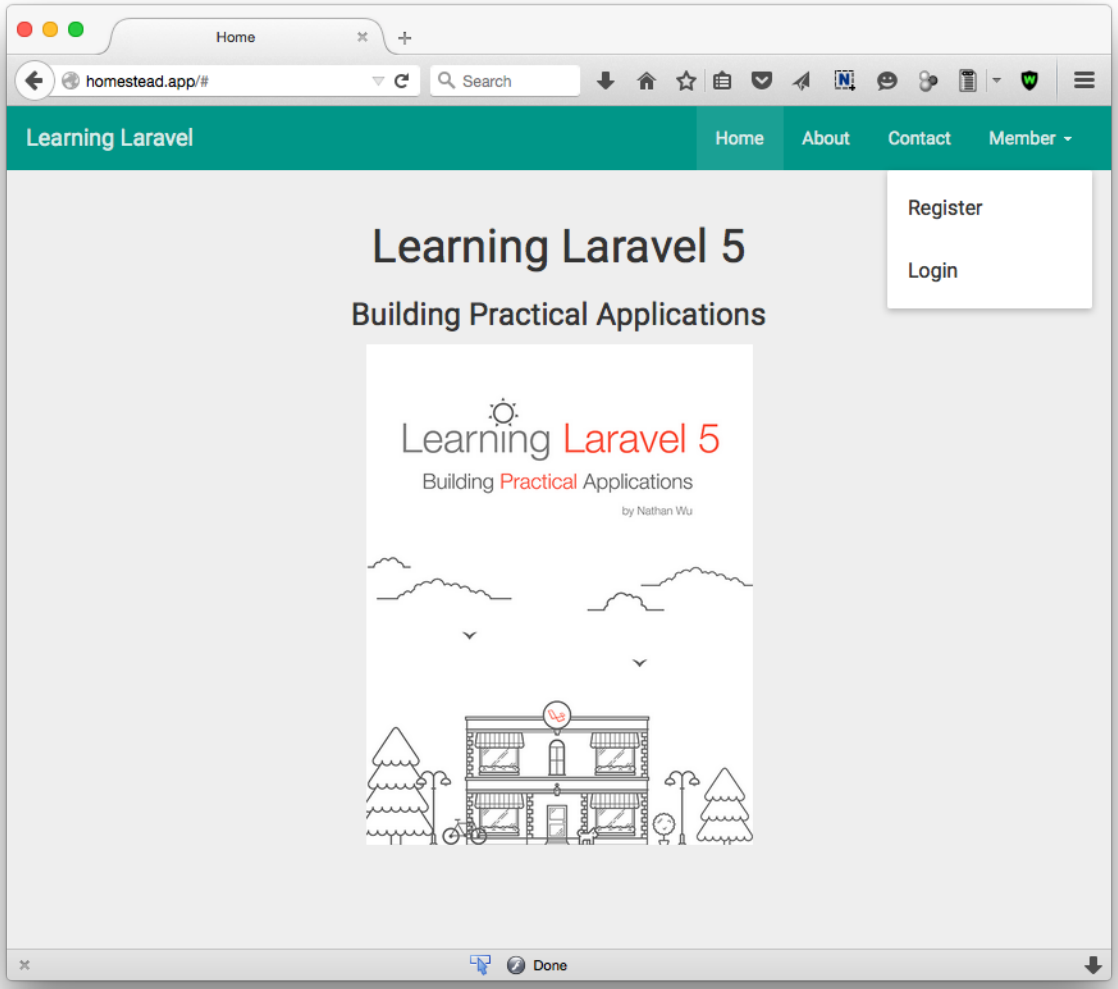
To make the link work, open **routes.php**, add:

```
Route::get('users/logout', 'Auth\AuthController@getLogout');
```

If you're using **Laravel 5.2**, open your **AuthController**, update the **constructor** (aka **construct method**) as follows:

```
public function __construct()
{
    $this->middleware('guest', ['except' => ['logout', 'getLogout']]);
}
```

As you see, when users visit the **users/logout** link, we will use **AuthController's getLogout** action to log the users out.



A new user

Try to test the functionality yourself! Now you can be able to log out!

Creating a login page

It's time to create our login form. As always, we're going to define two different actions on the **users/login** route:

```
Route::get('users/login', 'Auth\AuthController@getLogin');
Route::post('users/login', 'Auth\AuthController@postLogin');
```

The **GET** route will display the login form, the **POST** route will process the form.

As you may have guessed, you should create a **login** view now. The view should be placed at **views/auth/login.blade.php**.

```
@extends('master')
@section('name', 'Login')

@section('content')
    <div class="container col-md-6 col-md-offset-3">
        <div class="well well bs-component">

            <form class="form-horizontal" method="post">

                @foreach ($errors->all() as $error)
                    <p class="alert alert-danger">{{ $error }}</p>
                @endforeach

                 {!! csrf_field() !!}

                <fieldset>
                    <legend>Login</legend>

                    <div class="form-group">
                        <label for="email" class="col-lg-2 control-label">Email<\
/label>
                        <div class="col-lg-10">
                            <input type="email" class="form-control" id="email" \
name="email" value="{{ old('email') }}">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="password" class="col-lg-2 control-label">Pas\
```

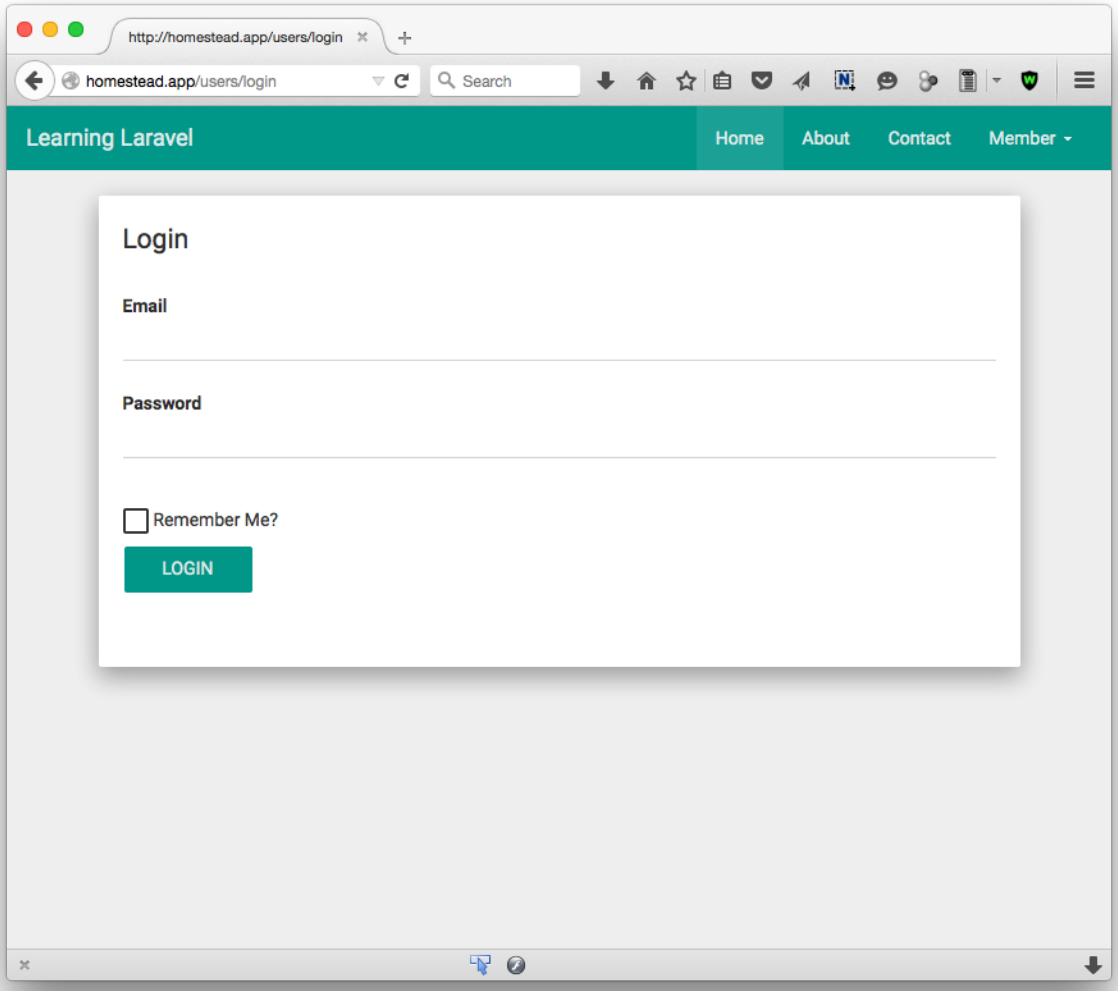
```
sword</label>
                        <div class="col-lg-10">
                            <input type="password" class="form-control"  name="p\
assword">
                        </div>
                    </div>

                    <div class="checkbox">
                        <label>
                            <input type="checkbox" name="remember" > Remember Me?
                        </label>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button type="submit" class="btn btn-primary">Login<\
/button>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
    </div>
@endsection
```

Our login form is simple, it has two fields: **email** and **password**. Users have to enter the correct email and password here to login.

Laravel also provides the **remember me** functionality out of the box. We can implement it by simply creating a remember checkbox.

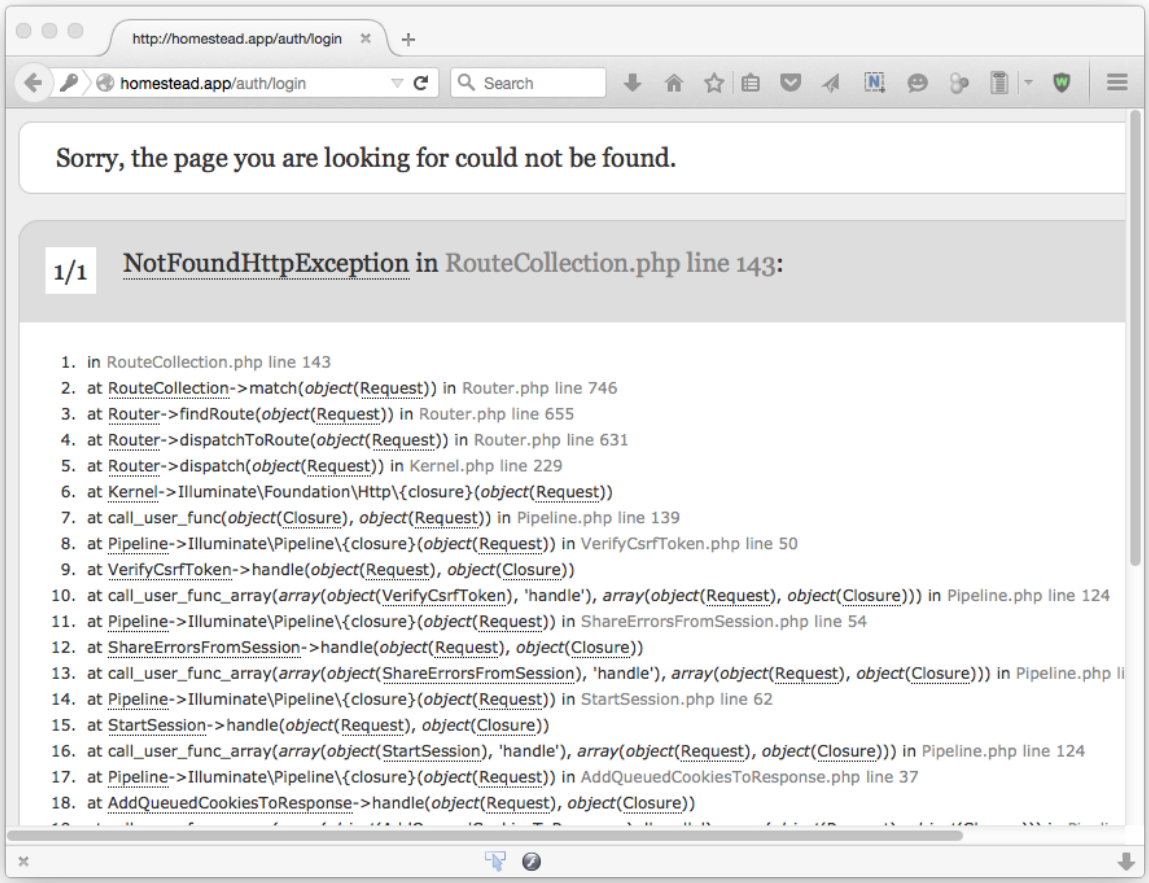
> **Note**: The name of the checkbox should be "**remember**".

Well done. Now let's go to http://homestead.app/users/login and try to login with your email and password!



Login form

If you try to login with a wrong information, you may encounter this error:



Error login

By default, Laravel use **auth/login** route to authenticate users, and users will be redirected to that route if they enter wrong credentials. However, we use **users/login** route, so we need to let Laravel know that by opening the **AuthController** class and adding a **loginPath** property:

```
protected $loginPath = '/users/login';
```

Now everything should be working fine.

Add authentication throttling to your application

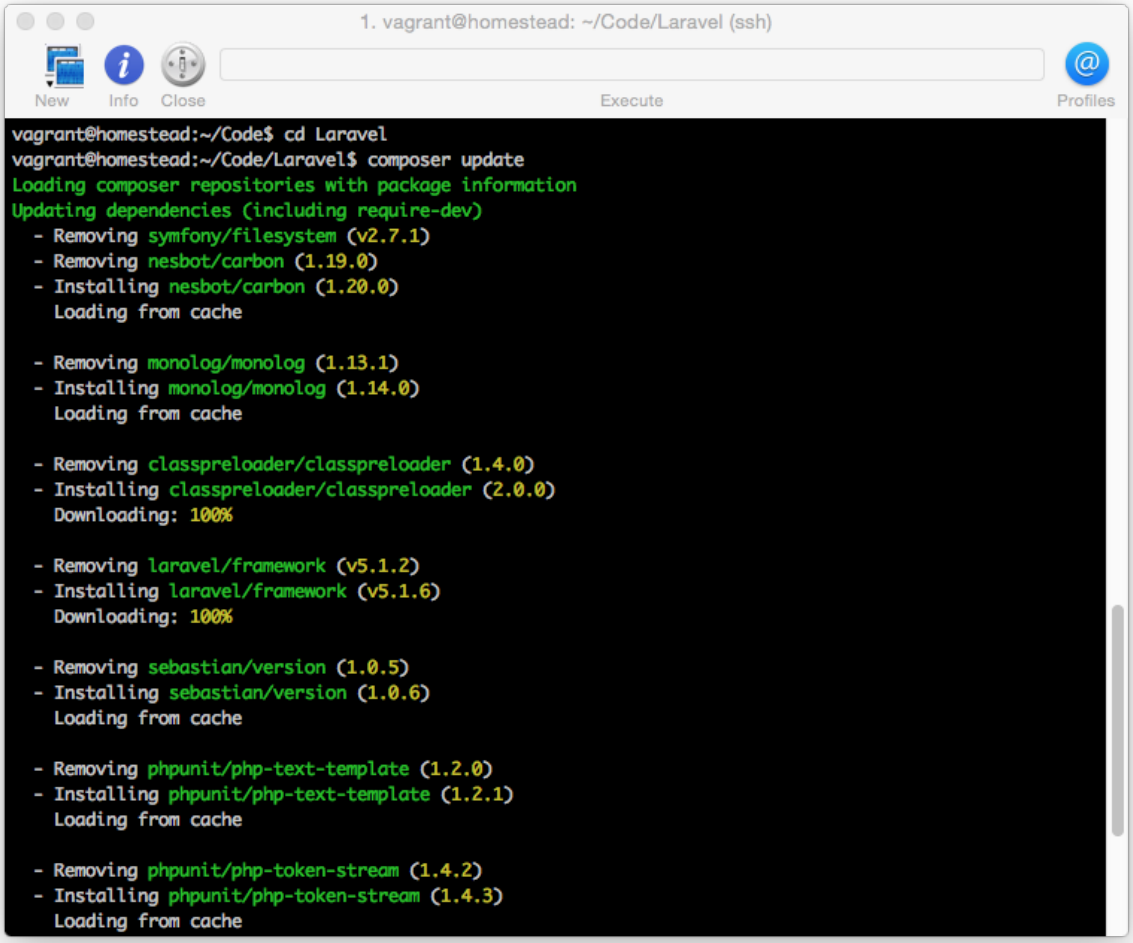
Laravel 5.1.4 introduces a new feature: "**Authentication Throttling**". This feature is used to throttle login attempts to your application. If users try to login many times, they can't be able to login for one minute.

> **Note**: If you're using Laravel 5.1.4 or newer, this feature has been implemented already. You may skip these steps. By the way, It's still good to know how it works.

As we're using the **AuthController** class for authentication, you can easily integrate this feature by doing the following steps:

First, be sure to update your application to **version 5.1.4** or newer. You can update your application by **vagrant ssh** to your **VM**, then **cd** to your Laravel root directory, and run:

```
composer update
```



Updating our application

Once updated, open **AuthController.php** file and find:

```
use App\Http\Controllers\Controller;
```

Add below:

```
use Illuminate\Foundation\Auth\ThrottlesLogins;
```

Find:

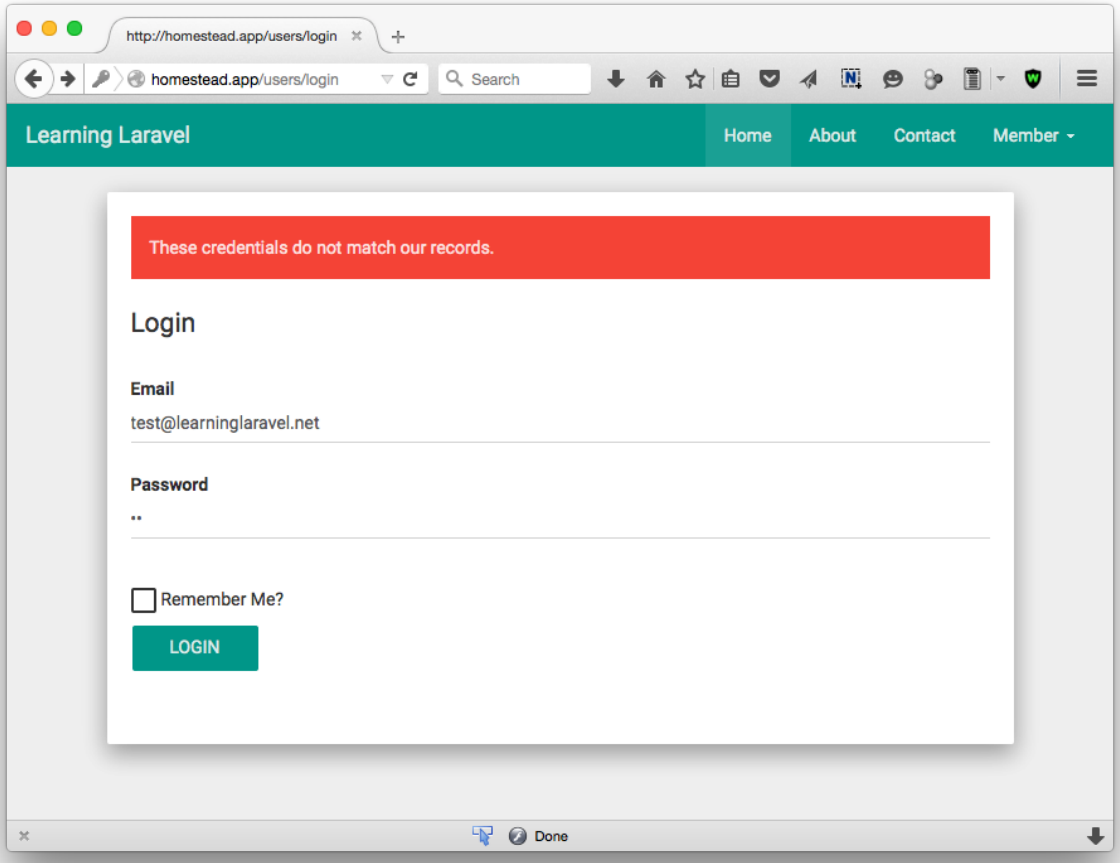
```
use AuthenticatesAndRegistersUsers;
```

Modify to:

```
use AuthenticatesAndRegistersUsers, ThrottlesLogins;
```

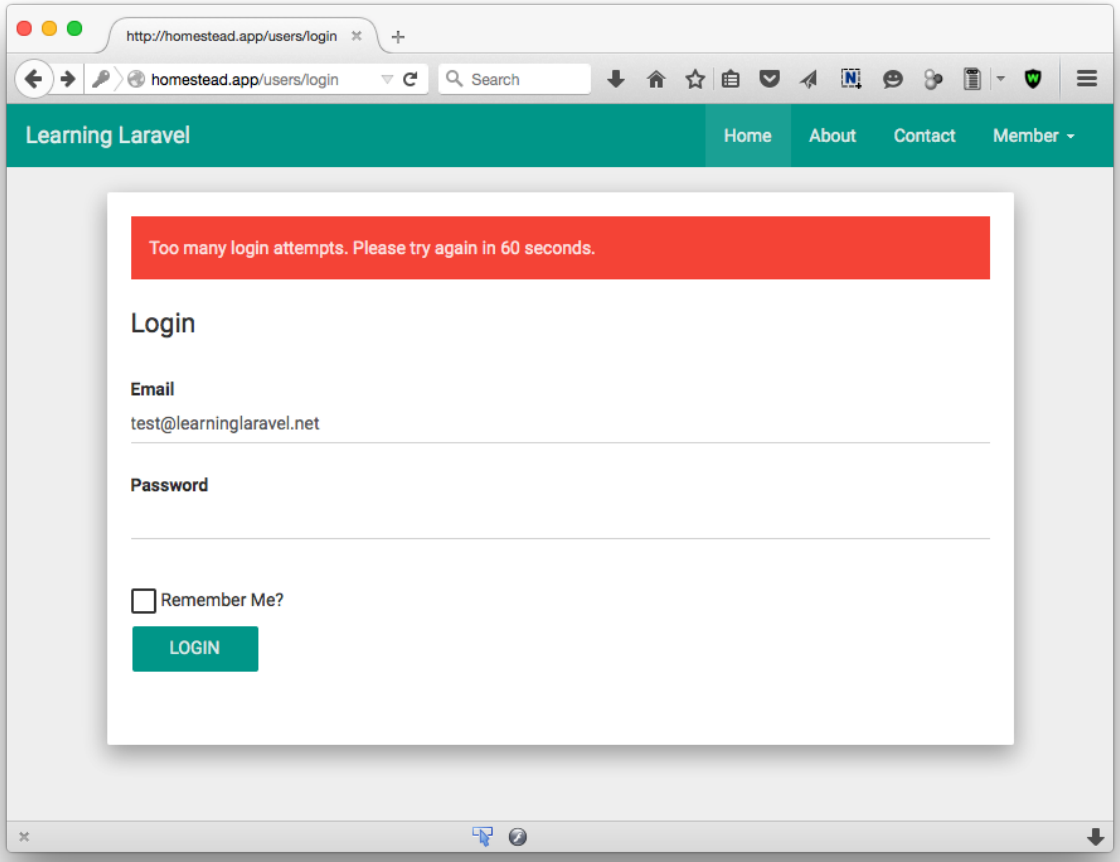
Done! You've implemented the throttling feature.

Now, let's try to login with wrong credentials;



Wrong credentials

When you try to login many times, there would be an error message:



Too many attempts

Building an admin area

Imagine that our application will have an administration section and a front end section, there would be many routes. We need to find a way to organize all the routes.

Additionally, we may want to allow only administrators to access our admin area. Fortunately, Laravel helps us to do that easily.

Let's open **routes.php** file and add:

```
Route::group(array('prefix' => 'admin', 'namespace' => 'Admin', 'middleware' => \
'auth'), function () {
    Route::get('users', 'UsersController@index');
});
```

By using **Route::group** we can group all related routes together and apply some specific rules for them.

```
'prefix' => 'admin'
```

We use the **prefix attribute** to **prefix** each route in the **route group** with a **URI (admin)**. In this case, when we go to http://homestead.app/admin or any routes that contain the admin prefix, Laravel will understand that we want to access the admin area.

Laravel 5.1 uses PSR-2 autoloading standard, which is a coding style. Your applications controllers, models and other classes must be namespaced.

What are namespaces? According to PHP docs: "**namespaces are designed to solve two problems that authors of libraries and applications encounter when creating re-usable code elements such as classes or functions.**". Simply put, let's think namespaces as the last names of persons. When many persons have the same name, we will use their last name to distinguish them apart.

To **namespace** a class, we can use the **namespace** keyword and declare the namespace at the top of the file before any other code. For instance, let's open **AuthController.php** class, you should see its namespace:

```
<?php
namespace App\Http\Controllers\Auth;
```

You can learn more about Namespace here:

http://php.net/manual/en/language.namespaces.rationale.php

As you may have seen, I have defined a namespace called **Admin** for our routes:

```
'namespace' => 'Admin'
```

If we have a class that has a namespace like this:

```
<?php

namespace App\Http\Controllers\Admin;
```

Laravel will know exactly which class that we want to load and where to find it.

Laravel 5 also has a new feature called **HTTP Middleware** (or simply **Middleware**). Basically, we use it to filter our applications' HTTP requests. For example, I use:

```
'middleware' => 'auth'
```

That means I want to use the **auth middeware** for this route group. Only authenticated users can access these routes. We'll learn more about Middleware soon.

List all users

We now have a route group for our admin control panel. Let's list all the users so that we can view and manage them easier.

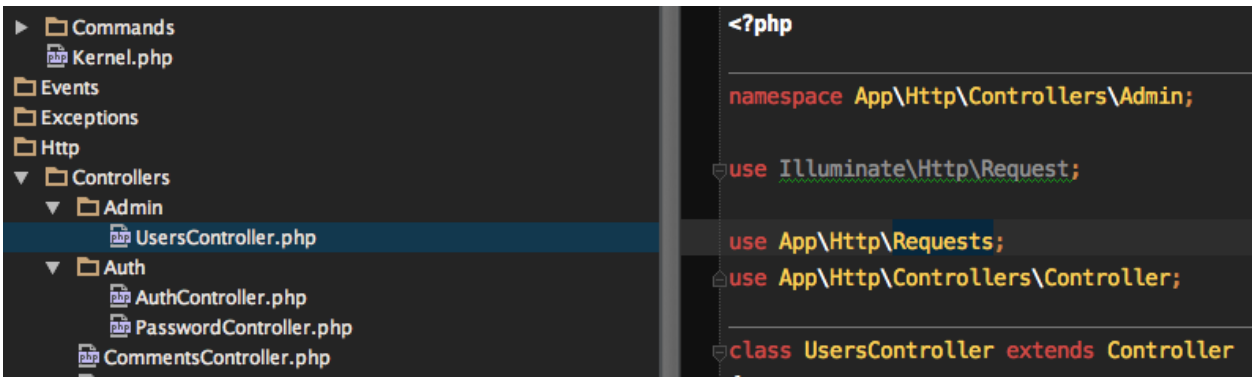
As you know, we have defined a route here:

```
Route::get('users', 'UsersController@index');
```

We don't have the **UsersController** yet, let's use **PHP Artisan** to create it:

```
php artisan make:controller Admin/UsersController
```

This time, our code is a little bit different. It has **/Admin** before the controller's name. Laravel is clever. When we code like this, it will automatically create a directory called **Admin**, and put the **UsersController** file into the **Admin directory** for you. More than that, our controller has been namespaced!



UsersController

At this point, I think that you've known how to list all the users. Basically, the process is very similar to what we've done to list all the tickets in Chapter 3.

First, we tell Laravel that we want to use the **User model**:

Find:

```
use App\Http\Controllers\Controller;
```

Add below:

```
use App\User;
```

Now, update the **index()** action as follows:

```
public function index()
{
    $users = User::all();
    return view('backend.users.index', compact('users'));
}
```

As you may have guessed, we will put all the administration views in the **backend** directory.

We're going to create a new view called **index.blade.php** to display all the users. Let's create a new **users** directory as well and put the index view inside.

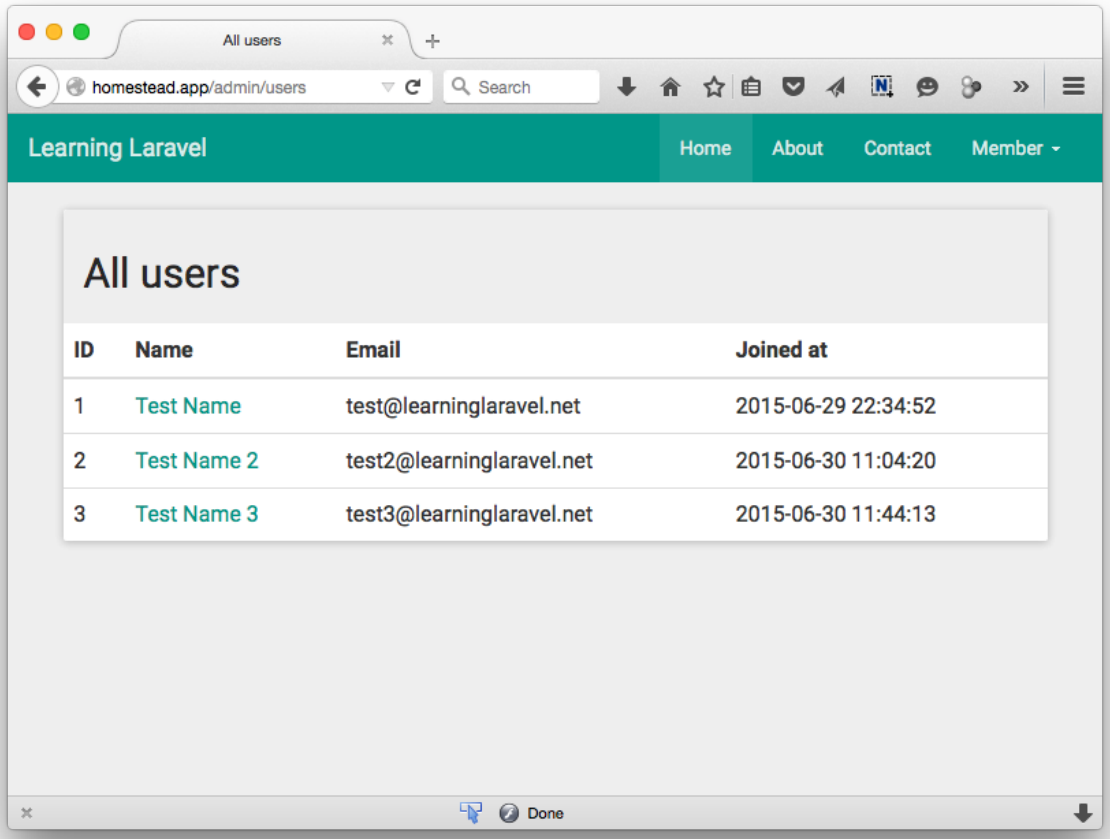
So the index view will be placed at **views/backend/users/index.blade.php**.

Here is the code:

```
@extends('master')
@section('title', 'All users')
@section('content')

    <div class="container col-md-8 col-md-offset-2">
        <div class="panel panel-default">
            <div class="panel-heading">
                <h2> All users </h2>
            </div>
            @if (session('status'))
                <div class="alert alert-success">
                    {{ session('status') }}
                </div>
            @endif
            @if ($users->isEmpty())
                <p> There is no user.</p>
            @else
                <table class="table">
                    <thead>
                    <tr>
```

```
                        <th>ID</th>
                        <th>Name</th>
                        <th>Email</th>
                        <th>Joined at</th>

                    </tr>
                    </thead>
                    <tbody>
                    @foreach($users as $user)
                        <tr>
                            <td>{!! $user->id !!}</td>
                            <td>
                                <a href="#">{!! $user->name !!} </a>
                            </td>
                            <td>{!! $user->email !!}</td>
                            <td>{!! $user->created_at !!}</td>
                        </tr>
                    @endforeach
                    </tbody>
                </table>
            @endif
        </div>
    </div>

@endsection
```

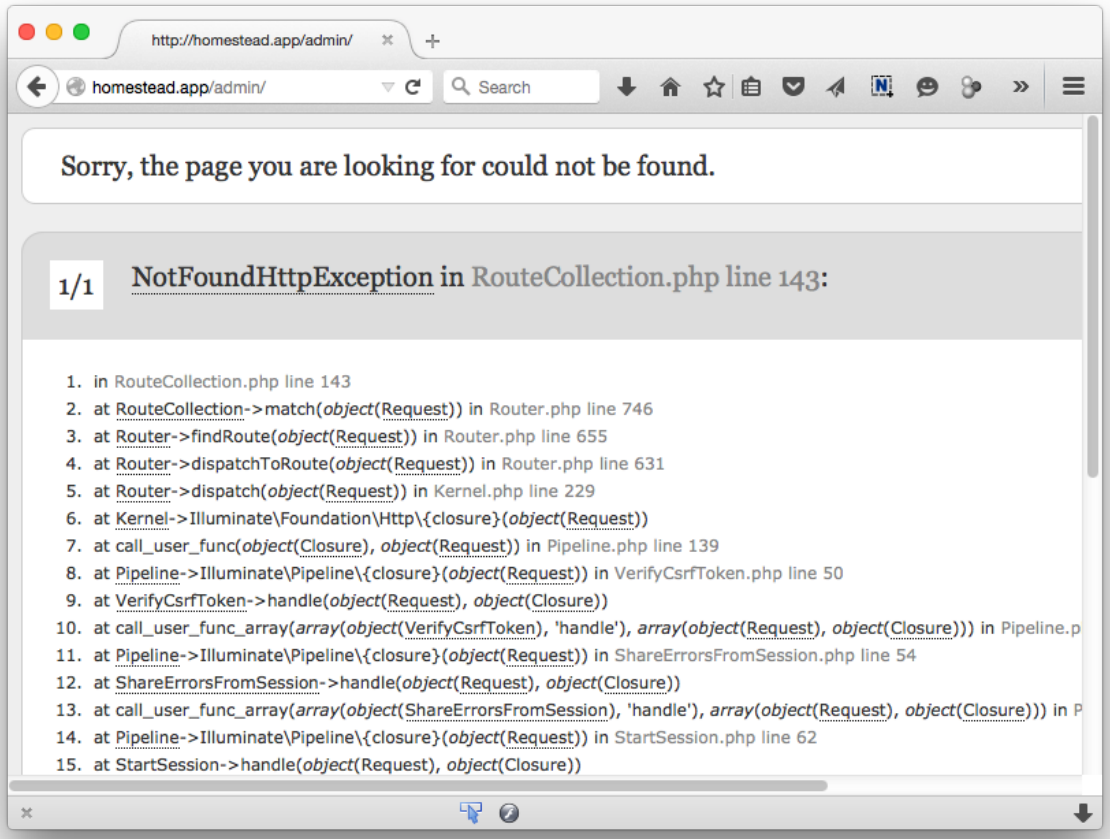
Now, be sure to login to our application and visit http://homestead.app/admin/users



View all users

Cool! We can be able to view all the users!

If you don't login and visit the page, you will get an error:



Error when viewing admin pages

Laravel will redirect you to http://homestead.app/auth/login as well.

Don't worry, it's because our **Middleware** are working fine. We will learn how to use **Middleware** and fix this bug in the next section.

All about Middleware

One of the best new features of Laravel 5 is **Middleware** (aka **HTTP Middleware**). Imagine that you have a layer between all requests and responses. That layer will help to handle all requests and return proper responses, even before the requests/responses are processed. We call the layer: "**Middleware**".

Take a look at the docs to learn more about it:

http://laravel.com/docs/master/middleware

We can use middleware to do many things. For example, middleware can help to authenticate users, log data for analytics, add CSRF protection, etc.

You can find all middleware in the **app/Http/Middleware** directory. Open the directory, you'll see four middleware:

1. Authenticated middleware: This middleware is responsible for authenticating users. If users are not logged in, they are redirected to the login page.
2. EncryptCookies middleware: Used to encrypt the application's cookies.
3. RedirectIfAuthenticated middleware: Redirect users to a page if they're not authenticated.
4. VerifyCsrfToken middleware: Used to manage RSRF tokens.

We've used the **Authenticated middleware** for our admin route group. Let's open it:

```
public function handle($request, Closure $next)
{
    if ($this->auth->guest()) {
        if ($request->ajax()) {
            return response('Unauthorized.', 401);
        } else {
            return redirect()->guest('auth/login');
        }
    }

    return $next($request);
}
```

As you see, this **Authenticated middleware** is just a class. We use the **handle method** to process all requests and define request filters.

By default, if users are not signed in, the middleware automatically redirects users to the **auth/login** URI.

```
return redirect()->guest('auth/login');
```

However, we use **users/login** route to access our login page. That's why Laravel doesn't understand where the **auth/login** route is, and it throws an error.

To fix the bug, we can just modify the line to:

```
 return redirect()->guest('users/login');
```

When you login with wrong credentials, you are redirected to the **users/login** route. Everything should be good to go.

Now let's say that we wanted to use a new middleware called **Manager** to make sure that only administrators can access the admin area.

Creating a new middleware

Creating a new middleware is easy, simply run this Artisan command:

```
php artisan make:middleware Manager
```

A new middleware called Manager will be created. You can find it in the **Middleware** directory.

```
<?php

namespace App\Http\Middleware;

use Closure;

class Manager
{
    public function handle($request, Closure $next)
    {
        return $next($request);
    }
}
```

One more step to do, we need to register the Manager middleware. Let's open the **Kernel.php** file, which can be found in the **Http** directory.

```
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \App\Http\Middleware\VerifyCsrfToken::class,
];

protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
];
```

There are two properties: **$middleware** and **$routeMiddleware**.

If you want to enable a middleware for every route, you can append it to the **$middleware** property. For example, you may add the Manager class like this:

```
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \App\Http\Middleware\VerifyCsrfToken::class,
   \App\Http\Middleware\Manager::class,
];
```

However, we just want to assign the **Manager middleware** to our admin route group. Therefore, all we need to do is append the Manager middleware to the **$routeMiddleware** property.

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'manager' => \App\Http\Middleware\Manager::class,
];
```

We can use **manager** as a **short-hand key** to reference the Manager middleware.

> **Note**:: Don't append the Manager middleware to the **$middleware** property. Only append the middleware to the **$routeMiddleware** property, because we only want to use it for our admin route group.

Next, open **routes.php** and modify the admin route group to enable the Manager middleware:

```
Route::group(array('prefix' => 'admin', 'namespace' => 'Admin', 'middleware' => \
'auth'), function () {
    Route::get('users', 'UsersController@index');
});
```

Change to:

```
Route::group(array('prefix' => 'admin', 'namespace' => 'Admin', 'middleware' => \
'manager'), function () {
    Route::get('users', 'UsersController@index');
});
```

Well done! You've got a solid foundation of Middleware. Keep in mind that you can use Middleware to do many things.

Even though our Manager middleware is working properly, we can't see any difference. The reason is, we don't create any request filters yet.

If we want to restrict access to the admin area, we need to add roles or permissons to our users. Fortunately, Laravel has a very popular package that can help us to implement the feature easily: **Entrust**.

Adding roles and permission to our app using Entrust

Many Laravel developers are using Entrust to add Role-based Permissions to their Laravel applications.You can find Entrust here:

https://github.com/Zizaco/entrust

Installing Entrust package for Laravel 5.2 (official package)

> **Note**: This section shows you how to install the official Entrust package for **Laravel 5.2**. If you use Laravel 5.0 or Laravel 5.1, please skip this section.

The official Entrust package now supports Laravel 5.2. If you have any issues, please submit your issues or view other issues at:

https://github.com/Zizaco/entrust/issues

Follow the steps below to install Entrust:

In order to install Entrust for Laravel 5.2, find the **require** section in your **composer.json** file, add this line:

```
    "zizaco/entrust": "5.2.x-dev"
```

Your **composer.json** file should look like this:

```
"require": {
    "php": ">=5.5.9",
    "laravel/framework": "5.2.*",
    "laravelcollective/html": "5.2.*",
    "zizaco/entrust": "5.2.x-dev"
},
```

Next, run **composer update** to install Entrust.

Now, open the **config/app.php** file and find the **providers** array, add:

```
Zizaco\Entrust\EntrustServiceProvider::class,
```

Then find the **aliases** array, add:

```
'Entrust'   => Zizaco\Entrust\EntrustFacade::class,
```

If you are going to use **Middleware** (requires Laravel 5.1+), find **routeMiddleware** array in **app/Http/Kernel.php** and add:

```
'role' => \Zizaco\Entrust\Middleware\EntrustRole::class,
'permission' => \Zizaco\Entrust\Middleware\EntrustPermission::class,
'ability' => \Zizaco\Entrust\Middleware\EntrustAbility::class,
```

Run this command to create a new **entrust.php** file, which can be found in the **config** directory.

```
php artisan vendor:publish
```

You may use this file to customize table names and model namespaces.

Entrust need some database tables to work. Run this command to generate the migrations:

```
php artisan entrust:migration
```

Say yes when you're asked to confirm.

Run this command to create Entrust tables:

```
php artisan migrate
```

You now have four tables:

- roles: stores role records
- permissions: stores permission records
- role_user: stores relations between roles and users
- permission_role: stores relations between roles and permissions

Next, we need to create two models: Role and Permission.

Create a new file called **Role.php**, place it inside the app directory.

```
<?php namespace App;

use Zizaco\Entrust\EntrustRole;

class Role extends EntrustRole
{
}
```

Create a new file called **Permission.php**, place it inside the app directory.

```
<?php namespace App;

use Zizaco\Entrust\EntrustPermission;

class Permission extends EntrustPermission
{
}
```

Finally, open **app/User.php** and replace this:

```
<?php

namespace App;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
```

with the following:

```
<?php

namespace App;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;
use Zizaco\Entrust\Traits\EntrustUserTrait;
```

```
class User extends Model implements AuthenticatableContract, CanResetPasswordCon\
tract
{
    use Authenticatable, CanResetPassword;
    use EntrustUserTrait;
```

Let's take a look at this line:

```
use EntrustUserTrait;
```

This is **Entrust trait**. When you add it to the **User** model, you can use the following methods: **roles()**, **hasRole($name)**, **can($permission)**, and **ability($roles, $permissions, $options)**.

You should have something like this:

```
<?php

namespace App;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;
use Zizaco\Entrust\Traits\EntrustUserTrait;

class User extends Model implements AuthenticatableContract, CanResetPasswordCon\
tract
{
    use Authenticatable, CanResetPassword;
    use EntrustUserTrait;

    /**
     * The database table used by the model.
     *
     * @var string
     */
    protected $table = 'users';
```

When you add a new file, you need to run this command to rebuild the class map:

```
composer dump-autoload
```

Good job! You've installed the **Entrust package** for Laravel 5.2!

Installing Entrust package for Laravel 5.2 (different branch)

> **Note**: This section shows you how to install Entrust (different branch) for **Laravel 5.2**. If you use an older version of Laravel or you've installed Entrust, please skip this section.

The official Entrust package doesn't support Laravel 5.2 yet. We have to install a different branch of Entrust to make it work properly. When the official Entrust package is updated, you may use the official version (check the next section).

In order to install Entrust for Laravel 5.2, find the **require** section in your **composer.json** file, add this line:

```
    "zizaco/entrust": "dev-laravel-5-2@dev"
```

Your **composer.json** file should look like this:

```
"require": {
    "php": ">=5.5.9",
    "laravel/framework": "5.2.*",
    "laravelcollective/html": "5.2.*",
    "zizaco/entrust": "dev-laravel-5-2@dev"
},
```

Add above:

```
"repositories": [
    {
        "type": "vcs",
        "url": "https://github.com/hiendv/entrust"
    }
],
```

Next, run **composer update** to install Entrust.

Now, open the **config/app.php** file and find the **providers** array, add:

```
'Zizaco\Entrust\EntrustServiceProvider',
```

Then find the **aliases** array, add:

```
'Entrust' => 'Zizaco\Entrust\EntrustFacade',
```

If you are going to use **Middleware** (requires Laravel 5.1+), find **routeMiddleware** array in **app/Http/Kernel.php** and add:

```
'role' => Zizaco\Entrust\Middleware\EntrustRole::class,
'permission' => Zizaco\Entrust\Middleware\EntrustPermission::class,
'ability' => Zizaco\Entrust\Middleware\EntrustAbility::class,
```

Run this command to create a **entrust.php** file, which can be found in the **config** directory.

```
php artisan vendor:publish
```

You may use this file to customize table names and model namespaces.

Entrust need some database tables to work. Run this command to generate the migrations:

```
php artisan entrust:migration
```

Say yes when you're asked to confirm.

Run this command to create Entrust tables:

```
php artisan migrate
```

You now have four tables:

- roles: stores role records
- permissions: stores permission records
- role_user: stores relations between roles and users
- permission_role: stores relations between roles and permissions

Next, we need to create two models: Role and Permission.

Create a new file called **Role.php**, place it inside the app directory.

```
<?php namespace App;

use Zizaco\Entrust\EntrustRole;

class Role extends EntrustRole
{
}
```

Create a new file called **Permission.php**, place it inside the app directory.

```
<?php namespace App;

use Zizaco\Entrust\EntrustPermission;

class Permission extends EntrustPermission
{
}
```

Finally, open **app/User.php** and replace this:

```
<?php

namespace App;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
```

with the following:

```
<?php

namespace App;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;
use Zizaco\Entrust\Traits\EntrustUserTrait;
```

```
class User extends Model implements AuthenticatableContract, CanResetPasswordCon\
tract
{
    use Authenticatable, CanResetPassword;
    use EntrustUserTrait;
```

Let's take a look at this line:

```
use EntrustUserTrait;
```

This is **Entrust trait**. When you add it to the **User** model, you can use the following methods: **roles()**, **hasRole($name)**, **can($permission)**, and **ability($roles, $permissions, $options)**.

You should have something like this:

```
<?php

namespace App;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;
use Zizaco\Entrust\Traits\EntrustUserTrait;

class User extends Model implements AuthenticatableContract, CanResetPasswordCon\
tract
{
    use Authenticatable, CanResetPassword;
    use EntrustUserTrait;

    /**
     * The database table used by the model.
     *
     * @var string
     */
    protected $table = 'users';
```

When you add a new file, you need to run this command to rebuild the class map:

```
composer dump-autoload
```

Good job! You've installed the **Entrust package** for Laravel 5.2!

Installing Entrust package (for Laravel 5.1)

> **Note**: This section shows you how to install Entrust for **Laravel 5.1 or Laravel 5.0**. If you use an older version of Laravel or you've installed Entrust, please skip this section.

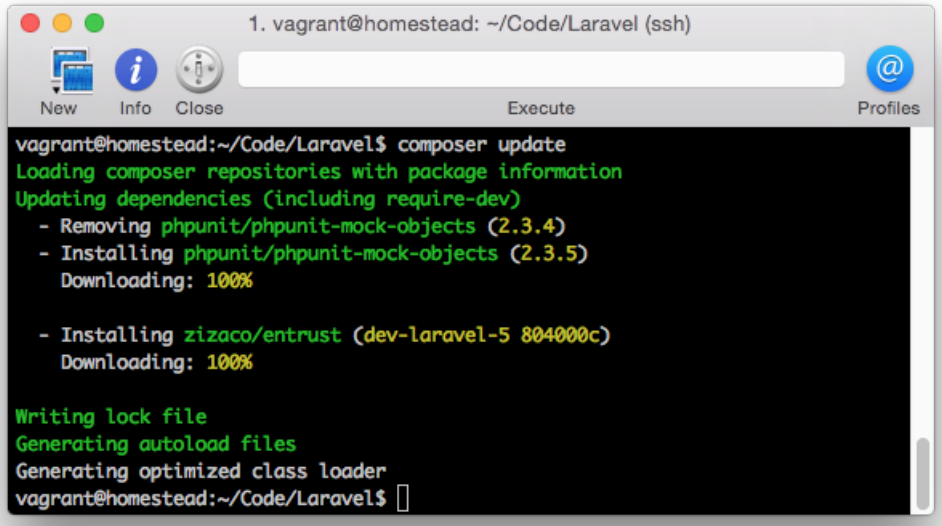
In order to install Entrust, we just need to add this line to the **composer.json** file:

```
"zizaco/entrust": "dev-laravel-5"
```

You should have something like this:

```
"require": {
    "php": ">=5.5.9",
    "laravel/framework": "5.1.*",
    "laravelcollective/html": "5.1.*",
   "zizaco/entrust": "dev-laravel-5"
},
```

Next, run **composer update** to install Entrust.



Installing Entrust

Now, open the **config/app.php** file and find the **providers** array, add:

```
'Zizaco\Entrust\EntrustServiceProvider',
```

Then find the **aliases** array, add:

```
'Entrust' => 'Zizaco\Entrust\EntrustFacade',
```

If you are going to use **Middleware** (requires Laravel 5.1+), find **routeMiddleware** array in **app/Http/Kernel.php** and add:

```
'role' => Zizaco\Entrust\Middleware\EntrustRole::class,
'permission' => Zizaco\Entrust\Middleware\EntrustPermission::class,
'ability' => Zizaco\Entrust\Middleware\EntrustAbility::class,
```

Run this command to create a **entrust.php** file, which can be found in the **config** directory.

```
php artisan vendor:publish
```

You may use this file to customize table names and model namespaces.

Entrust need some database tables to work. Run this command to generate the migrations:

```
php artisan entrust:migration
```

Say yes when you're asked to confirm.

Run this command to create Entrust tables:

```
php artisan migrate
```

You now have four tables:

- roles: stores role records
- permissions: stores permission records
- role_user: stores relations between roles and users
- permission_role: stores relations between roles and permissions

Next, we need to create two models: Role and Permission.

Create a new file called **Role.php**, place it inside the app directory.

```
<?php namespace App;

use Zizaco\Entrust\EntrustRole;

class Role extends EntrustRole
{
}
```

Create a new file called **Permission.php**, place it inside the app directory.

```
<?php namespace App;

use Zizaco\Entrust\EntrustPermission;

class Permission extends EntrustPermission
{
}
```

Finally, open **app/User.php** and replace this:

```
<?php

namespace App;

// ...

class User extends Model implements AuthenticatableContract, CanResetPasswordCon\
tract

{
```

with the following:

```
<?php

namespace App;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;
use Zizaco\Entrust\Traits\EntrustUserTrait;

class User extends Model implements AuthenticatableContract, CanResetPasswordCon\
tract
{
    use Authenticatable, CanResetPassword;
    use EntrustUserTrait;
```

Let's take a look at this line:

```
use EntrustUserTrait;
```

This is **Entrust trait**. When you add it to the **User** model, you can use the following methods: **roles()**, **hasRole($name)**, **can($permission)**, and **ability($roles, $permissions, $options)**.

You should have something like this:

```
<?php

namespace App;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;
use Zizaco\Entrust\Traits\EntrustUserTrait;

class User extends Model implements AuthenticatableContract, CanResetPasswordCon\
tract
{
    use Authenticatable, CanResetPassword;
    use EntrustUserTrait;
```

```
    /**
     * The database table used by the model.
     *
     * @var string
     */
    protected $table = 'users';
```

When you add a new file, you need to run this command to rebuild the class map:

```
composer dump-autoload
```

Good job! You've installed the **Entrust package**!

Create Entrust roles

Once installed, we can be able to create user roles. Let's say we will have two roles: **manager** and **member**.

As an exercise let's create a simple **role creation form**:

First, edit the **routes.php** file and update the admin route group as follows:

```
Route::group(array('prefix' => 'admin', 'namespace' => 'Admin', 'middleware' => \
'manager'), function () {
    Route::get('users', [ 'as' => 'admin.user.index', 'uses' => 'UsersController\
@index']);
    Route::get('roles', 'RolesController@index');
    Route::get('roles/create', 'RolesController@create');
    Route::post('roles/create', 'RolesController@store');
});
```

We define routes to view all roles and create a new role.

Again, create **RolesController** by running this command:

```
php artisan make:controller Admin/RolesController
```

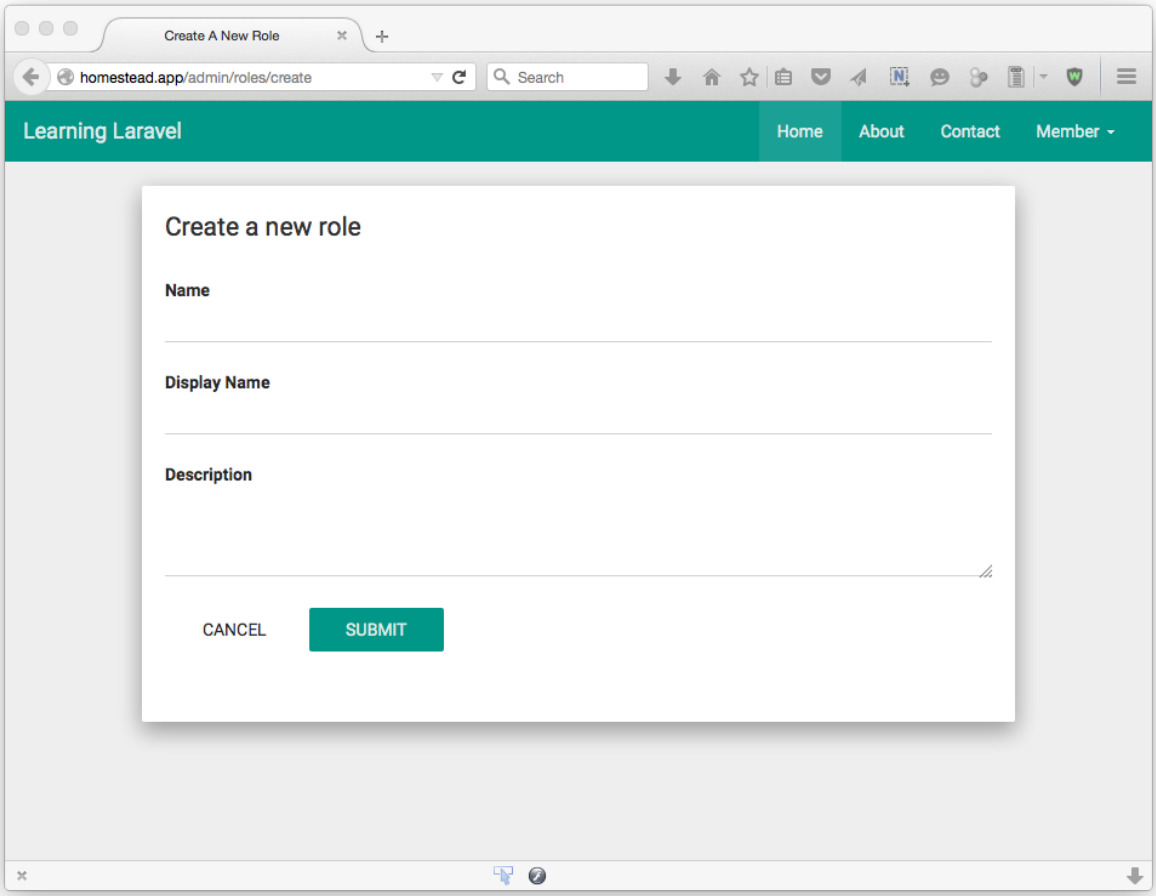
Open RolesController, update the **create action** as follows:

```
public function create()
{
    return view('backend.roles.create');
}
```

Create a new **roles** directory, place it inside the **views/backend** directory. Then create a new view called **create**:

```
@extends('master')
@section('title', 'Create A New Role')

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="well well bs-component">

            <form class="form-horizontal" method="post">

                @foreach ($errors->all() as $error)
                    <p class="alert alert-danger">{{ $error }}</p>
                @endforeach

                @if (session('status'))
                    <div class="alert alert-success">
                        {{ session('status') }}
                    </div>
                @endif

                <input type="hidden" name="_token" value="{!! csrf_token() !!}">

                <fieldset>
                    <legend>Create a new role</legend>
                    <div class="form-group">
                        <label for="name" class="col-lg-2 control-label">Name</l\
abel>
                        <div class="col-lg-10">
                            <input type="text" class="form-control" id="name" na\
me="name">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="display_name" class="col-lg-2 control-label"\
```

```
>Display Name</label>
                        <div class="col-lg-10">
                            <input type="display_name" class="form-control" id="\
display_name" name="display_name">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="description" class="col-lg-2 control-label">\
Description</label>
                        <div class="col-lg-10">
                            <textarea class="form-control" rows="3" id="descript\
ion" name="description"></textarea>
                        </div>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button type="reset" class="btn btn-default">Cancel<\
/button>
                            <button type="submit" class="btn btn-primary">Submit\
</button>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
    </div>
@endsection
```



Create a new role

We should have a nice role creation form at http://homestead.app/admin/roles/create.

Next, let's update the **store** action of the **RolesController** to save the data.

```
public function store(RoleFormRequest $request)
{
    $role = new Role(array(
        'name' => $request->get('name'),
        'display_name' => $request->get('display_name'),
        'description' => $request->get('description')
    ));

    $role->save();

    return redirect('/admin/roles/create')->with('status', 'A new role has been \
```

```
created!');
}
```

We use **RoleFormRequest** here to validate the form, but we don't have the request file yet. Let's create it:

```
php artisan make:request RoleFormRequest
```

Open it and find:

```
public function authorize()
{
    return false;
}
```

Update to:

```
public function authorize()
{
    return true;
}
```

Here is the rule:

```
public function rules()
{
    return [
        'name' => 'required',
    ];
}
```

The **Display Name** and **Description** fields are optional. We only need to require users to enter the **role's name**.

Be sure that you have told Laravel that you want to use **Role** and RoleFormRequest in the RolesController:

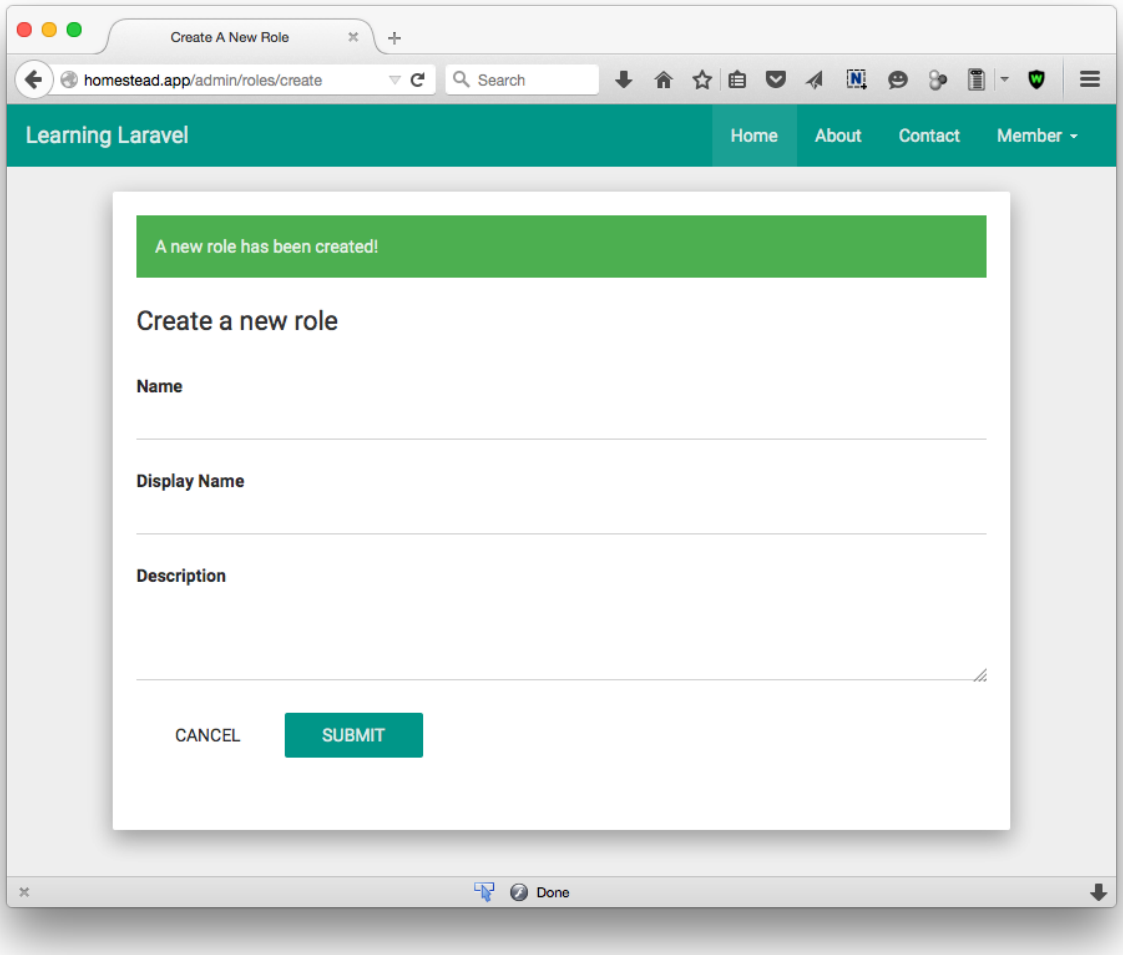
```
use App\Role;
use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;
use App\Http\Requests\RoleFormRequest;
```

Lastly, Open **Role.php** and add:

```
class Role extends EntrustRole
{
    protected $fillable = ['name', 'display_name', 'description'];
}
```

The **$fillable** property makes the columns mass assignable.

Now, go to http://homestead.app/admdin/roles/create and create two new roles: **manager** and **member**.



Create a role successfully

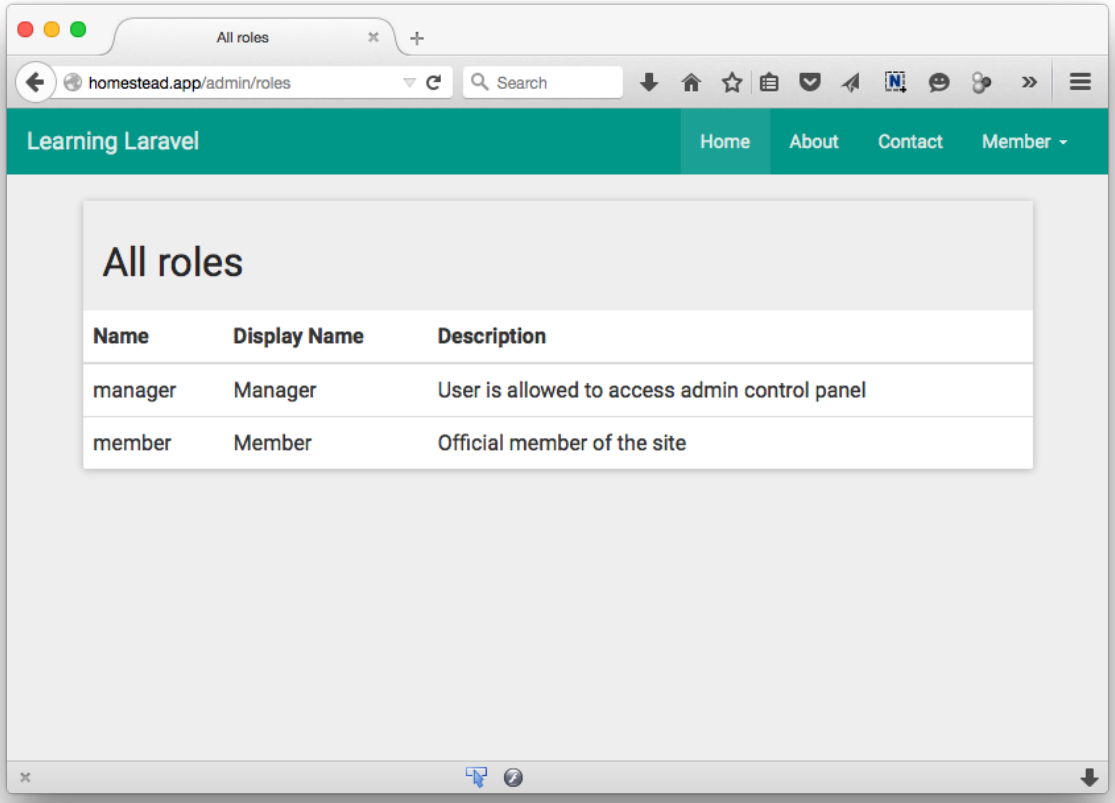
To view all roles, update the **index action** of our RolesController as follows:

```
public function index()
{
    $roles = Role::all();
    return view('backend.roles.index', compact('roles'));
}
```

Then create a new **index** view at **views/backend/roles/index.blade.php**:

```
@extends('master')
@section('title', 'All roles')
@section('content')

    <div class="container col-md-8 col-md-offset-2">
        <div class="panel panel-default">
            <div class="panel-heading">
                <h2> All roles </h2>
            </div>
            @if (session('status'))
                <div class="alert alert-success">
                    {{ session('status') }}
                </div>
            @endif
            @if ($roles->isEmpty())
                <p> There is no role.</p>
            @else
                <table class="table">
                    <thead>
                    <tr>
                        <th>Name</th>
                        <th>Display Name</th>
                        <th>Description</th>
                    </tr>
                    </thead>
                    <tbody>
                    @foreach($roles as $role)
                        <tr>
                            <td>{!! $role->name !!}</td>
                            <td>{!! $role->display_name !!}</td>
                            <td>{!! $role->description !!}</td>

                        </tr>
                    @endforeach
                    </tbody>
                </table>
            @endif
        </div>
    </div>

@endsection
```

Go to http://homestead.app/admin/roles. You should see a list of roles.



Role list

Assign roles to users

In this section, we will learn how to edit users and assign roles to them.

Let's start by adding these routes to our **admin route group**:

```
Route::get('users/{id?}/edit', 'UsersController@edit');
Route::post('users/{id?}/edit','UsersController@update');
```

Next, open users/index view and find:

```
<a href="#">{!! $user->name !!} </a>
```

Update the link to:

```
<a href="{!! action('Admin\UsersController@edit', $user->id) !!}">{!! $user->nam\
e !!} </a>
```

Open **UsersController**, update the **edit** action:

```
public function edit($id)
{
    $user = User::whereId($id)->firstOrFail();
    $roles = Role::all();
    $selectedRoles = $user->roles->lists('id')->toArray();
    return view('backend.users.edit', compact('user', 'roles', 'selectedRoles'));
}
```

All we need to do is find a correct user using the user id and list all the roles for users to select.

The **$selectedRoles** is an array that holds the current role's IDs of users.

As you see, we use the Role model here. Don't forget to add this line at the top of the UsersController:

```
use App\Role;
```

Here is the **users/edit** view:

```
@extends('master')
@section('name', 'Edit a user')

@section('content')
    <div class="container col-md-6 col-md-offset-3">
        <div class="well well bs-component">

            <form class="form-horizontal" method="post">

                @foreach ($errors->all() as $error)
                    <p class="alert alert-danger">{{ $error }}</p>
                @endforeach

                @if (session('status'))
                    <div class="alert alert-success">
                        {{ session('status') }}
                    </div>
                @endif
```

```
                {!! csrf_field() !!}

                <fieldset>
                    <legend>Edit user</legend>
                    <div class="form-group">
                        <label for="name" class="col-lg-2 control-label">Name</l\
abel>

                        <div class="col-lg-10">
                            <input type="text" class="form-control" id="name" pl\
aceholder="Name" name="name"
                                   value="{{ $user->name }}">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="email" class="col-lg-2 control-label">Email<\
/label>

                        <div class="col-lg-10">
                            <input type="email" class="form-control" id="email" \
placeholder="Email" name="email"
                                   value="{{ $user->email }}">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="select" class="col-lg-2 control-label">Role<\
/label>

                        <div class="col-lg-10">
                            <select class="form-control" id="role" name="role[]"\
 multiple>
                                @foreach($roles as $role)
                                    <option value="{!! $role->id !!}"  @if(in_ar\
ray($role->id, $selectedRoles))
                                            selected="selected" @endif >{!! $rol\
e->display_name !!}
                                    </option>
                                @endforeach
                            </select>
                        </div>
```

```
                    </div>

                    <div class="form-group">
                        <label for="password" class="col-lg-2 control-label">Pas\
sword</label>

                        <div class="col-lg-10">
                            <input type="password" class="form-control" name="pa\
ssword">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="password" class="col-lg-2 control-label">Con\
firm password</label>

                        <div class="col-lg-10">
                            <input type="password" class="form-control" name="pa\
ssword_confirmation">
                        </div>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button type="reset" class="btn btn-default">Cancel<\
/button>
                            <button type="submit" class="btn btn-primary">Submit\
</button>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
    </div>
@endsection
```

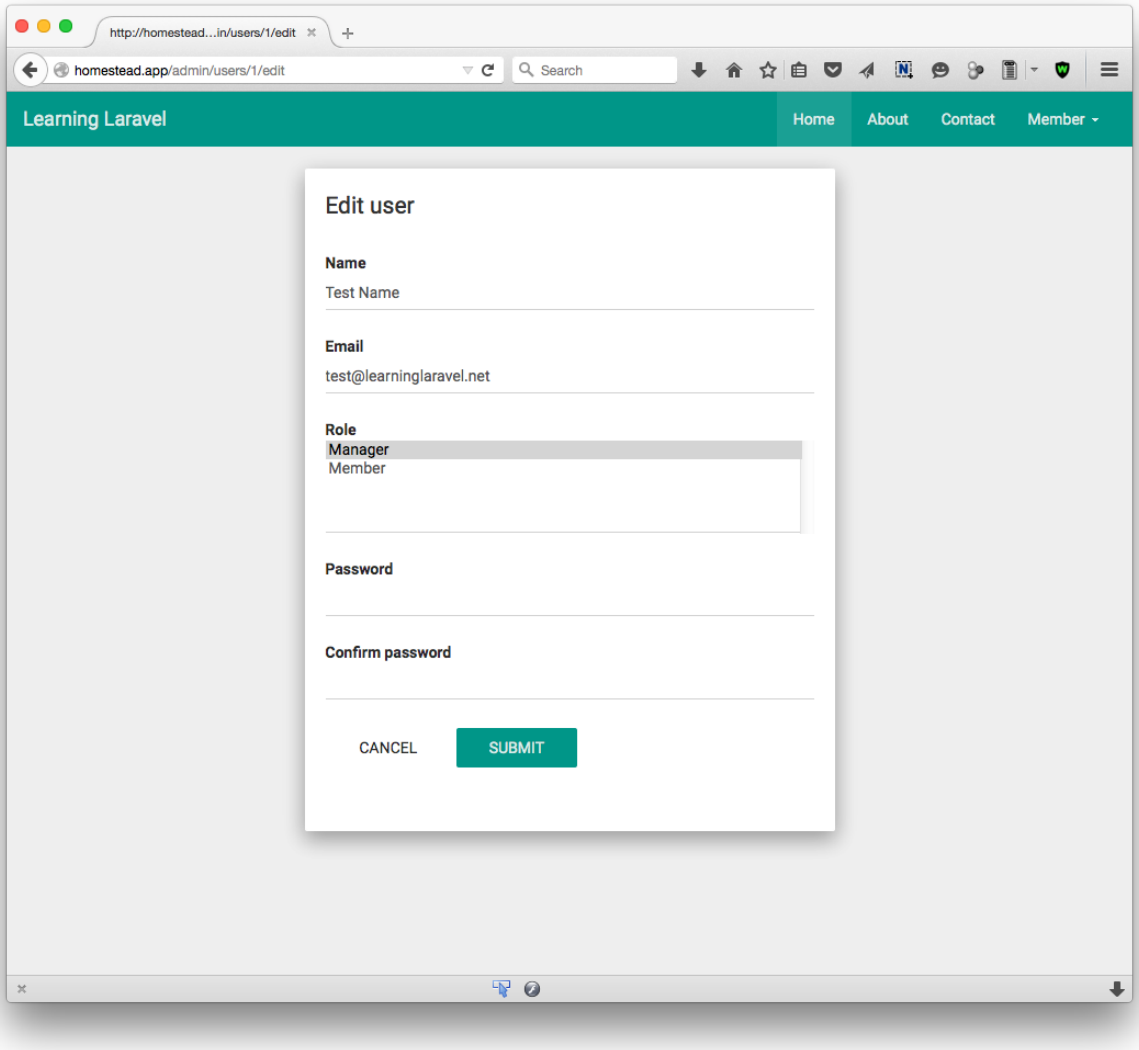
Let's look at this code:

```
<select class="form-control" id="role" name="role[]" multiple>
    @foreach($roles as $role)
        <option value="{!! $role->id !!}"  @if(in_array($role->id, $selectedRole\
s)) selected="selected" @endif >
                {!! $role->display_name !!}
        </option>
    @endforeach
</select>
```

This will display a multiple select box for us. We use @foreach to iterate over $roles and display the select options.

To display which option is selected, we use **in_array** function to check if $selectedRoles array contains the role's id.

Save the changes and go to http://homestead.app/admin/users. Click on the name of a user that you want to edit.



Edit user form

If you click the **Submit button** now, nothing will happen. We have to edit the **update** action of UsersController to save data to our database:

```
public function update($id, UserEditFormRequest $request)
{
    $user = User::whereId($id)->firstOrFail();
    $user->name = $request->get('name');
    $user->email = $request->get('email');
    $password = $request->get('password');
    if($password != "") {
        $user->password = Hash::make($password);
    }
    $user->save();
    $user->saveRoles($request->get('role'));

    return redirect(action('Admin\UsersController@edit', $user->id))->with('stat\
us', 'The user has been updated!');
}
```

We use user's id to find the user and then save the changes to the database using **$user->save()** method.

Notice how we handle the password:

```
    $password = $request->get('password');
    if($password != "") {
        $user->password = Hash::make($password);
    }
```

First, we check if the password is empty. We only save the password when users enter a new one, using:

```
$user->password = Hash::make($password);
```

This will create a **hashed password**. Before saving a password to the database, remember that you must **hash** it using the **Hash::make()** method.

For more information, visit:

http://laravel.com/docs/master/hashing#introduction

We'll need to include the **Hash** facade and **UserEditFormRequest** as well. Find:

```
class UsersController extends Controller
```

Add above:

```
use App\Http\Requests\UserEditFormRequest;
use Illuminate\Support\Facades\Hash;
```

You may notice that there is a new **saveRoles()** method here:

```
$user->saveRoles($request->get('role'));
```

Unfortunately, Entrust doesn't have any method to automatically sync (attach and detach) multiple roles. Therefore, we have to create a new **saveRoles** method to handle this scenario:

Open the **User.php** model, add:

```
public function saveRoles($roles)
{
    if(!empty($roles))
    {
        $this->roles()->sync($roles);
    } else {
        $this->roles()->detach();
    }
}
```

Basically, this method will retrieve the **$roles** array, which contains roles' ID, and attach the appropriate roles to the user. If there is no role, it will detach the role from the user.

As always, generate the **UserEditFormRequest** by running this command:

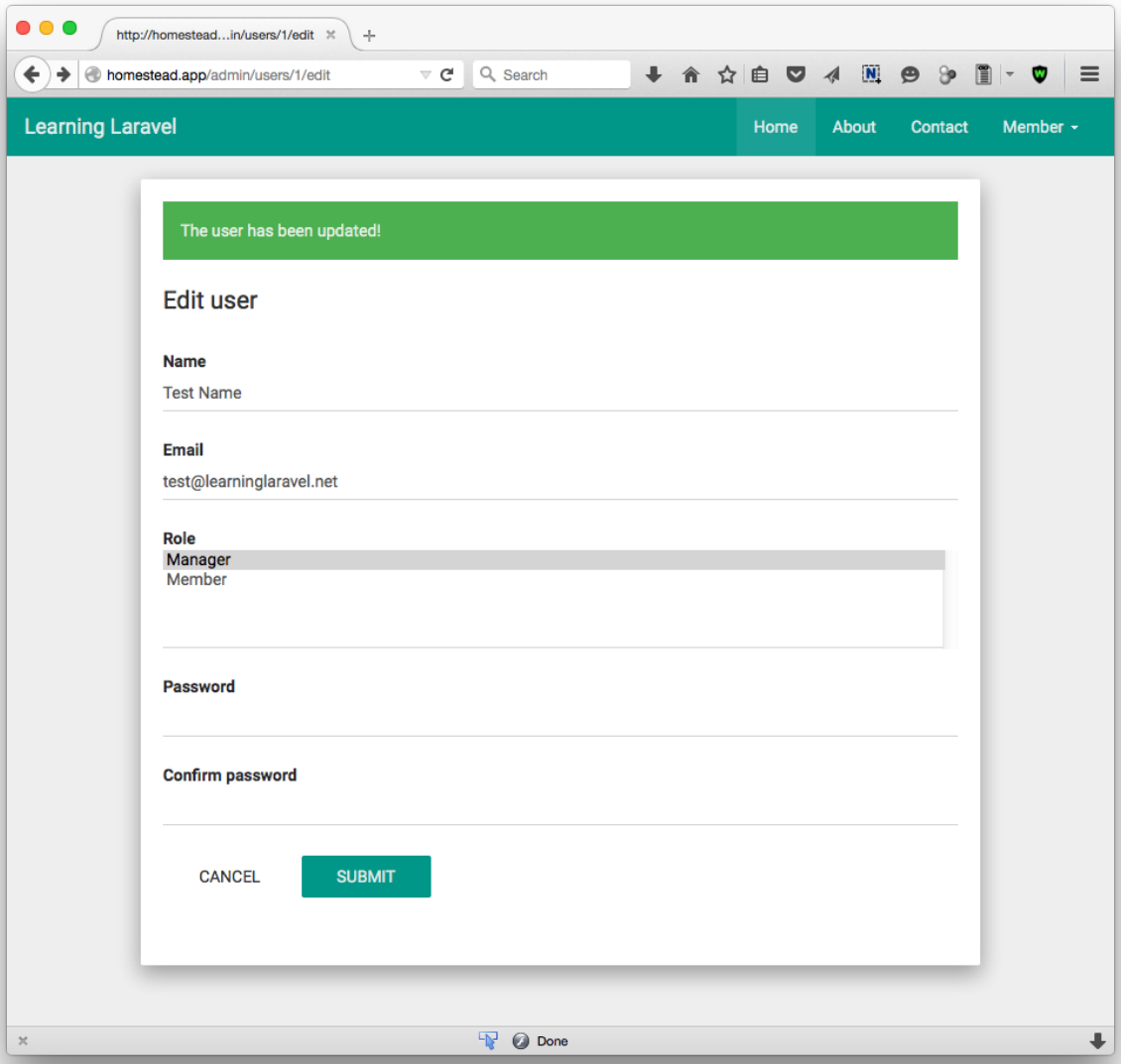
```
php artisan make:request UserEditFormRequest
```

Be sure to change **return false** to **true**. Here are the rules:

```
<?php

namespace App\Http\Requests;

use App\Http\Requests\Request;

class UserEditFormRequest extends Request
{
    public function authorize()
    {
        return true;
```

```
    }

    public function rules()
    {
        return [
            'name' => 'required',
            'email'=> 'required',
            'role'=> 'required',
            'password'=>'alpha_num|min:6|confirmed',
            'password_confirmation'=>'alpha_num|min:6',
        ];
    }
}
```

Sweet! Now let's try to assign a role to a user!



Edit a user

Restrict access to Manager users

Absolutely, we don't want anyone to access the admin area, except our Manager users. Open our **Manager** middleware:

```
public function handle($request, Closure $next)
{
    return $next($request);
}
```

As you see, there is no filter here, everyone can access the admin area.

There are many ways to restrict access, but the easiest way is using the **Auth** facade:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\Auth;

class Manager
{
    public function handle($request, Closure $next)
    {
        if(!Auth::check()) {
            return redirect('users/login');
        } else {
            $user = Auth::user();
            if($user->hasRole('manager'))
            {
                return $next($request);
            } else {
                return redirect('/home');
            }
        }
    }
}
```

Let's see the code line by line.

First, we tell Laravel that we want to use the **Auth** facade:

```
use Illuminate\Support\Facades\Auth;
```

Then we use **Auth:check()** method to check if the user is logged in. If not, then the user is redirected to the login page.

```
if(!Auth::check()) {
    return redirect('users/login');
} else {
```

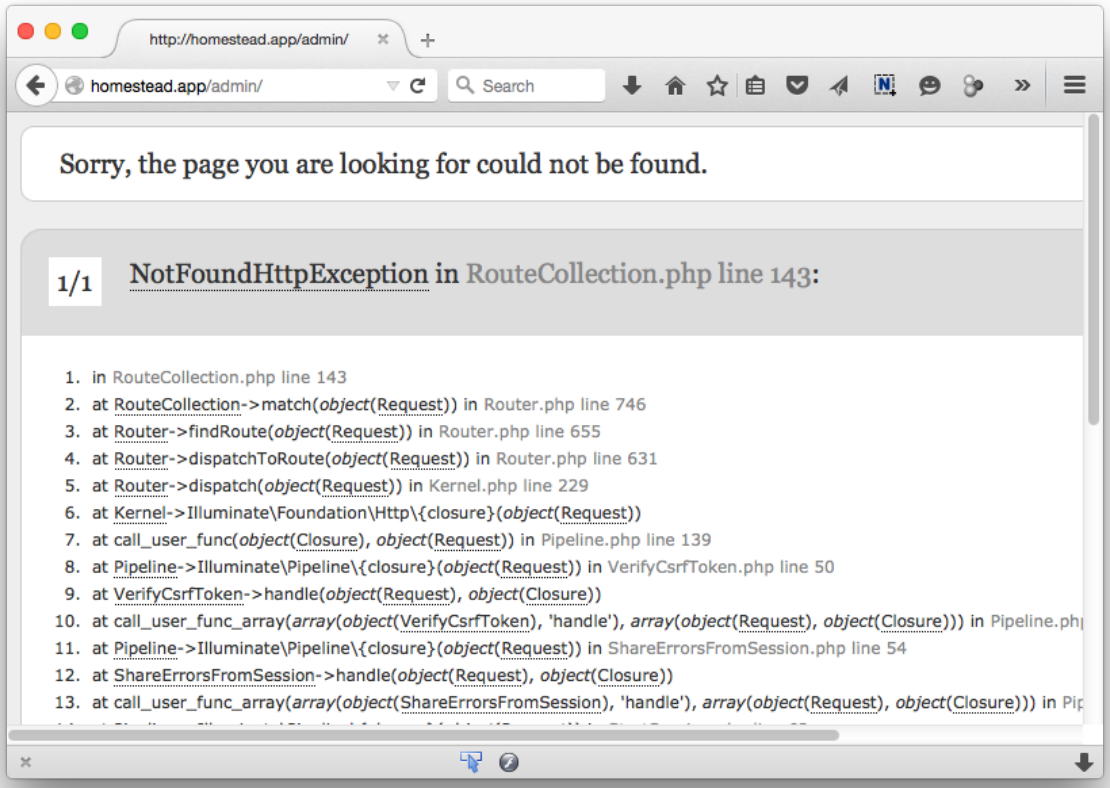
Otherwise, we use **Auth::user()** method to retrieve the authenticated user. This way, we can check if the user has the manager role. If not, the user is redirected back to the home page.

```
$user = Auth::user();
if($user->hasRole('manager'))
{
    return $next($request);
} else {
    return redirect('/home');
}
```

Now, try to access the admin area to test our Manager middleware. If you don't sign in and you're not a manager, you can't access any routes of the admin route group.

Create an admin dashboard page

Since we haven't created the admin home page, when we access http://homestead.app/admin, there is an error:



Error when accessing the admin home page

Let's move onto creating the admin home page now so that we can access the admin area easily. Update our **routes.php** file, add this route into the admin route group:

```
Route::get('/', 'PagesController@home');
```

Next, create the **Admin/PagesController**:

```
php artisan make:controller Admin/PagesController
```

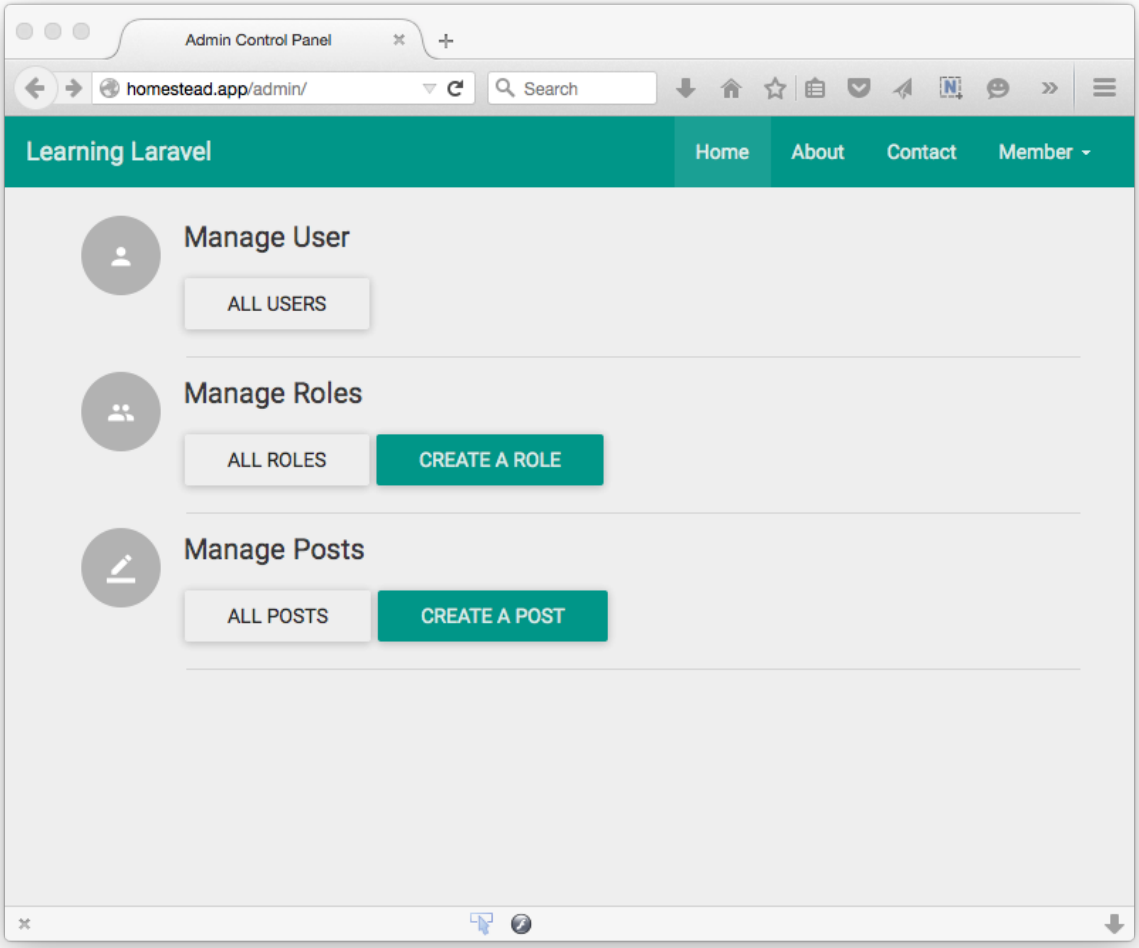
Remove all the default actions and update the **Admin/PagesController** as follows:

```
<?php

namespace App\Http\Controllers\Admin;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

class PagesController extends Controller
{
    public function home()
    {
        return view('backend.home');
    }
}
```

Finally, create a new **home** view (home.blade.php) in the **backend** directory:

```
@extends('master')
@section('title', 'Admin Control Panel')

@section('content')

    <div class="container">
        <div class="row banner">

            <div class="col-md-12">

                <div class="list-group">
                    <div class="list-group-item">
                        <div class="row-action-primary">
                            <i class="mdi-social-person"></i>
                        </div>
                        <div class="row-content">
                            <div class="action-secondary"><i class="mdi-social-i\
nfo"></i></div>
                            <h4 class="list-group-item-heading">Manage User</h4>
                                <a href="/admin/users" class="btn btn-default bt\
n-raised">All Users</a>
                        </div>
                    </div>
```

```
                    <div class="list-group-separator"></div>
                    <div class="list-group-item">
                        <div class="row-action-primary">
                            <i class="mdi-social-group"></i>
                        </div>
                        <div class="row-content">
                            <div class="action-secondary"><i class="mdi-material\
-info"></i></div>
                            <h4 class="list-group-item-heading">Manage Roles</h4>
                            <a href="/admin/roles" class="btn btn-default btn-ra\
ised">All Roles</a>
                            <a href="/admin/roles/create" class="btn btn-primary\
 btn-raised">Create A Role</a>
                        </div>
                    </div>
                    <div class="list-group-separator"></div>
                    <div class="list-group-item">
                        <div class="row-action-primary">
                            <i class="mdi-editor-border-color"></i>
                        </div>
                        <div class="row-content">
                            <div class="action-secondary"><i class="mdi-material\
-info"></i></div>
                            <h4 class="list-group-item-heading">Manage Posts</h4>
                            <a href="/admin/posts" class="btn btn-default btn-ra\
ised">All Posts</a>
                            <a href="/admin/posts/create" class="btn btn-primary\
 btn-raised">Create A Post</a>
                        </div>
                    </div>
                    <div class="list-group-separator"></div>
                </div>

            </div>

        </div>
    </div>

@endsection
```



Admin home page

I just add some buttons to access other admin pages here. Feel free to change the layout to your liking.

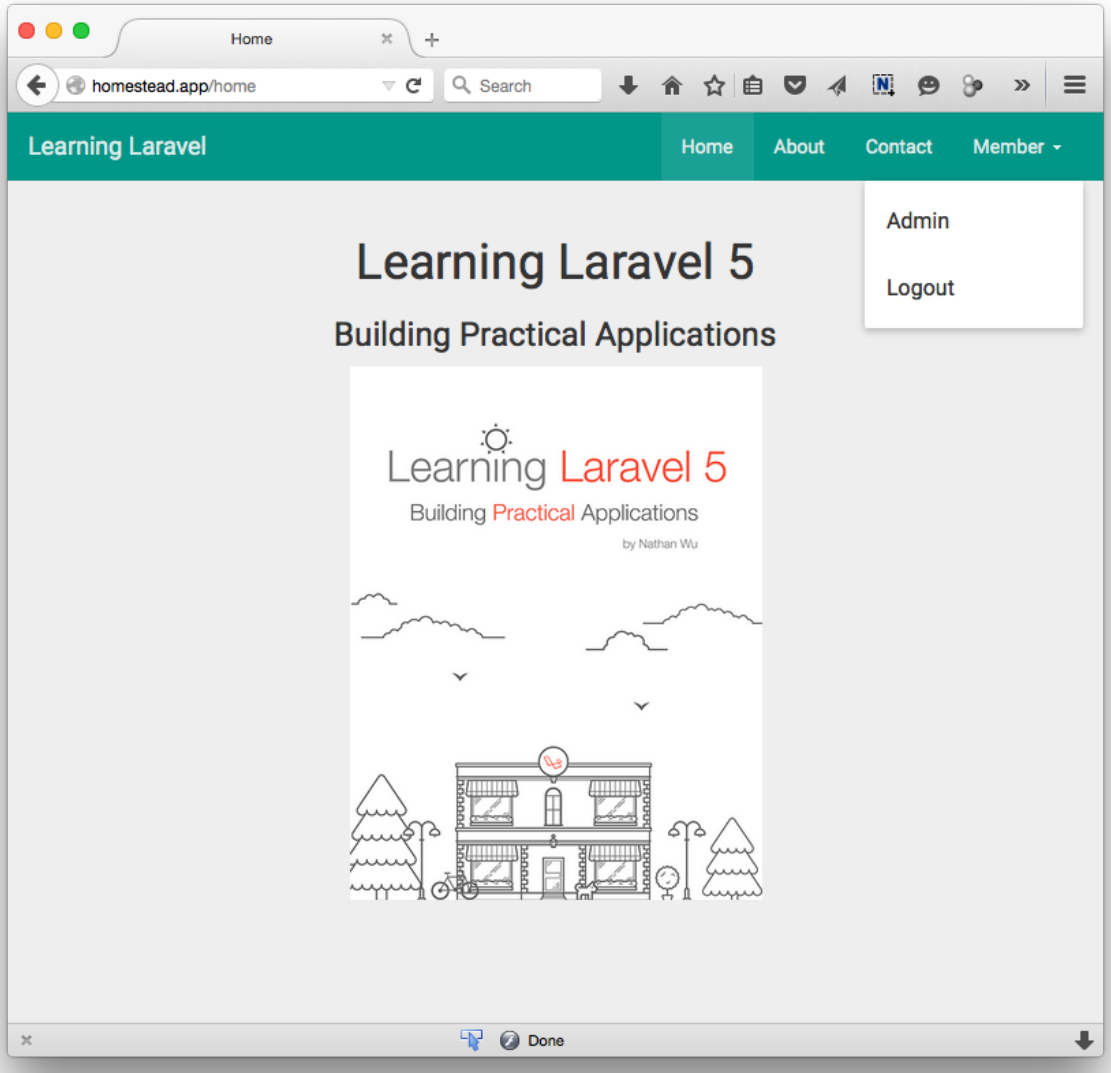
One more thing, how about adding a link to our navigation bar to access the admin area easier? Open the **shared/navbar** view and find:

```
@if (Auth::check())
    <li><a href="/users/logout">Logout</a></li>
@else
```

Update to:

```
@if (Auth::check())
    @if(Auth::user()->hasRole('manager'))
        <li><a href="/admin">Admin</a></li>
    @endif
    <li><a href="/users/logout">Logout</a></li>
@else
```

We can use the **hasRole** method to check if the user is a manager in **views** as well.



Admin link

Create a new post

Now that we have everything in order, we're going to build a form to create blog posts in this section.

Actually, the process is very similar to what we've done to create users, tickets, and roles.

To get started, let's create a new **Post** model and its migration:

```
php artisan make:model Post -m
```

You can also generate the post's migration at the same time by adding the **-m** option.

Open **timestamp_create_posts_table.php**, which can be found in the **migrations** directory. Update the up method as follows:

```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->increments('id');
        $table->string('title', 255);
        $table->text('content');
        $table->string('slug')->nullable();
        $table->tinyInteger('status')->default(1);
        $table->integer('user_id');
        $table->timestamps();
    });
}
```

Don't forget to run the **migrate** command to add a new **posts** table and **its** columns into our database:

```
php artisan migrate
```

Open **routes.php**, add these routes to the **admin route group**:

```
Route::get('posts', 'PostsController@index');
Route::get('posts/create', 'PostsController@create');
Route::post('posts/create', 'PostsController@store');
Route::get('posts/{id?}/edit', 'PostsController@edit');
Route::post('posts/{id?}/edit','PostsController@update');
```

Create a new **Admin/PostsController** by running this command:

```
php artisan make:controller Admin/PostsController
```

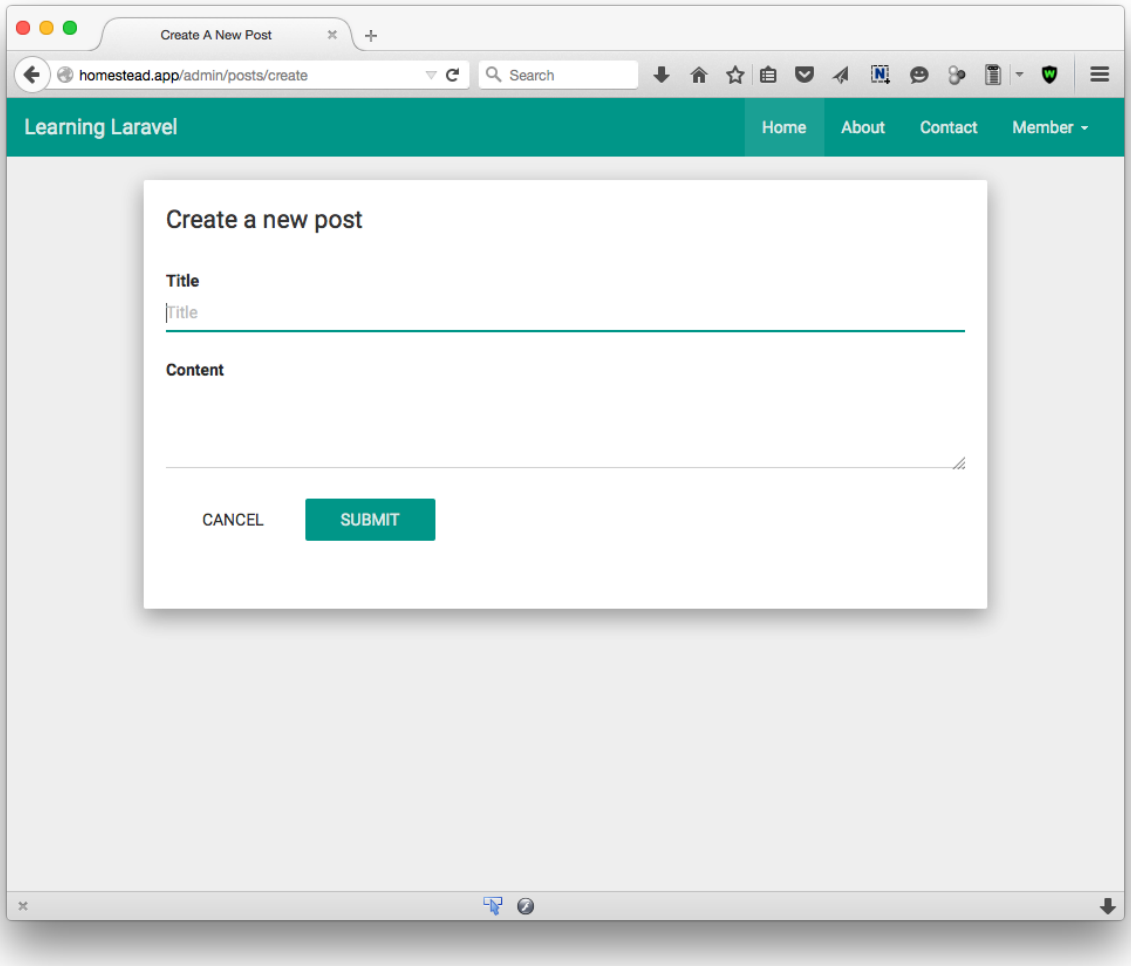
Open **Admin/PostsController**, update the **create** action as follows:

```
public function create()
{
    return view('backend.posts.create');
}
```

Create a new **posts** folder in the **backend** directory. Place a new **create** view there:

```
@extends('master')
@section('title', 'Create A New Post')

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="well well bs-component">

            <form class="form-horizontal" method="post">

                @foreach ($errors->all() as $error)
                    <p class="alert alert-danger">{{ $error }}</p>
                @endforeach

                @if (session('status'))
                    <div class="alert alert-success">
                        {{ session('status') }}
                    </div>
                @endif

                <input type="hidden" name="_token" value="{!! csrf_token() !!}">

                <fieldset>
                    <legend>Create a new post</legend>
                    <div class="form-group">
                        <label for="title" class="col-lg-2 control-label">Title<\
/label>
                        <div class="col-lg-10">
                            <input type="text" class="form-control" id="title" p\
laceholder="Title" name="title">
                        </div>
                    </div>
```

```
                    <div class="form-group">
                        <label for="content" class="col-lg-2 control-label">Cont\
ent</label>
                        <div class="col-lg-10">
                            <textarea class="form-control" rows="3" id="content"\
 name="content"></textarea>
                        </div>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button type="reset" class="btn btn-default">Cancel<\
/button>
                            <button type="submit" class="btn btn-primary">Submit\
</button>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
    </div>
@endsection
```



Create a new post

When we visit http://homestead.app/admin/posts/create, we can see a new post creation form.

However, we also need to create **Post Categories**. The Post Categories allow the users to select a category for the post they are creating.

Run this command to create a new **Category** model and its migration:

```
php artisan make:model Category -m
```

Next, open the **timestamp_create_categories_table.php** and update the **up** method as follows:

```
public function up()
{
    Schema::create('categories', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name', 255);
        $table->timestamps();
    });
}
```

This will create a new column called **name** to store categories' name.

Create a Many-to-Many relation

As you know, a post may have multiple categories, and a category may be associated with many posts. This is a **many-to-many** relation.

Many-to-many relations need an intermediate table (aka pivot table) to work. The table will have two columns, that store the foreign keys of two related models. For example, the **role_user** table is a pivot table, which consists the **users_id** and **role_id**.

The pivot table of posts and categories will be called **category_post** table (alphabetical order). When we use Entrust, the **role_user** table is automatically created for us, but we have to create our **category_post** manually.

Run this Artisan command to generate the **category_post** migration:

```
php artisan make:migration create_category_post_table --create=category_post
```

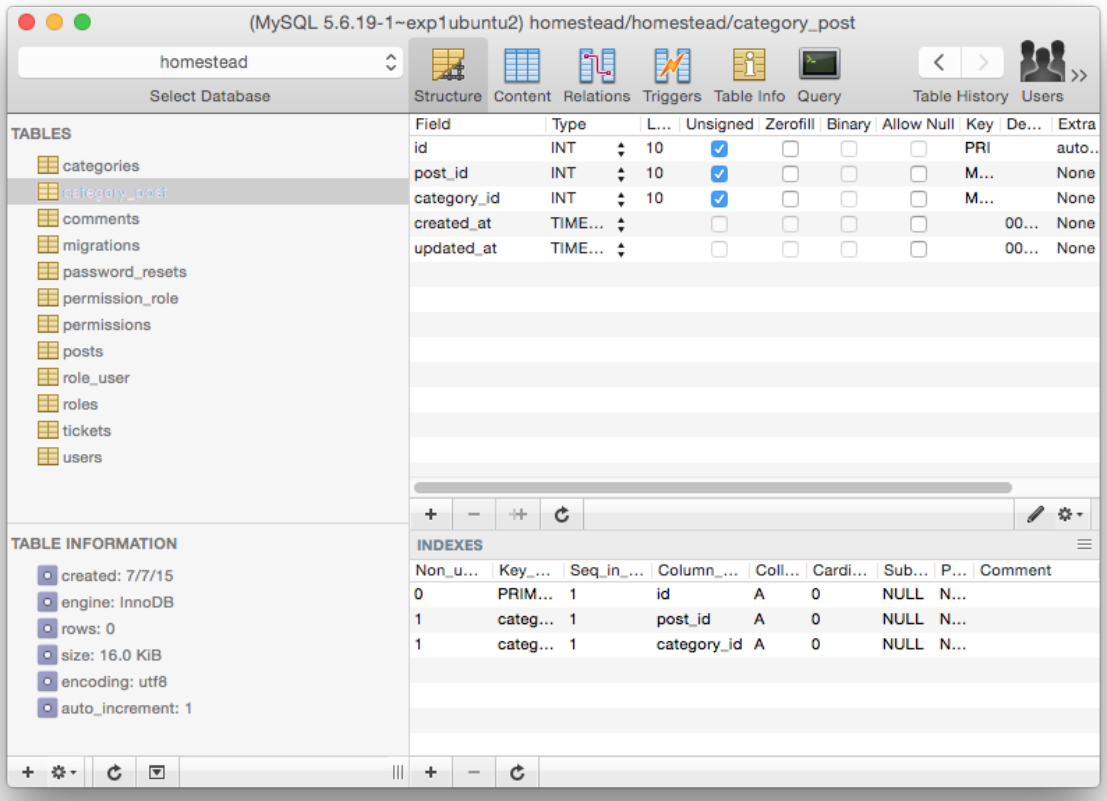
Next, open the new **timestamp_create_category_post_table** migration file and modify the **up** method:

```
public function up()
{
    Schema::create('category_post', function (Blueprint $table) {
        $table->increments('id')->unsigned();
        $table->integer('post_id')->unsigned()->index();
        $table->integer('category_id')->unsigned()->index();
        $table->timestamps();

        $table->foreign('category_id')
            ->references('id')->on('categories')
            ->onUpdate('cascade')
            ->onDelete('cascade');
```

```
        $table->foreign('post_id')
            ->references('id')->on('posts')
            ->onUpdate('cascade')
            ->onDelete('cascade');
    });
}
```

Save the changes and run the **migrate** command:

```
php artisan migrate
```



Category_post table

Check your database to make sure that you have all these tables: **posts**, **categories** and **category_post**.

Alternatively, you can use **Laravel 5 Extended Generators** package to generate the pivot table faster:

https://github.com/laracasts/Laravel-5-Generators-Extended

Great! Now open our **Post** model and update it to look like this:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $guarded = ['id'];

    public function categories()
    {
        return $this->belongsToMany('App\Category')->withTimestamps();
    }

}
```

We use **$guarded** property to make all columns mass assignable, except the id column.

The **belongsToMany** method is used to let Laravel know that this model has many-to-many relation.

Open our **Category** model and update it to look like this:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Category extends Model
{
    protected $guarded = ['id'];

    public function posts()
    {
        return $this->belongsToMany('App\Post')->withTimestamps();
    }
}
```

You've successfully defined a **many-to-many** relation.

Create and view categories

Let's make a form to create categories. Open **routes.php**, add these routes to the **admin route group**:

```
Route::get('categories', 'CategoriesController@index');
Route::get('categories/create', 'CategoriesController@create');
Route::post('categories/create', 'CategoriesController@store');
```

Generate a new **CategoriesController** by running this command:

```
php artisan make:controller Admin/CategoriesController
```

Open the newly created **CategoriesController** and update the **create** action:

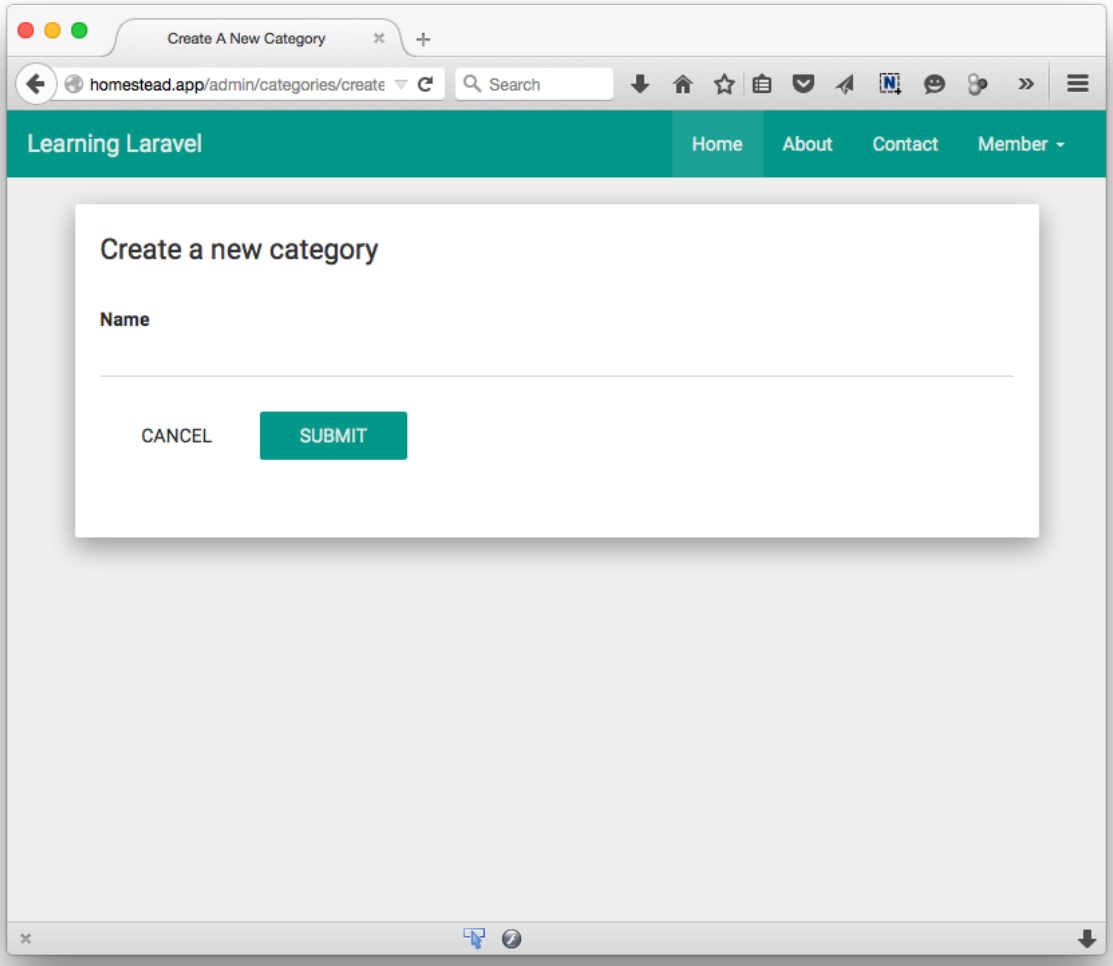
```
public function create()
{
    return view('backend.categories.create');
}
```

As usual, create a new **categories** folder inside the **backend** directory. Put a new **create** view in the **categories** directory:

```
@extends('master')
@section('title', 'Create A New Category')

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="well well bs-component">

            <form class="form-horizontal" method="post">

                @foreach ($errors->all() as $error)
                    <p class="alert alert-danger">{{ $error }}</p>
                @endforeach

                @if (session('status'))
                    <div class="alert alert-success">
                        {{ session('status') }}
                    </div>
                @endif
```

```
                <input type="hidden" name="_token" value="{!! csrf_token() !!}">

                <fieldset>
                    <legend>Create a new category</legend>
                    <div class="form-group">
                        <label for="name" class="col-lg-2 control-label">Name</l\
abel>
                        <div class="col-lg-10">
                            <input type="text" class="form-control" id="name" na\
me="name">
                        </div>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button type="reset" class="btn btn-default">Cancel<\
/button>
                            <button type="submit" class="btn btn-primary">Submit\
</button>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
    </div>
@endsection
```

Visit http://homestead.app/admin/categories/create, you should have a form to create a new category:



Create a new category

Good job! Now open the **CategoriesController** again, update the **store** action as follows:

```
public function store(CategoryFormRequest $request)
{
    $category = new Category(array(
        'name' => $request->get('name'),
    ));

    $category->save();

    return redirect('/admin/categories/create')->with('status', 'A new category \
has been created!');
```

```
}
```

Tell Laravel that you want to use the **CategoryFormRequest** and the **Category** model:

```
use App\Category;
use App\Http\Requests\CategoryFormRequest;
```

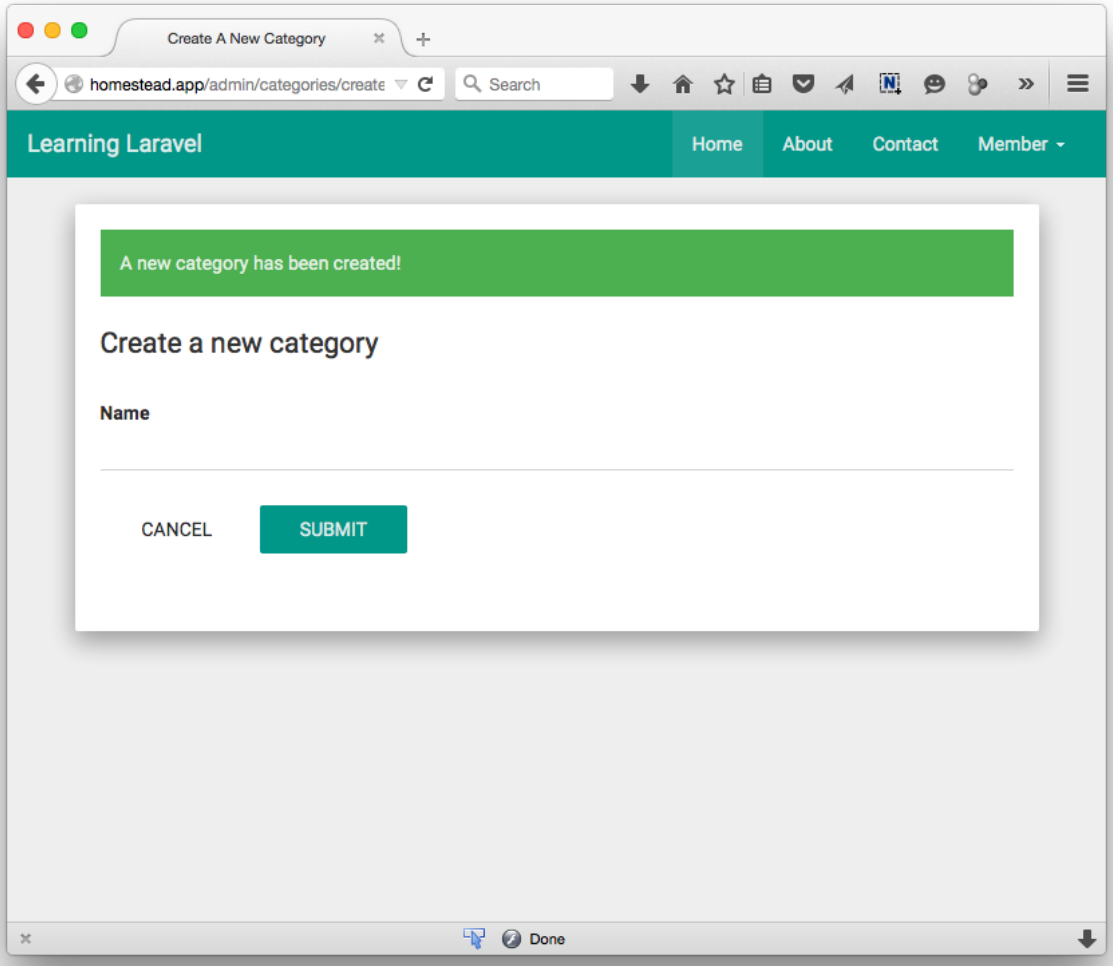
Generate a new **CategoryFormRequest** file by running this:

```
php artisan make:request CategoryFormRequest
```

Modify the **authorize** method and the **rules** method to look like this:

```
public function authorize()
{
    return true;
}

public function rules()
{
    return [
        'name'=> 'required|min:3',
    ];
}
```

Save the changes and visit the category form again. Create some categories (News, Laravel, Announcement, etc.) :



Create a new category successfully

To view all categories, open **CategoriesController** again, update the **index** action:

```
public function index()
{
    $categories = Category::all();
    return view('backend.categories.index', compact('categories'));
}
```

Create a new **index** view and place it in the **categories** directory:

```
@extends('master')
@section('title', 'All categories')
@section('content')

    <div class="container col-md-8 col-md-offset-2">
        <div class="panel panel-default">
            <div class="panel-heading">
                <h2> All categories </h2>
            </div>
            @if (session('status'))
                <div class="alert alert-success">
                    {{ session('status') }}
                </div>
            @endif
            @if ($categories->isEmpty())
                <p> There is no category.</p>
            @else
                <table class="table">
                    <tbody>
                    @foreach($categories as $category)
                        <tr>
                            <td>{!! $category->name !!}</td>
                        </tr>
                    @endforeach
                    </tbody>
                </table>
            @endif
        </div>
    </div>

@endsection
```

Now you can be able to view all categories at http://homestead.app/admin/categories.

Let's update the admin home page (dashboard) to include the category links there. Open the **backend/home** view:

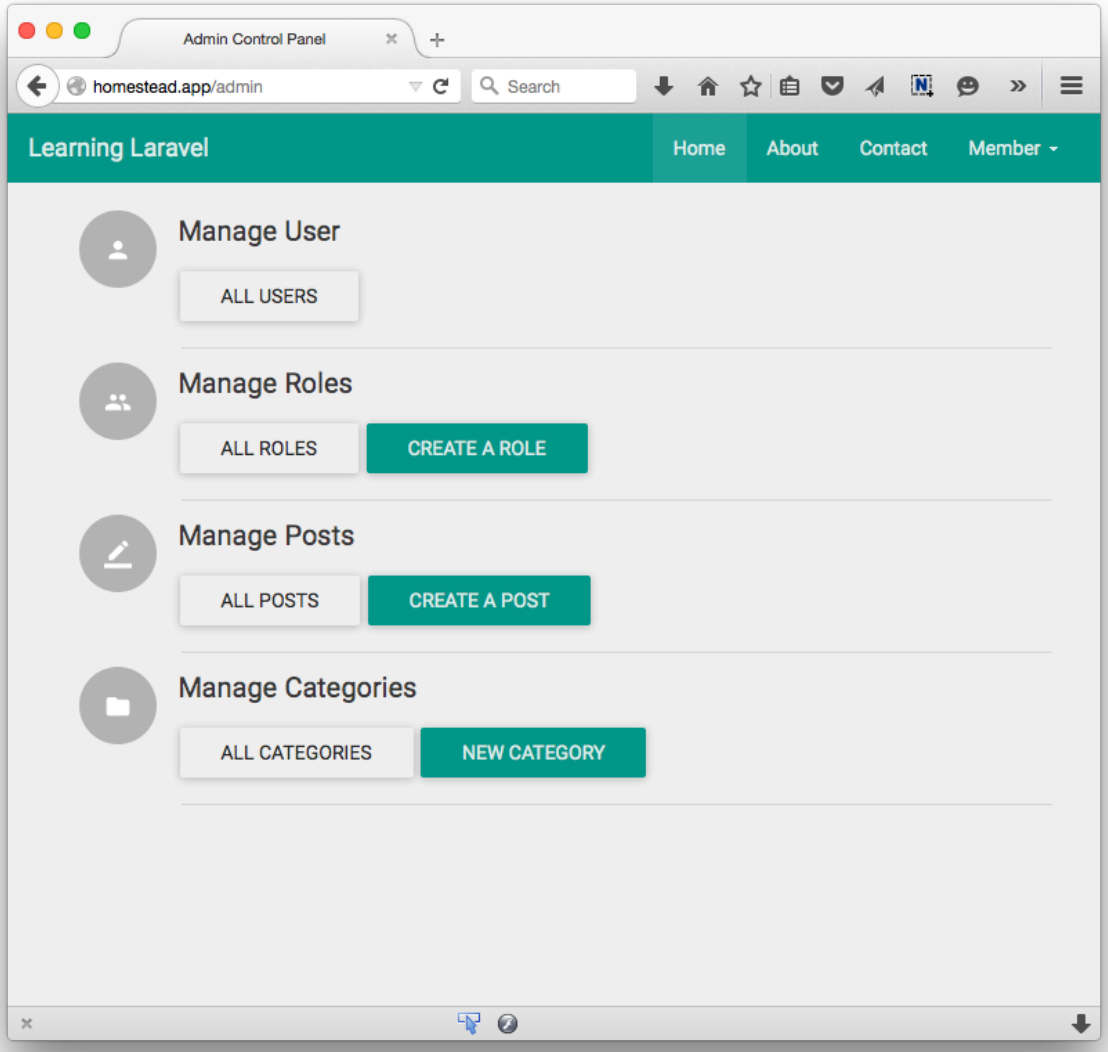
Find (at the end of the file):

```
        <a href="/admin/posts/create" class="btn btn-primary btn-raised">Create \
A Post</a>
    </div>
</div>
<div class="list-group-separator"></div>
```

Add below:

```
                <div class="list-group-item">
                    <div class="row-action-primary">
                        <i class="mdi-file-folder"></i>
                    </div>
                    <div class="row-content">
                        <div class="action-secondary"><i class="mdi-material-inf\
o"></i></div>
                        <h4 class="list-group-item-heading">Manage Categories</h\
4>
                        <a href="/admin/categories" class="btn btn-default btn-r\
aised">All Categories</a>
                        <a href="/admin/categories/create" class="btn btn-primar\
y btn-raised">New Category</a>
                    </div>
                </div>
                <div class="list-group-separator"></div>
```

Here is our new admin home page:



New admin home page

Select categories when creating a post

Now we can create a new post with categories.

Open **PostsController** and find:

```
class PostsController extends Controller
```

Add above:

```
use App\Category;
use App\Post;
use Illuminate\Support\Str;
```

Update the **create** action as follows:

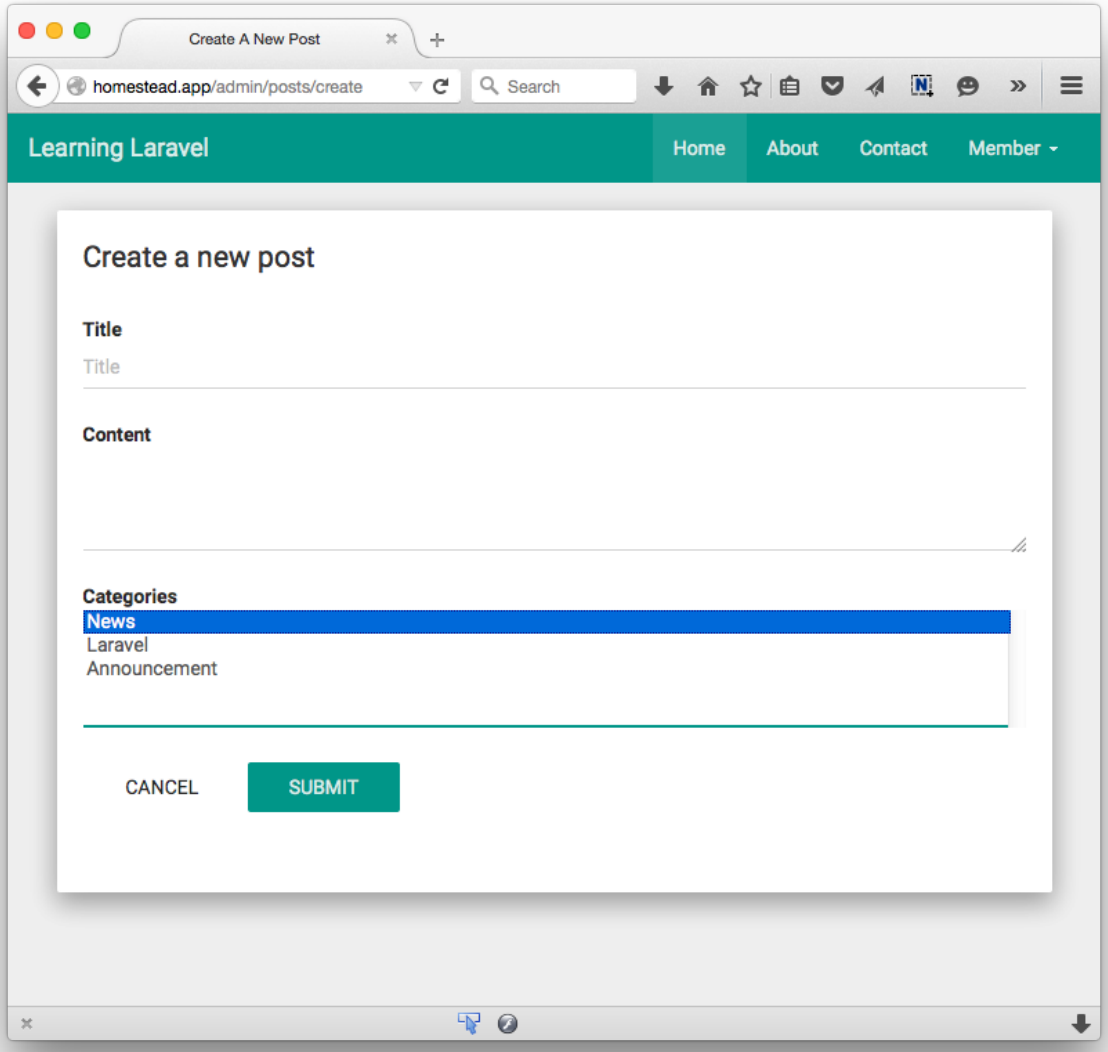
```
public function create()
{
    $categories = Category::all();
    return view('backend.posts.create', compact('categories'));
}
```

Next, open the **posts/create** view and find:

```
<div class="form-group">
    <label for="content" class="col-lg-2 control-label">Content</label>
    <div class="col-lg-10">
        <textarea class="form-control" rows="3" id="content" name="content"></te\
xtarea>
    </div>
</div>
```

Add this code below to allow users to select the categories:

```
<div class="form-group">
<label for="categories" class="col-lg-2 control-label">Categories</label>

    <div class="col-lg-10">
    <select class="form-control" id="category" name="categories[]" multiple>
            @foreach($categories as $category)
                <option value="{!! $category->id !!}">
                    {!! $category->name !!}
                </option>
            @endforeach
        </select>
    </div>
</div>
```



Create a new post with categories

We now have a new nice post creation form!

Now let's create the PostFormRequest first:

```
php artisan make:request PostFormRequest
```

Modify the **authorize** method and the **rules** method of the **PostFormRequest** to look like this:

```
public function authorize()
{
    return true;
}

public function rules()
{
    return [
        'title' => 'required',
        'content'=> 'required',
        'categories' => 'required',
    ];
}
```

Finally, open the **PostController** again and find:

```
class PostsController extends Controller
```

Add this code above to use PostFormRequest in this controller:

```
use App\Http\Requests\PostFormRequest;
```

Next, update the **store** action:

```
public function store(PostFormRequest $request)
{
    $post= new Post(array(
        'title' => $request->get('title'),
        'content' => $request->get('content'),
        'slug' => Str::slug($request->get('title'), '-'),
    ));

    $post->save();
    $post->categories()->sync($request->get('categories'));

    return redirect('/admin/posts/create')->with('status', 'The post has been cr\
eated!');
}
```

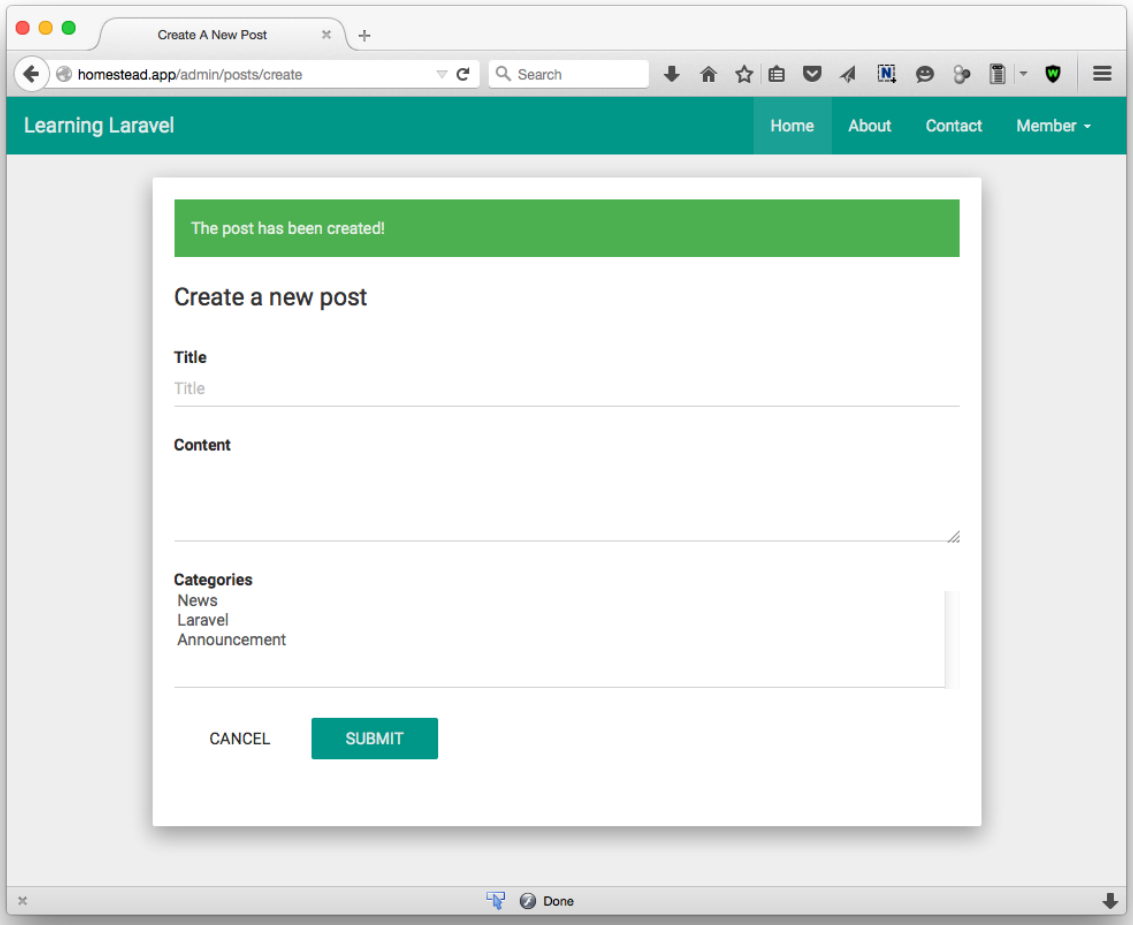
To create the post's slug, we can use **Str::slug** method to automatically generate a friendly slug from the given post's title.

After saving the post to the database using **$post->save**, we can use:

```
$post->categories()->sync($request->get('categories'));
```

to attach the selected categories to the post.

Try to create a new post now.



Create a new post successfully

Congratulations! You've just created your first blog post!

View and edit posts

Now that you know how to create a post, let's next create pages to view all the posts and edit them.

Display all posts

As a reminder, we've already defined this route in the previous section:

```
Route::get('posts', 'PostsController@index');
```

We can update the **index** action of the **PostsController** as follows:

```
public function index()
{
    $posts = Post::all();
    return view('backend.posts.index', compact('posts'));
}
```

Next, create a new **index** view at **backend/posts/index.blade.php**:

```
@extends('master')
@section('title', 'All posts')
@section('content')

    <div class="container col-md-8 col-md-offset-2">
        <div class="panel panel-default">
            <div class="panel-heading">
                <h2> All posts </h2>
            </div>
            @if (session('status'))
                <div class="alert alert-success">
                    {{ session('status') }}
                </div>
            @endif
            @if ($posts->isEmpty())
                <p> There is no post.</p>
            @else
                <table class="table">
                    <thead>
                    <tr>
                        <th>ID</th>
                        <th>Title</th>
                        <th>Slug</th>
                        <th>Created At</th>
                        <th>Updated At</th>

                    </tr>
                    </thead>
                    <tbody>
                    @foreach($posts as $post)
```

```
                        <tr>
                            <td>{!! $post->id !!}</td>
                            <td>
                                <a href="#">{!! $post->title !!} </a>
                            </td>
                            <td>{!! $post->slug !!}</td>
                            <td>{!! $post->created_at !!}</td>
                            <td>{!! $post->updated_at !!}</td>
                        </tr>
                    @endforeach
                    </tbody>
                </table>
            @endif
        </div>
    </div>

@endsection
```

Go to http://homestead.app/admin/posts to view all the posts.

Edit a post

We also have defined these routes:

```
Route::get('posts/{id?}/edit', 'PostsController@edit');
Route::post('posts/{id?}/edit','PostsController@update');
```

Let's modify the **edit** action to display a post edit form:

```
public function edit($id)
{
    $post = Post::whereId($id)->firstOrFail();
    $categories = Category::all();
    $selectedCategories = $post->categories->lists('id')->toArray();;
    return view('backend.posts.edit', compact('post', 'categories', 'selectedCat\
egories'));
}
```

Create a new **edit** view at **backend/posts/edit.blade.php**:

```
@extends('master')
@section('title', 'Edit A Post')

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="well well bs-component">

            <form class="form-horizontal" method="post">

                @foreach ($errors->all() as $error)
                    <p class="alert alert-danger">{{ $error }}</p>
                @endforeach

                @if (session('status'))
                    <div class="alert alert-success">
                        {{ session('status') }}
                    </div>
                @endif

                <input type="hidden" name="_token" value="{!! csrf_token() !!}">

                <fieldset>
                    <legend>Edit a post</legend>
                    <div class="form-group">
                        <label for="title" class="col-lg-2 control-label">Title<\
/label>
                        <div class="col-lg-10">
                            <input type="text" class="form-control" id="title" p\
laceholder="Title" name="title" value="{{ $post->title }}">
                        </div>
                    </div>
                    <div class="form-group">
                        <label for="content" class="col-lg-2 control-label">Cont\
ent</label>
                        <div class="col-lg-10">
                            <textarea class="form-control" rows="3" id="content"\
 name="content">{!! $post->content !!}</textarea>
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="select" class="col-lg-2 control-label">Categ\
```

```
ories</label>

                        <div class="col-lg-10">
                            <select class="form-control" id="categories" name="c\
ategories[]" multiple>
                                @foreach($categories as $category)
                                    <option value="{!! $category->id !!}"  @if(i\
n_array($category->id, $selectedCategories))
                                            selected="selected" @endif >{!! $cat\
egory->name !!}
                                    </option>
                                @endforeach
                            </select>
                        </div>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button type="reset" class="btn btn-default">Cancel<\
/button>
                            <button type="submit" class="btn btn-primary">Submit\
</button>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
    </div>
@endsection
```

After creating the form, you'll need to modify the **posts/index** view to add links to the posts.

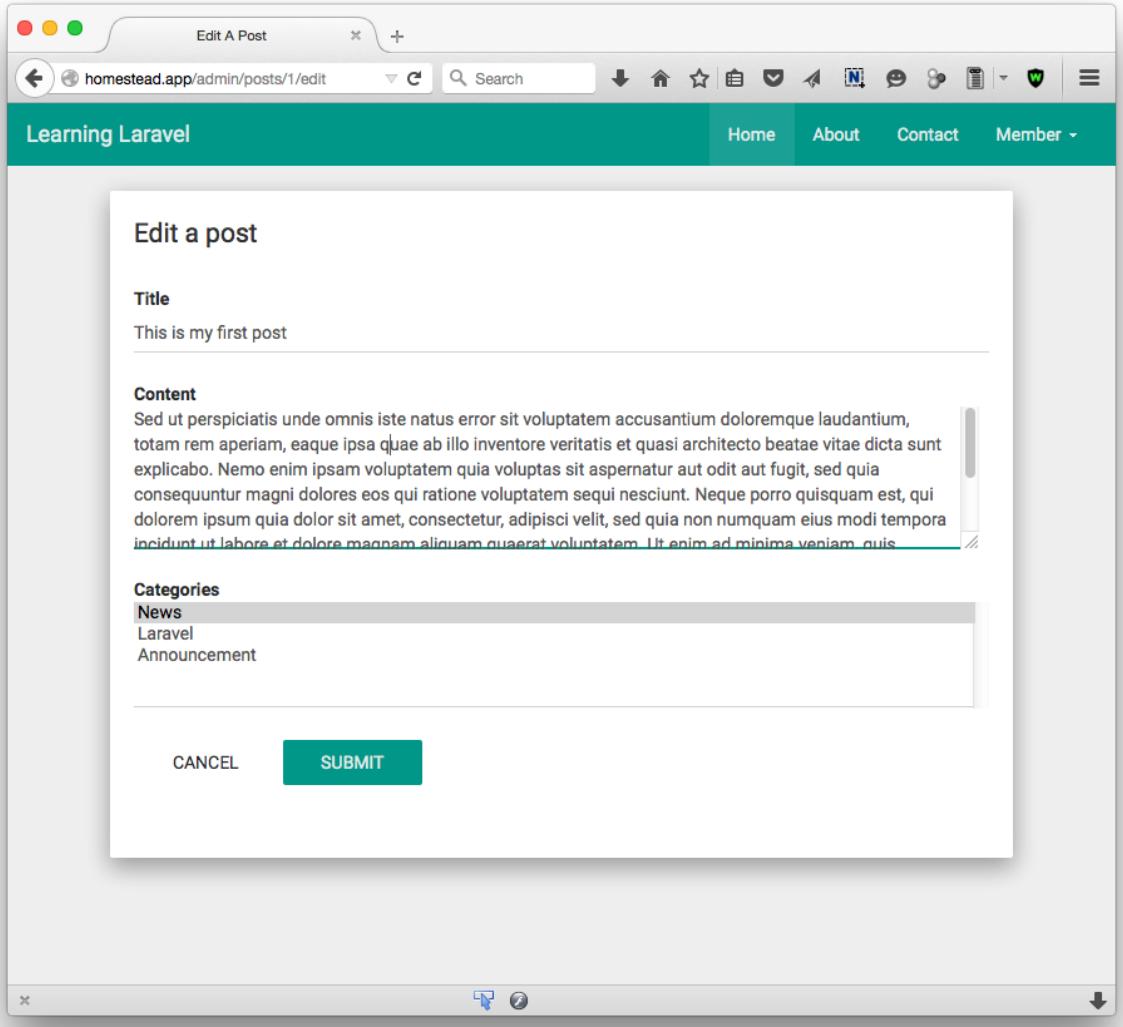
Find:

```
<a href="#">{!! $post->title !!} </a>
```

Edit to:

```
<a href="{!! action('Admin\PostsController@edit', $post->id) !!}">{!! $post->tit\
le !!} </a>
```

Go to http://homestead.app/admin/posts and click on the post title, you should be able to see a post edit form:



Post edit form

As always, create a new **PostEditFormRequest** by running this command:

```
php artisan make:request PostEditFormRequest
```

Modify the **authorize** method and the **rules** method to look like this:

```
public function authorize()
{
    return true;
}

public function rules()
{
    return [
        'title' => 'required',
        'content'=> 'required',
        'categories' => 'required',
    ];
}
```

Once the file is saved, open **PostsController** and find:

```
class PostsController extends Controller
```

Add above:

```
use App\Http\Requests\PostEditFormRequest;
```

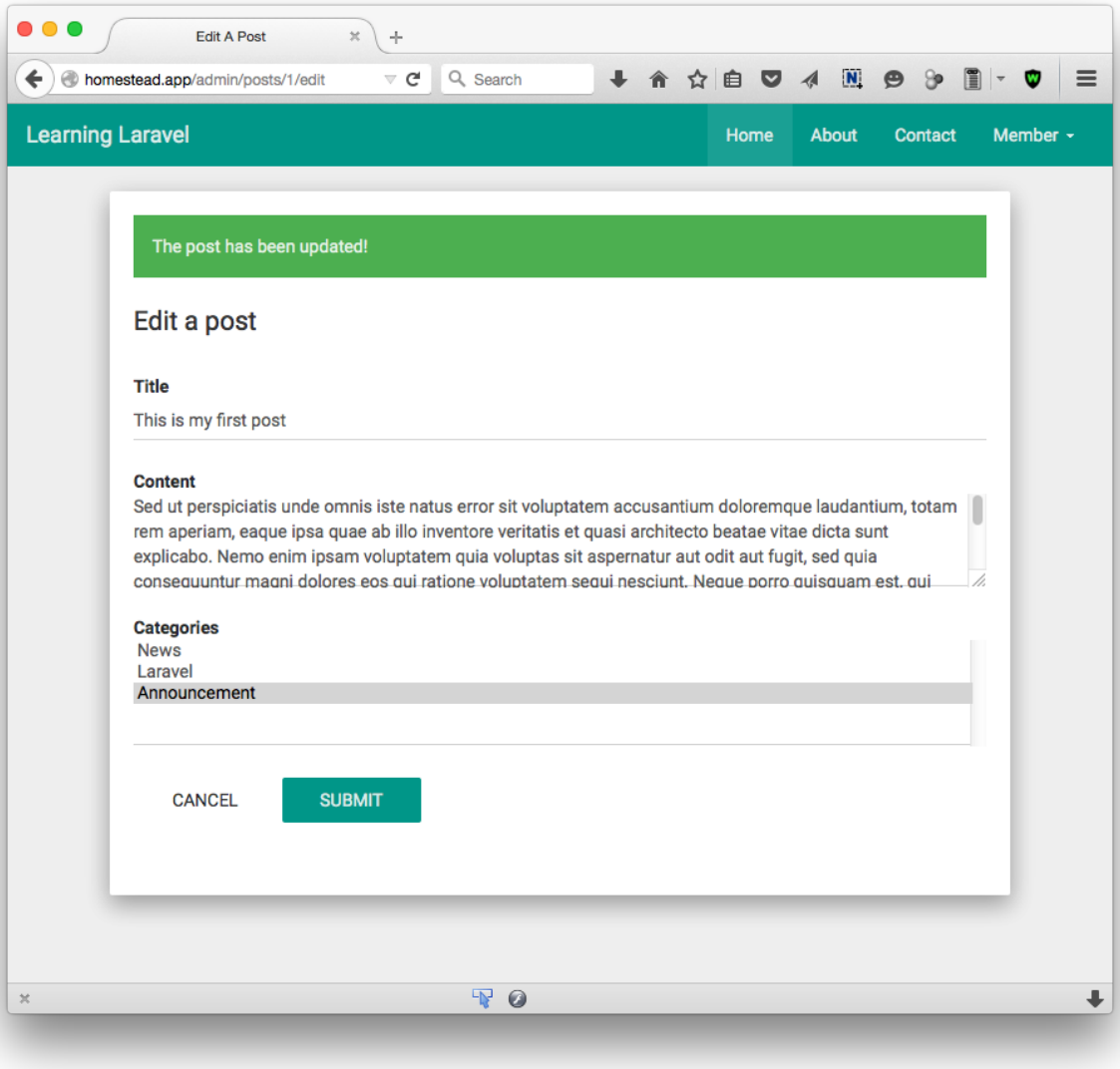
Finally, update the **update** method to save the changes to our database:

```
public function update($id, PostEditFormRequest $request)
{
    $post = Post::whereId($id)->firstOrFail();
    $post->title = $request->get('title');
    $post->content = $request->get('content');
    $post->slug = Str::slug($request->get('title'), '-');

    $post->save();
    $post->categories()->sync($request->get('categories'));

    return redirect(action('Admin\PostsController@edit', $post->id))->with('stat\
us', 'The post has been updated!');
}
```

Instead of creating a new **saveCategories** method (similar to the saveRoles method), we can **sync** the categories like this:

```
$post->categories()->sync($request->get('categories'));
```

Now try to edit a post and submit the changes.



Edit a post successfully

Well done! Once submitted, you should be able to update a post!

Display all blog posts

In this section, we will create a blog page that lists all blog posts. This page is public. Everyone can access it and read the posts.

Let's register a blog route by adding the following code to **routes.php**:

```
Route::get('/blog', 'BlogController@index');
```

We would need to create the **BlogController**:

```
php artisan make:controller BlogController
```

Insert the following code into it:

Find:

```
class BlogController extends Controller
```

Tell Laravel that you want to use the Post model in this controller. Add above:

```
use App\Post;
```

Now, update the **index** action:

```
public function index()
{
    $posts = Post::all();
    return view('blog.index', compact('posts'));
}
```

Create a new **index** view at **views/blog** directory. The following are the contents of the **views/blog/index.blade.php** file:

```
@extends('master')
@section('title', 'Blog')
@section('content')

    <div class="container col-md-8 col-md-offset-2">

        @if (session('status'))
            <div class="alert alert-success">
                {{ session('status') }}
            </div>
        @endif
```

```
        @if ($posts->isEmpty())
            <p> There is no post.</p>
        @else
            @foreach ($posts as $post)
                <div class="panel panel-default">
                    <div class="panel-heading">{!! $post->title !!}</div>
                    <div class="panel-body">
                        {!! mb_substr($post->content,0,500) !!}
                    </div>
                </div>
            @endforeach
        @endif
    </div>

@endsection
```

We use **mb_substr** (Multibyte String) function to display only **500 characters** of the post.

If you want to learn more about the function, visit:

http://php.net/manual/en/function.mb-substr.php

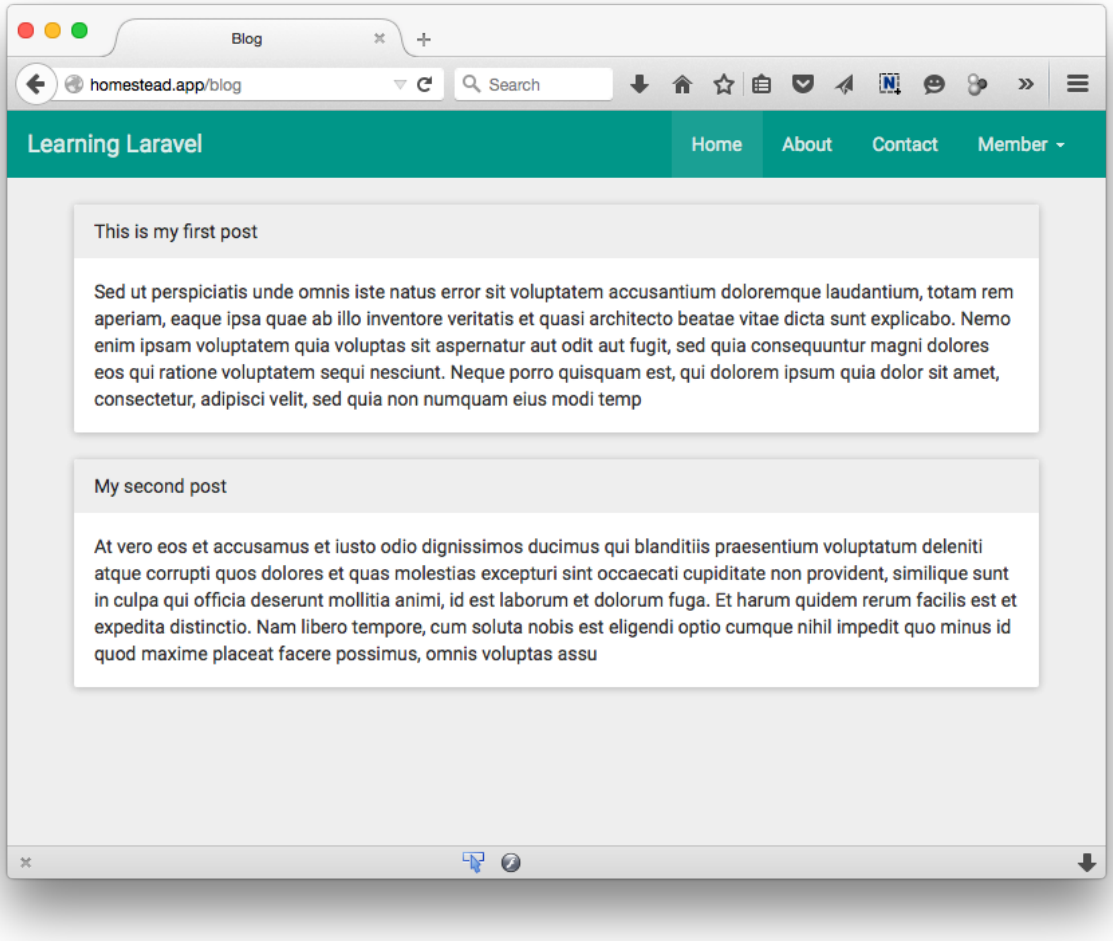
We should add a **blog** link to our navigation bar to access the blog page faster. Open **shared/-navbar.blade.php** and find:

```
<li><a href="/about">About</a></li>
```

Add above:

```
<li><a href="/blog">Blog</a></li>
```

Head over to your browser and visit the blog page.



Blog page

Now, everyone can see all your blog posts!

Display a single blog post

When we click on the post's title, we should see the full post.

Open **routes.php**, register this route:

```
Route::get('/blog/{slug?}', 'BlogController@show');
```

Before updating the **show** method, let's think about how we can implement the **blog post comment** feature.

> Note: I assume that you've created a **Comment** model. If not, please read Chapter 3 to know how to implement the comment feature.

Normally, we have to create a different Comment model (PostComment, for example) to list a post's comments. That means we have two columns: **comments** and **postcomments**. Fortunately, by defining **Polymorphic Relations**, we can store comments for our tickets and for our posts using only a single **comments** table!

Using Polymorphic Relations

You can read about this relationship here:

http://laravel.com/docs/master/eloquent-relationships#many-to-many-polymorphic-relations

To build this relationship, our **comments** table must have two columns: **post_id** and **post_type**. The **post_id** column contains the **ID** of the tickets or the posts, while the **post_type** contains the **class name** of the owning model (**AppTicket** or **AppPost**).

Currently, the **comments** table has the post_id column but it doesn't have the post_type column yet. Let's create a migration to add the column into our comments table:

```
php artisan make:migration add_post_type_to_comments_table --table=comments
```

This will create a new file called **timestamp_add_post_type_to_comments_table.php** in your migrations directory.

Open the file and modify it to look like this:

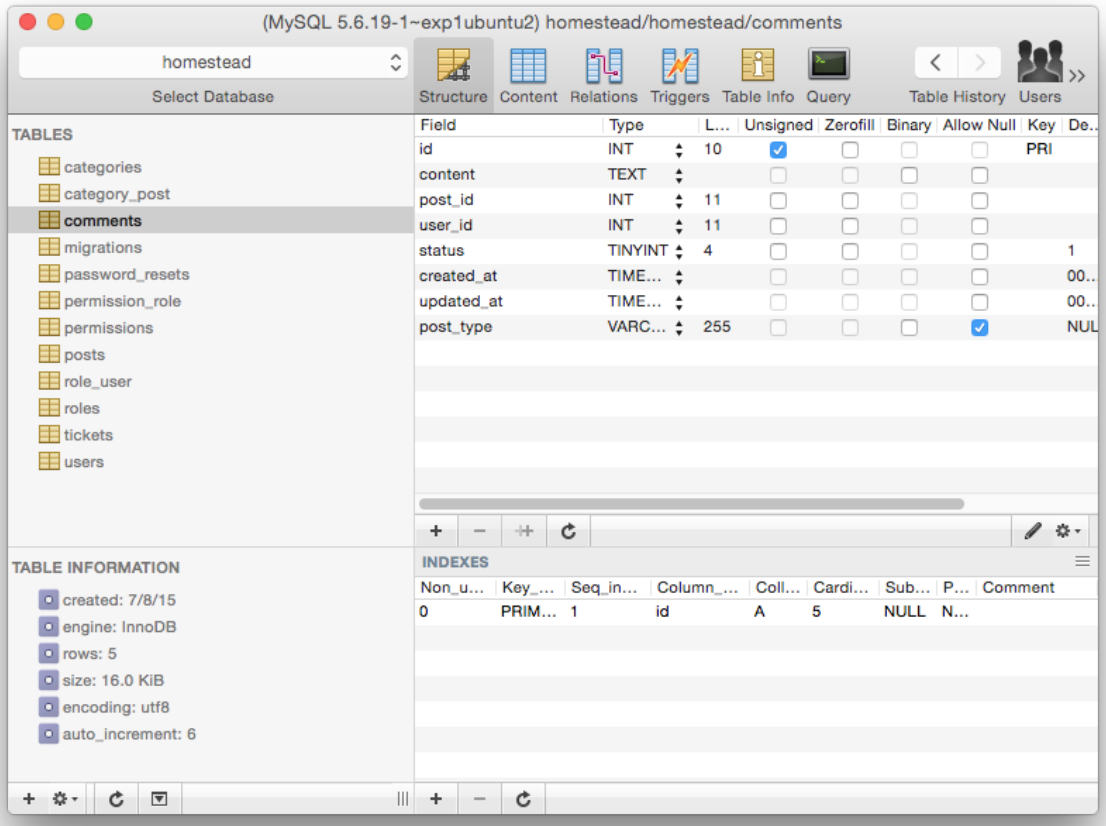
```
public function up()
{
    if(Schema::hasColumn('comments', 'post_type')) {

    } else {
        Schema::table('comments', function (Blueprint $table) {
            $table->string('post_type')->nullable();
        });
    }
}

public function down()
{
    Schema::table('comments', function (Blueprint $table) {
        $table->dropColumn('post_type');
    });
}
```

We check if the **comments** table has the **post_type** column. If not, we add the **post_type** column into the table.

Don't forget to run the migrate command:

```
php artisan migrate
```

Now our comments table should have the post_type column.



Post_type column

It's time to build the Polymorphic relationship. Open the **Comment** model, update it as follows:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    protected $guarded = ['id'];

    public function post()
    {
        return $this->morphTo();
    }
}
```

The **post()** method will get all of the owning models.

Open the **Ticket** model, update the **comments()** method to:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Ticket extends Model
{
    protected $fillable = ['title', 'content', 'slug', 'status', 'user_id'];

    public function comments()
    {
        return $this->morphMany('App\Comment', 'post');
    }
}
```

Open the **Post** model, update it as follows:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $guarded = ['id'];

    public function categories()
    {
        return $this->belongsToMany('App\Category')->withTimestamps();
    }

    public function comments()
    {
        return $this->morphMany('App\Comment', 'post');
    }
}
```

The **comments()** method will get all of the post's comments . Done! You've defined Polymorphic relations.

Now, open **BlogController** and update the **show** method:

```php
public function show($slug)
{
    $post = Post::whereSlug($slug)->firstOrFail();
    $comments = $post->comments()->get();
    return view('blog.show', compact('post', 'comments'));
}
```

Here is the **blog/show** view:

```
@extends('master')
@section('title', 'View a post')
@section('content')

    <div class="container col-md-8 col-md-offset-2">
            <div class="well well bs-component">
                <div class="content">
                    <h2 class="header">{!! $post->title !!}</h2>
                    <p> {!! $post->content !!} </p>
                </div>
                <div class="clearfix"></div>
            </div>

            @foreach($comments as $comment)
                <div class="well well bs-component">
                    <div class="content">
                        {!! $comment->content !!}
                    </div>
                </div>
            @endforeach

            <div class="well well bs-component">
                <form class="form-horizontal" method="post" action="/comment">

                    @foreach($errors->all() as $error)
                        <p class="alert alert-danger">{{ $error }}</p>
                    @endforeach

                    @if(session('status'))
                        <div class="alert alert-success">
                            {{ session('status') }}
                        </div>
                    @endif

                    <input type="hidden" name="_token" value="{!! csrf_token() !\
!}">
                    <input type="hidden" name="post_id" value="{!! $post->id !!}\
">
                    <input type="hidden" name="post_type" value="App\Post">

                    <fieldset>
                        <legend>Comment</legend>
```

```
                        <div class="form-group">
                            <div class="col-lg-12">
                                <textarea class="form-control" rows="3" id="cont\
ent" name="content"></textarea>
                            </div>
                        </div>

                        <div class="form-group">
                            <div class="col-lg-10 col-lg-offset-2">
                                <button type="reset" class="btn btn-default">Can\
cel</button>
                                <button type="submit" class="btn btn-primary">Po\
st</button>
                            </div>
                        </div>
                    </fieldset>
                </form>
            </div>
    </div>

@endsection
```

Notice that we've added a new hidden field called **post_type**. Because this is the post's comment, the value of the field is: "**AppPost**" (class name).

Open **CommentsController** and find:

```
    $comment = new Comment(array(
        'post_id' => $request->get('post_id'),
        'content' => $request->get('content'),
    ));
```

Add **post_type** to save its value into the database:

```
    $comment = new Comment(array(
        'post_id' => $request->get('post_id'),
        'content' => $request->get('content'),
        'post_type' => $request->get('post_type')
    ));
```

Finally, open the **ticket/show** view and find:

```
 <input type="hidden" name="post_id" value="{!! $ticket->id !!}">
```

Add below:

```
<input type="hidden" name="post_type" value="App\Ticket">
```

We add a new hidden field called **post_type**, which has the "**AppTicket**" value, to let Laravel know that the comments of this view belong to the **Ticket** model.

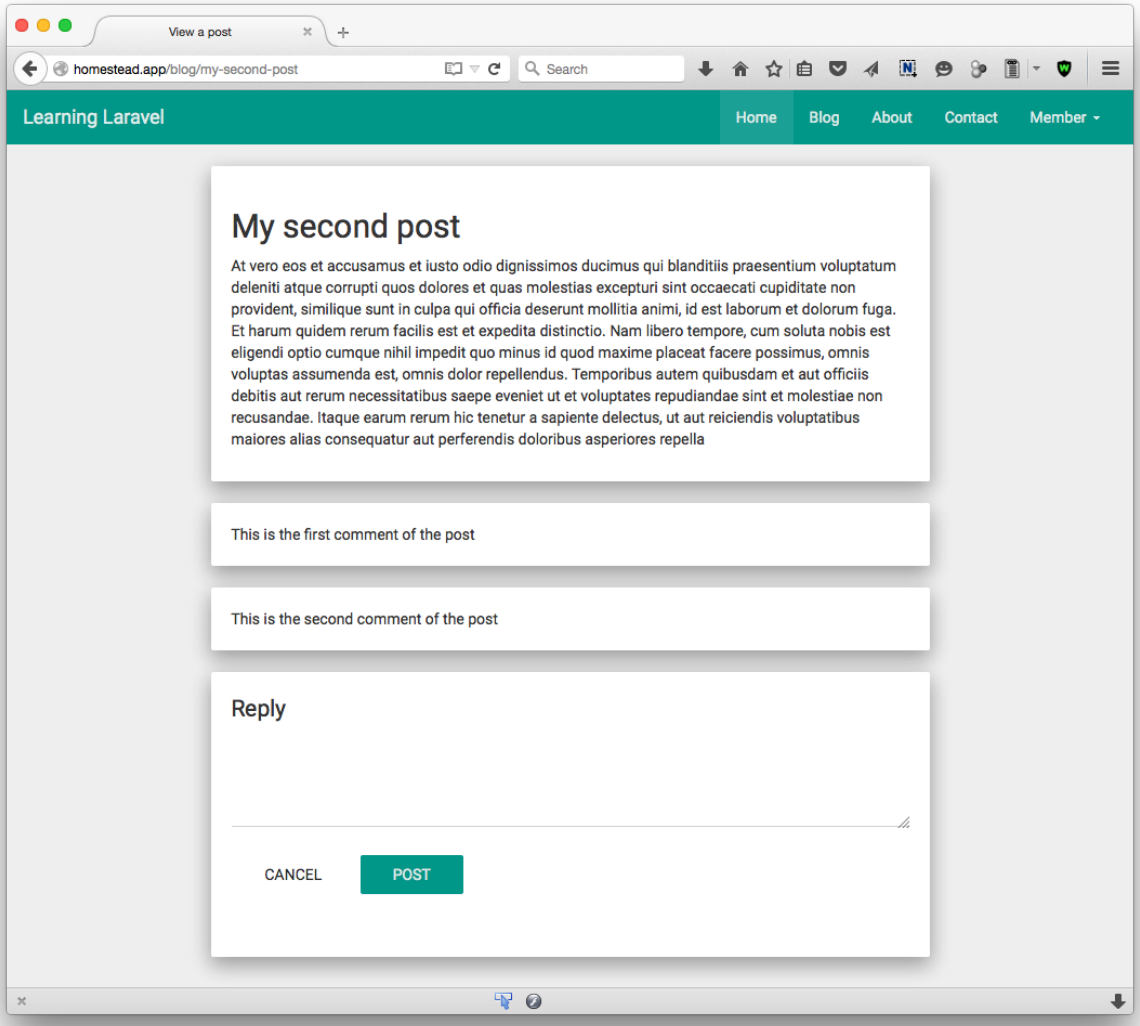
One more thing to do, open the **blog/index** view and find:

```
<div class="panel-heading">{!! $post->title !!}</div>
```

Modify to:

```
<div class="panel-heading"><a href="{!! action('BlogController@show', $post->slu\
g) !!}">{!! $post->title !!}</a></div>
```

Now, go to your browser and view a post. Try to post a comment as well.



View a single blog post

Let's check the database to see how things work:

id	content	post_id	user_id	status	created_at	updated_at	post_type
1	This is my first comment!	2	0	1	2015-06-26 15:46:39	2015-06-26 15:46:39	App\Ticket
2	This is my second comment!	2	0	1	2015-06-26 16:56:06	2015-06-26 16:56:06	App\Ticket
3	This is the first comment of the post	2	0	1	2015-07-08 07:55:07	2015-07-08 07:55:07	App\Post
4	testing comment	6	0	1	2015-07-08 08:00:16	2015-07-08 08:00:16	App\Ticket
5	This is the second comment of the post	2	0	1	2015-07-08 12:26:43	2015-07-08 12:26:43	App\Post

Comments table

As you see, even though the **comments** have the same **post_id** (which is 2), their **post_type** values are different. The ORM uses the **post_type** column to determine which model is related to the comments.

It's a bit confusing. However, if you know how to use Polymorphic Relations, you will gain many benefits.

Seeding our database

Instead of creating test data manually, we can use Laravel's **Seeder** class to populating our database. In this section, we will learn how to use **Seeder** by creating sample users for our application.

To create a seeder, run this command:

```
php artisan make:seeder UserTableSeeder
```

This will create a class called **UserTableSeeder**, which is placed in **database/seeds** directory.

```
public function run()
{
    DB::table('users')->insert([
     [
        'name' => 'Nathan',
        'email' => str_random(12).'@email.com',
        'password' => bcrypt('yourPassword'),
         'created_at'       => new DateTime,
         'updated_at'       => new DateTime,
    ],
    [
        'name' => 'David',
        'email' => str_random(12).'@email.com',
        'password' => bcrypt('yourPassword'),
        'created_at'       => new DateTime,
        'updated_at'       => new DateTime,

    ],
    [
        'name' => 'Lisa',
        'email' => str_random(12).'@email.com',
        'password' => bcrypt('yourPassword'),
        'created_at'       => new DateTime,
        'updated_at'       => new DateTime,
```

```
    ],
    ]);
}
```

Within the **run** method, we use **database query builder** to create dummy user data. Keep in mind that we can also load data from a **JSON** or **CSV** file.

Instead of using the Hash facade to hash the password, you can also use the **bcrypt()** helper function to do that.

Read more about **Database Query Builder** here:

http://laravel.com/docs/master/queries

You may also want to try Faker, which is a popular PHP package that helps to create fake data effectively.

https://github.com/fzaninotto/Faker

After writing the seeder class, open **database/seeds/DatabaseSeeder.php**.

```
public function run()
{
    Model::unguard();

    // $this->call('UserTableSeeder');

    Model::reguard();
}
```

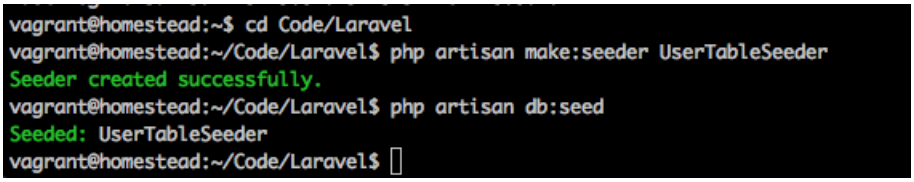
Normally, we would create many seeder classes to seed different dummy data. For example, **UserTableSeeder** is used to create fake users, **PostTableSeeder** is used to create dummy posts, etc. We use the **DatabaseSeeder** file to control the order of seeder classes. In this case, we only have one UserTableSeeder class, so let's write like this:

```
public function run()
{
    Model::unguard();

    $this->call('UserTableSeeder');

    Model::reguard();
}
```

To seed data, run this Artisan command:

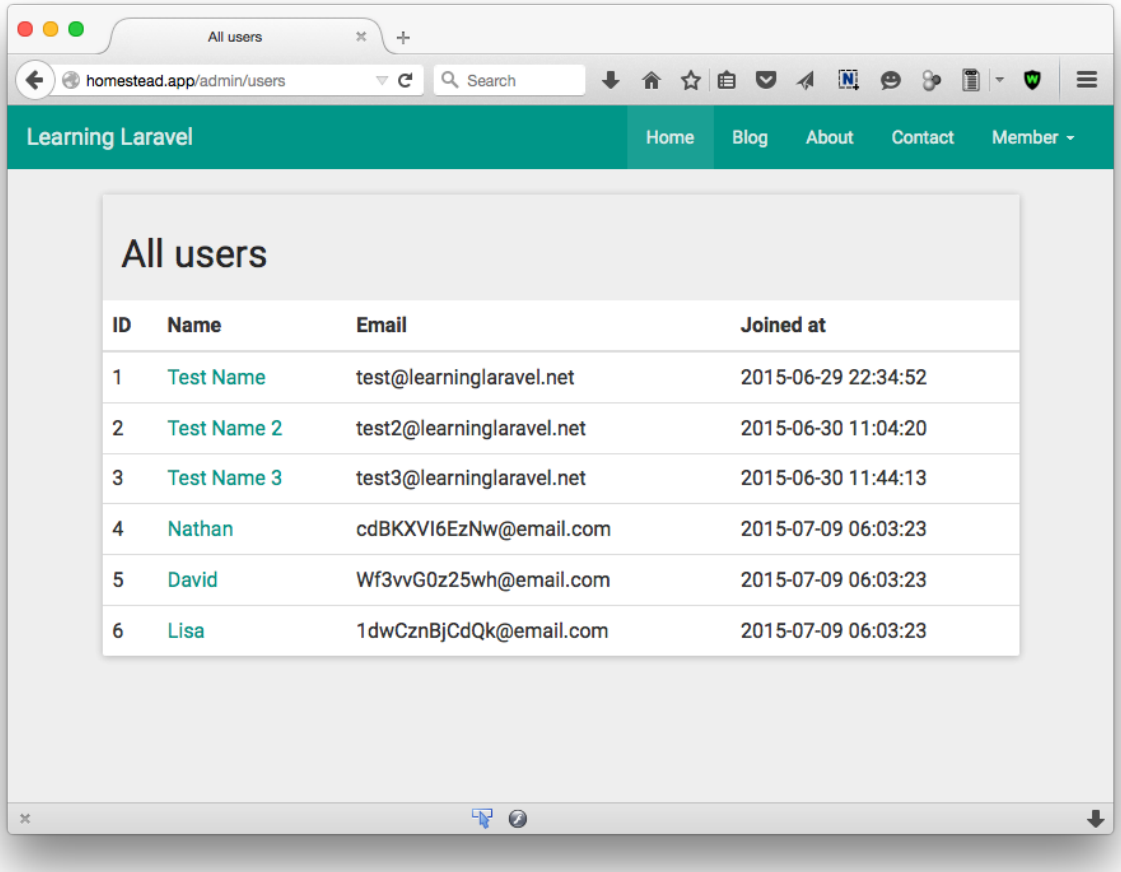
```
php artisan db:seed
```

This will execute the **run** method of the **UserTableSeed** class and insert data into our database.



Seeding our database

Now check your database or visit http://homestead.app/admin/users, there are some new users:



New users

Seeding is very useful. You may try to use this feature to create posts, tickets and other data to test your application!

Localization

You can easily translate strings to multiple languages using Laravel localization feature.

All language files are stored in the **resources/lang** directory.

By default, there is an **en (English)** directory, which contains English strings. If you want to support other languages, create a new directory at the same level as the **en** directory. Please note that all language directories should be named using **ISO 639-1 Code**.

Read more about ISO 639-1 Code here:

http://www.loc.gov/standards/iso639-2/php/code_list.php

Open **en/passwords.php** file:

```
<?php

return [

    'password' => 'Passwords must be at least six characters and match the confi\
rmation.',
    'user' => "We can't find a user with that e-mail address.",
    'token' => 'This password reset token is invalid.',
    'sent' => 'We have e-mailed your password reset link!',
    'reset' => 'Your password has been reset!',

];
```

As you see, a language file simply returns an array that contains translated strings.

You can change the default language by editing this:

```
'locale' => 'en',
```

The line can be found in the **config/app.php** configuration file.

Let's create a new language file to translate our application!

Go to **resources/lang/en** directory, create a new file called **main.php**, which looks like this:

```
<?php

return [

    'subtitle' => 'Fastest way to learn Laravel',

];
```

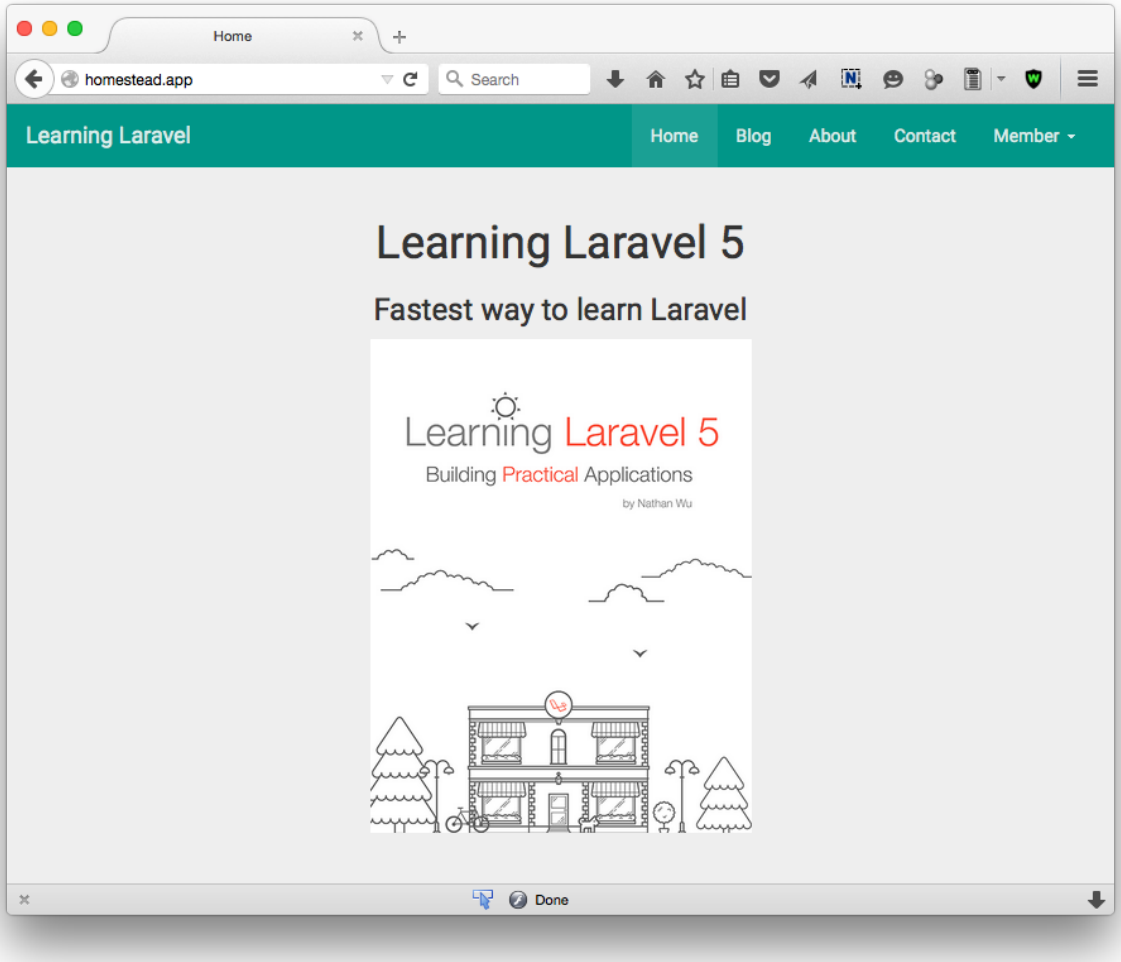
Next, open the **home** view (views/home.blade.php file) and find:

```
<h3 class="text-center margin-top-100 editContent">Building Practical Applicatio\
ns</h3>
```

Change to:

```
<h3 class="text-center margin-top-100 editContent">{!! trans('main.subtitle') !!\
}</h3>
```

Head over to your browser:



New home page

The **trans** helper function is used to retrieve lines from language files:

```
{!! trans('main.subtitle') !!}
```

main is the name of the language file (main.php). **subtitle** is the key of the language line.

As you notice, we may use the localization feature to manage strings as well. For example, put all your application's strings into the **main.php** file; when you want to edit a string, you can find it easily.

Don't forget to read the docs to learn more about localization:

http://laravel.com/docs/master/localization

Chapter 4 Summary

Congratulations! You've built a complete blog application!

Our application is not perfect yet, but you may use it as a starter template and apply some concepts that you've known to build a larger application.

Towards the end of this chapter, let's review what we have learned:

- You've known how to authenticate users and add login throttling to your app.
- You now understand more about routes and route group.
- Using Middleware, you can handle requests/responses effectively.
- Many Laravel applications are using Entrust, it's one of the best packages to manage roles and permissions. We only use the "roles" feature in this book. Try to create some permissions to secure your apps better.
- Understanding Many-to-Many Relations and Polymorphic Relations is hard at first, but you'll gain many benefits later.
- Now you can be able to seed your database! Seeding is easy. Right?
- The Laravel localization feature is simple, but it's very helpful and powerful. Try to use it in all your applications to manage all the strings.

Remember that, this is just a beginning, there is still so much more to learn.

Hopefully, you will have a successful application someday! Enjoy the journey!

If you wish to learn more about Laravel, check our Laravel 5 Cookbook out!

> **Note:** I'll be updating all chapters to fix some mistakes in both code and grammar. Also, more sections will be added later. If you would like to give feedback, report bugs, or ask any questions, please email us at support@learninglaravel.net. Thank you.

Chapter 5: Deploying Our Laravel Applications

Currently, we're just working locally on our personal computers. We will have to deploy our applications to some hosting services or servers so that everyone can access it. There are many ways to make your application visible to the rest of the world!

In this chapter, I will show you how to deploy your Laravel applications using these popular methods:

1. Deploying your apps on shared hosting services
2. Deploying your apps using DigitalOcean

If you find out some better solutions, feel free to contact us. I'll update the book to talk about other approaches.

To deploy a Laravel application, you may have to follow these steps:

- Create a different directory structure. (If you're using shared hosting)
- Setup your web server (If you're using servers or cloud services, such as DigitalOcean, Linode, etc.)
- Upload your applications to your host/server.
- Give proper permissions to your files.
- Create a database for your application.

Deploying your apps on shared hosting services

Basically, your Laravel applications are just PHP applications, that means you can upload them directly to any supported shared hostings, and they may work just fine.

However, Laravel is not designed to work on shared hosting services. Therefore, you will need to do some extra configurations. I don't recommend to use this approach, but the choice is yours.

Here are some popular web hosting services that you can use:

Host Gator

GoDaddy

Blue Host

Please note that each web hosting service requires a different configuration, so bear in mind that you need to find a way and spend extra time to make your Laravel applications work properly.

Deploying on Godaddy shared hosting

First, let's assume that you have a **new Laravel application** and your **home directory** is:

```
/home/content/learninglaravel/public_html
```

Now, follow these steps:

- You will need to create a directory on the same level as the **public_html** directory. This directory will hold your Laravel application. Because it's on the same level as your **public_html** directory, others cannot access it. This makes the application more secure.
- Upload your **Laravel application** to the new directory using an FTP client (such as Filezilla, Transmit, CuteFTP, etc.), **except the public folder**.
- Upload the **public** folder to the **public_html** directory.

Good job! Now open the **index.php** file and change two paths:

```
require __DIR__.'/../bootstrap/autoload.php';
```

and

```
$app = require_once __DIR__.'/../bootstrap/app.php';
```

Last step, go to the **public_html** directory and modify the **.htaccess** file (If you don't have one, create a new file):

```
RewriteEngine On
RewriteCond %{REQUEST_URI} !^public
RewriteRule ^(.*)$ public/$1 [L]
```

Well done, now it's time to visit your **website URL**! (www.yourdomain.com). You should see the Laravel welcome screen.

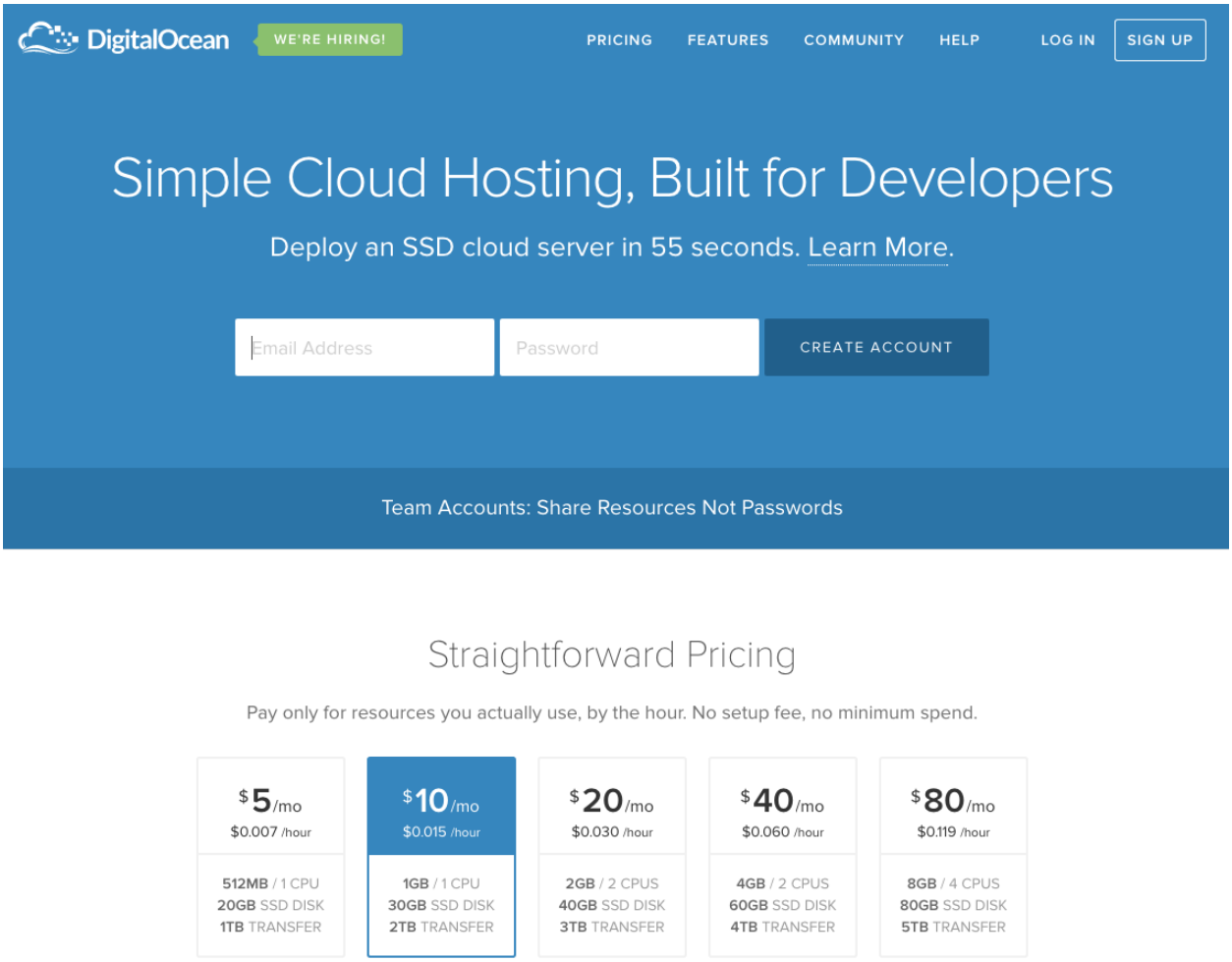
Laravel welcome screen

You've just deployed your Laravel app!

> **Note**: Please note that your application may not work properly on shared hosting and there is a security risk. Laravel is not designed to work on a shared host.

Deploying your apps using DigitalOcean

DigitalOcean is one of the best cloud server providers that you can find around the world. You can get their cheapest SSD Cloud Server for just $5 a month. More than 450,000 developers have been using DigitalOcean to deploy their applications!



DigitalOcean pricing

DigitalOcean is also my best favorite hosting solution. The performance is really amazing!

In this section, I'll show you how to install Laravel with **Nginx** on a **Ubuntu 14.04 LTS VPS**. (Ubuntu 14.04 Long Term Support Virtual Private Server)

Why do I choose **Ubuntu 14.04**? There are some newer versions of Ubuntu (14.10, 15.04, etc.), but the 14.04 is an LTS version, that means we will receive updates and support for at least five years.

Alternatively, you can use Laravel Forge to install the VPS and configure everything for you. However, you will have to pay $10/month.

Deploy a new Ubuntu server

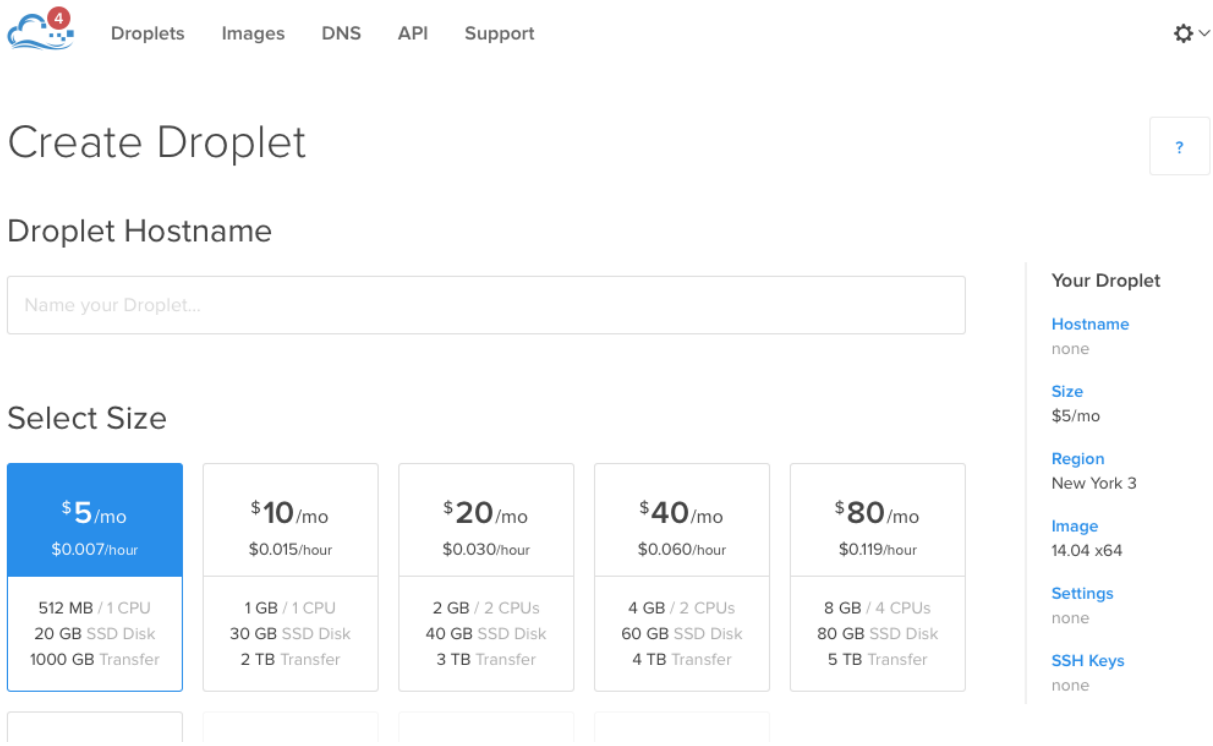
First, you will need to register a new account at DigitalOcean. You can use this link to get **$10** for free, that means you can use their **$5 cloud server** for **two months**.

https://www.digitalocean.com/?refcode=5f7e95cb014e

> **Note**: You will need to provide your credit card information or Paypal to activate your account.

After your account has been activated. You will need to create a "**droplet**", which is a cloud server. Click on the **Create Droplet** button or go to:

https://cloud.digitalocean.com/droplets/new



Create a droplet

Follow these steps:

- Give your Droplet a name (For example, learninglaravel).
- Select your droplet size and region that you like.
- At the **Select Image** section, be sure to choose **Ubuntu 14.04 x64**.
- You may skip other settings.
- Click "**Create Droplet**" to create your first cloud server!

Wait for a few seconds and...

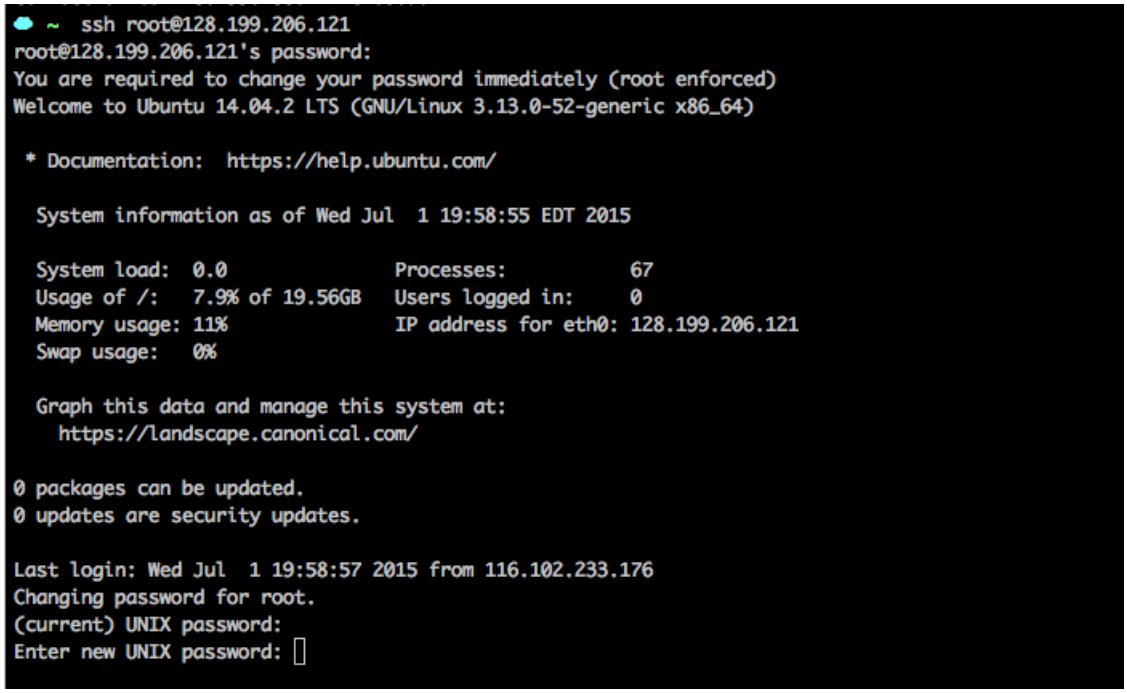
Congratulations! You just have a new Ubuntu VPS!

Check your email to get the **username** and **password**, you will need to use them to access your server.

```
Droplet Name: learninglaravel
IP Address: 128.199.206.121
Username: root
Password: yourPassword
```

Great! Now you can access the new server via **Terminal** or **Git Bash** by using this command:

```
ssh root@128.199.206.121
```



Access your first server

The first time you login, it will ask you to **change the password**. Enter the **current Unix password** again, and then enter your **new password** to change it.

Finally, run this command to check and update all current packages to the latest version:

```
apt-get update && apt-get upgrade
```

We're now ready to install Nginx and other packages!

Install Nginx

Install Nginx is very simple, you just need to run this command:

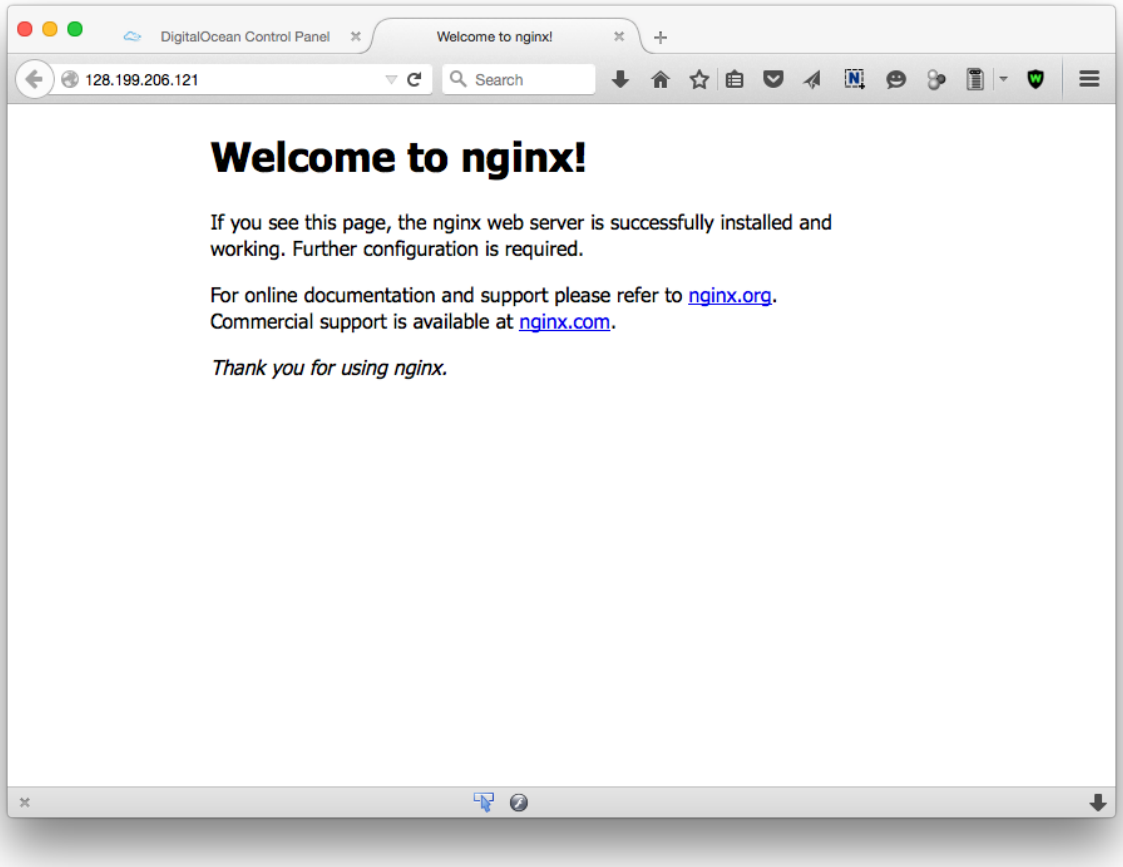
```
sudo apt-get install nginx
```

Enter **y** and hit **Return** (**Enter** on Windows) if it ask you to confirm anything.

Done! You can now visit your Nginx server via the **IP address**:

http://128.199.206.121

> **Note**: Of course that your IP address must be different.



Your website

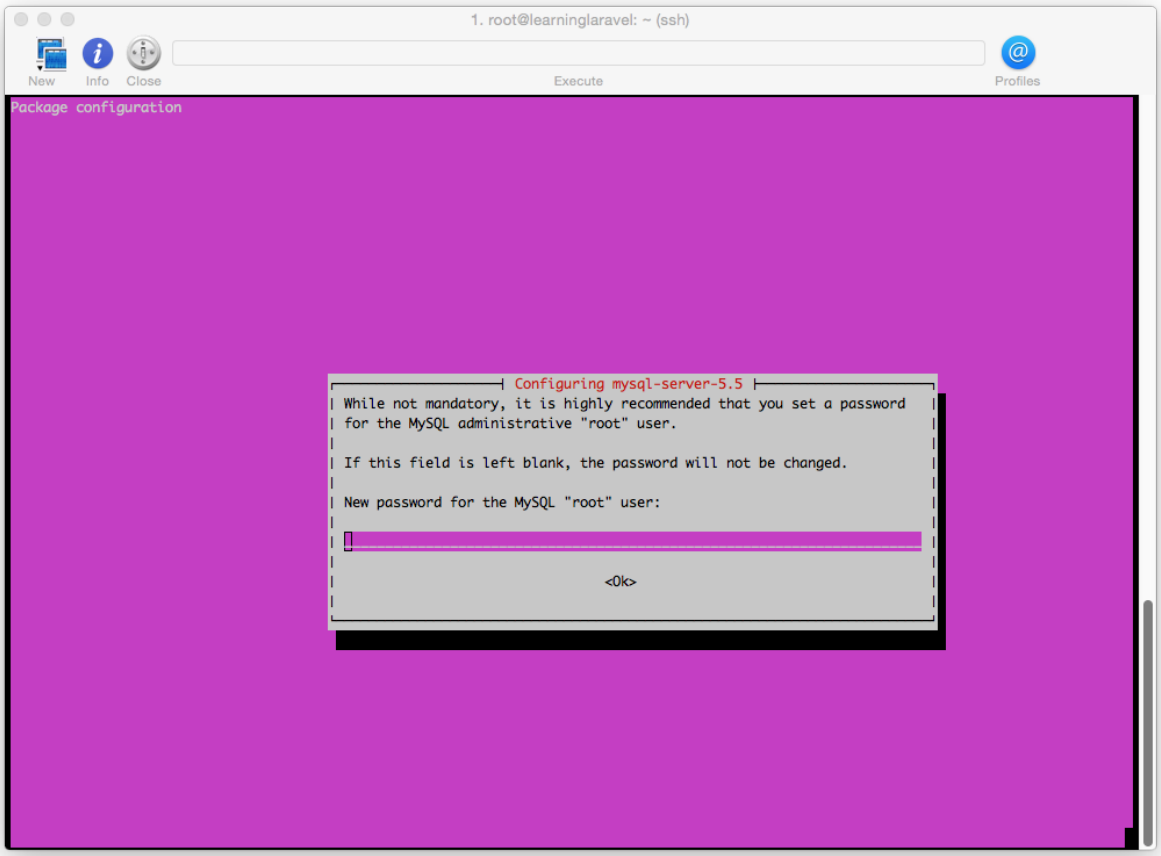
Install MySQL Server

> **Note**: We're using MySQL in this book, so I will install MySQL, you can install other databases if you like.

To get started, we need to install MySQL to store our data. Run this command:

```
sudo apt-get install mysql-server
```

Say **Y** to everything. If you're asked to enter a new password for the MySQL root user, give it a password.



MySQL password

Next, you need to tell MySQL to configure the databases for you and run a security script to ensure that your settings are safe:

```
sudo mysql_install_db
```

```
sudo mysql_secure_installation
```

If it asks you for a root password again, just say **n** (no).

Say **Yes** to everything else.

You should see this message:

```
All done!  If you've completed all of the above steps, your MySQL
installation should now be secure.

Thanks for using MySQL!
```

Good job! You've installed MySQL. We will install PHP in the next section.

Install Nginx, PHP and other packages

Nginx doesn't come with PHP by default, so you have to install PHP and some required packages to run Laravel. Run this command:

```
apt-get install php5-fpm php5-cli php5-mcrypt git php5-mysql
```

Say **Y** (yes) if it asks you anything.

You should have PHP installed.

One more thing to do, we need to change a setting to make PHP more secure. Open the **php.ini** file:

sudo nano /etc/php5/fpm/php.ini

Find (near the end of the file):

```
;cgi.fix_pathinfo=1
```

You can use **Ctrl + V** to quickly move to the next page. Uncomment it and change it to 0

```
cgi.fix_pathinfo=0
```

Once complete, press **Ctrl + X**, and say **Y** to save the file. Press **Return (or Enter)** to exit.

By doing this, you tell PHP that you don't want PHP to look for the nearest file if it cannot find your requested file. if you don't make the change, there is a possible security risk.

After that, we need to edit the **server block** (aka **virtual hosts**) file. Open it:

```
sudo nano /etc/nginx/sites-available/default
```

Find:

```
root /usr/share/nginx/html;
```

This is the path to your Laravel application, we don't have the Laravel application yet, but let's change it to:

```
root /var/www/learninglaravel.net/html;
```

Find:

```
index index.html index.htm;
```

Change to:

```
index index.php index.html index.htm;
```

Find:

```
    location / {
            # First attempt to serve request as file, then
            # as directory, then fall back to displaying a 404.
            try_files $uri $uri/ =404;
            # Uncomment to enable naxsi on this location
            # include /etc/nginx/naxsi.rules
    }
```

Add below:

```
    location ~ \.php$ {
            try_files $uri /index.php =404;
            fastcgi_pass unix:/var/run/php5-fpm.sock;
            fastcgi_index index.php;
            fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
            include fastcgi_params;
    }
```

Save the file and exit.

Because we don't have the **/var/www/learninglaravel.net/html** directory yet, let's create it.

```
sudo mkdir -p /var/www/learninglaravel.net/html
```

Be sure to give it a proper permission:

```
sudo chown -R www-data:www-data /var/www/learninglaravel.net/html

sudo chmod 755 /var/www
```

Now, make a test file called **index.html** to test our configurations:

```
sudo nano /var/www/learninglaravel.net/html/index.html
```

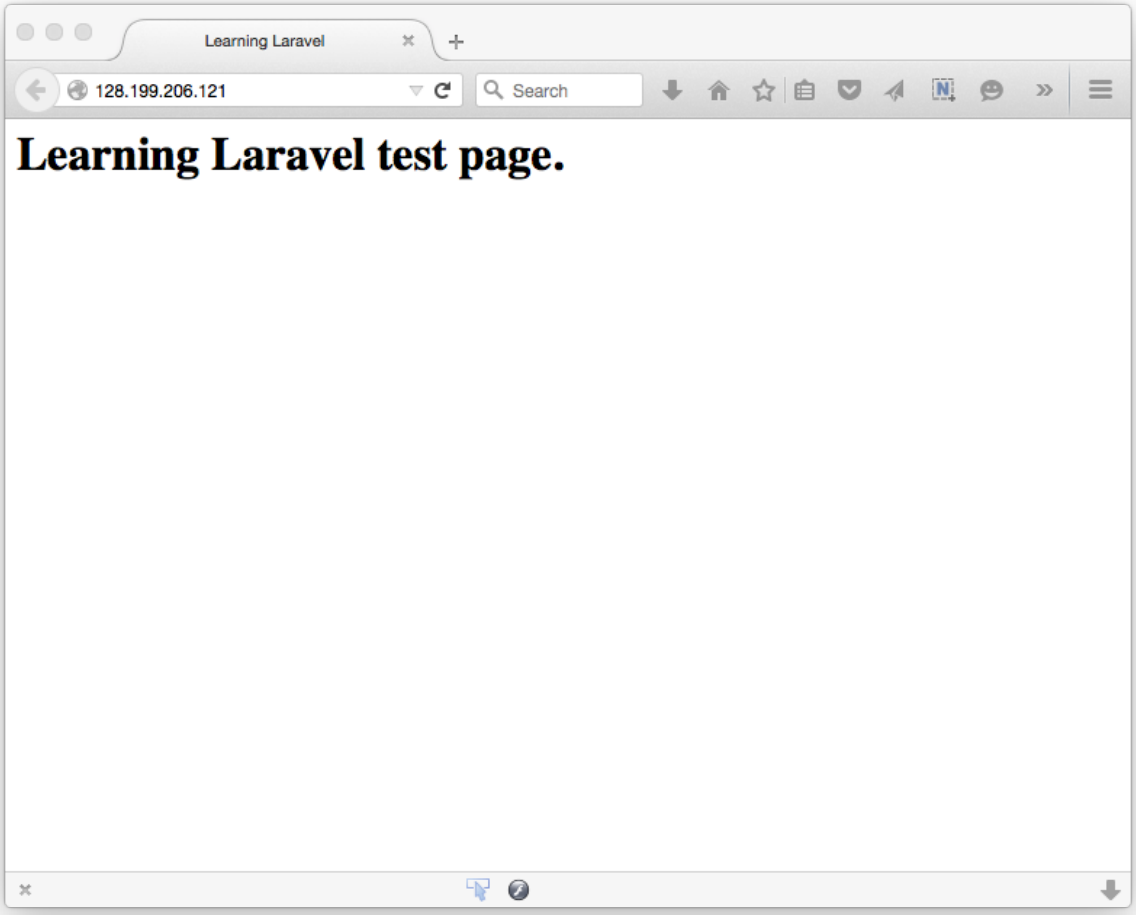
Here is the content:

```
<html>
<head>
<title>Learning Laravel</title>
</head>
<body>
<h1>Learning Laravel test page.</h1>
</body>
</html>
```

Finally, restart PHP and Nginx:

```
service php5-fpm restart
service nginx restart
```

Now when you visit your website via its IP address, you should see:



Your website

Install Laravel

Now that we have everything in order, we will be going to install **Composer** and use it to download Laravel Installer!

If you're installing Laravel on 512MB droplets, you must add a swapfile to Ubuntu to prevent it from running out of memory.

```
dd if=/dev/zero of=/swapfile bs=1024 count=512k
mkswap /swapfile
swapon /swapfile
```

> **Note**: If you reboot, you have to add the swapfile again.

Run this simple command to install Composer:

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

Next, we will use **Composer** to download the **Laravel Installer**:

```
composer global require "laravel/installer=~1.1"
```

If you read the Laravel docs, you may see this:

"Make sure to place the ~/.composer/vendor/bin directory in your PATH so the laravel executable can be located by your system."

Ok, let's do that by running these commands:

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

Once finished, we're finally at the part that we've been waiting for: **Install Laravel!**

We will put our Laravel application at **/var/www/learninglaravel.net/**. Type the following to get there:

```
cd /var/www/learninglaravel.net/
```

It's time to install Laravel:

```
laravel new laravel
```

This is a pretty standard process. I hope you understand what we've done. If you don't, please read Chapter 1.

By now, we should have our Laravel app installed at **/var/www/learninglaravel.net/laravel**.

Once that step is done, we must give the directories proper permissions:

```
chown -R www-data /var/www/learninglaravel.net/laravel/storage
chmod -R 775 /var/www/learninglaravel.net/laravel/public
chmod -R 0777 /var/www/learninglaravel.net/laravel/storage

chgrp -R www-data /var/www/learninglaravel.net/laravel/public
chmod -R 775 /var/www/learninglaravel.net/laravel/storage
```

These commands should do the trick.

One last step, edit the **server block** file again:

```
sudo nano /etc/nginx/sites-available/default
```

Find:

```
root /var/www/learninglaravel.net/html;
```

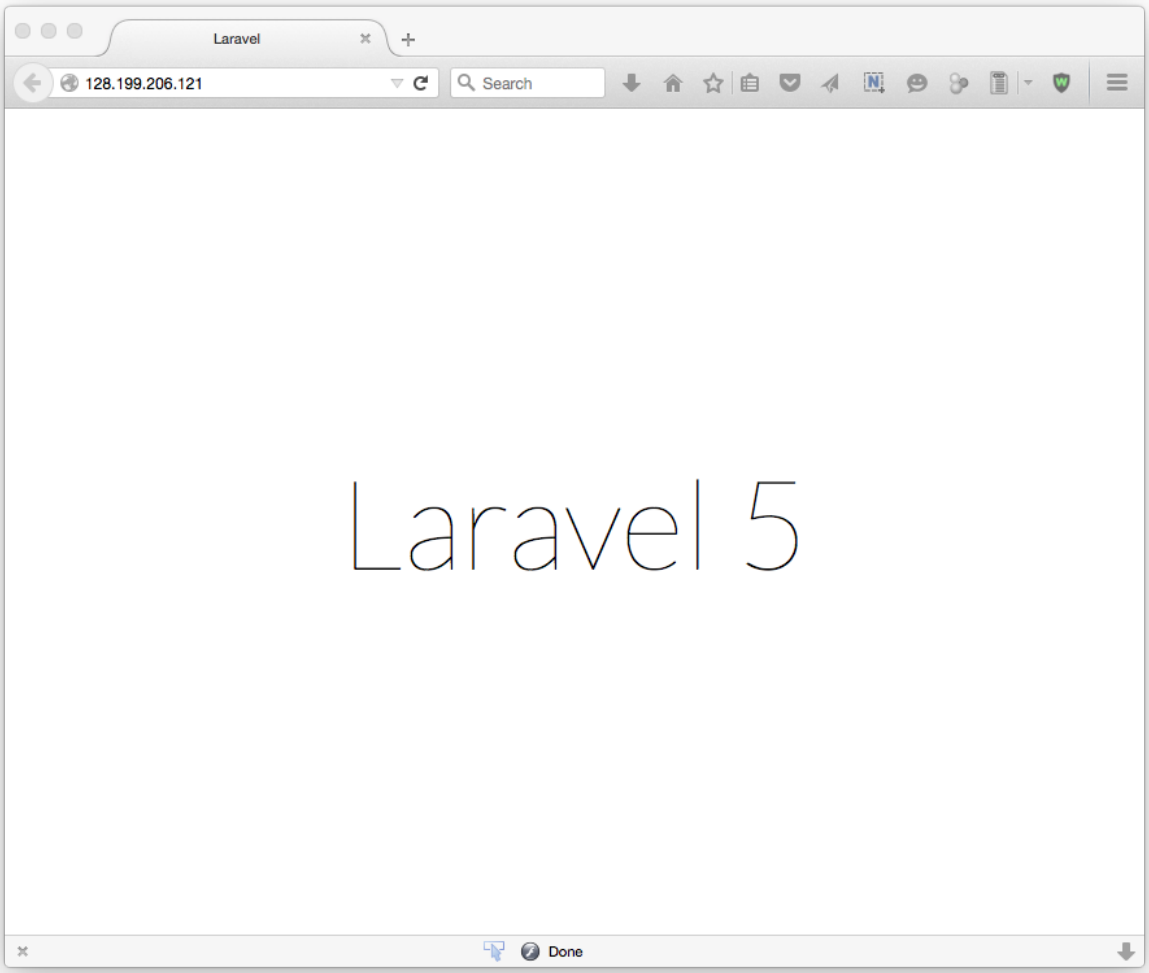
Change to:

```
root /var/www/learninglaravel.net/laravel/public;
```

Restart Nginx:

```
service nginx restart
```

Go ahead and visit your Laravel app in browser:



Your new Laravel application on DigitalOcean

Your application is now ready to rock the world!

Possible Errors

If you see this error:

```
Whoops, looks like something went wrong.

No supported encrypter found. The cipher and / or key length are invalid.
```

This is a Laravel 5.1 bug. Sometimes, your app doesn't have a correct application key (this key is generated automatically when installing Laravel)

You need to run these commands to fix this bug:

```
php artisan key:generate
```

```
php artisan config:clear
```

Finally, restart your Nginx server:

```
service nginx restart
```

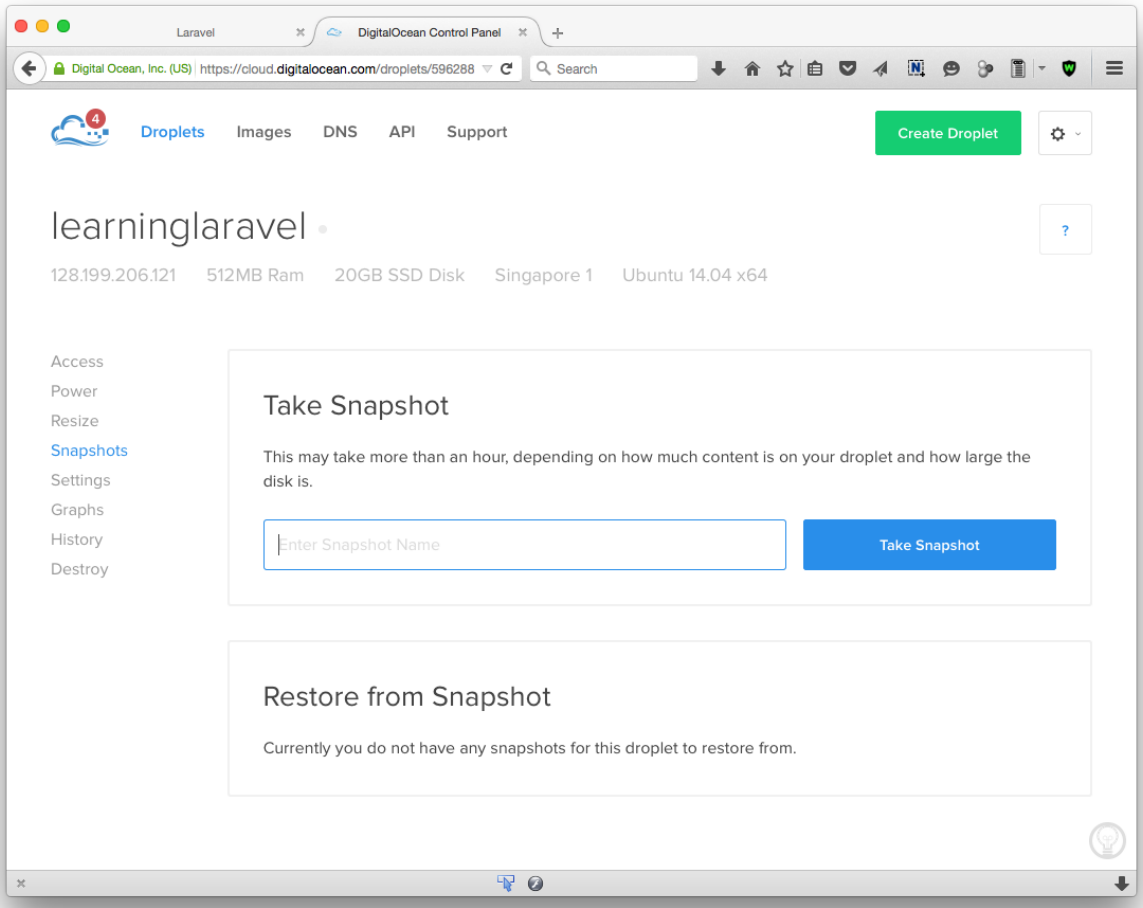
Take a snapshot of your application

I know that the process is a bit complicated. The great thing is, you can take a snapshot of your VPS, and then you can restore it later. When you have new projects, you don't have to start over again! Everything can be done by **two clicks**!

To take a snapshot, shutdown your server first:

```
shutdown -h now
```

Now, go to **DigitalOcean Control Panel**. Go to your **droplet**. Click on the **Snapshots** button to view the Snapshots section.



Take a snapshot

Enter a name and then take a snapshot! You may use this snapshot to restore your VPS later by using the **Restore from Snapshot** functionality.

Little tips

Tip 1:

If you have a domain and you want to connect it to your site, open the **server block** file, and edit this line:

```
server_name localhost;
```

Modify to:

```
server_name yourDomain.com;
```

Now you can be able to access your site via your domain.

Tip 2:

You can access your server using FTP as well (to upload, download files, etc.), use this information:

```
Host: IP address or your domain

User: root

Password: your password

Port: 22
```

Chapter 5 Summary

Congratulations! You now know how to deploy your Laravel applications!

I hope you like this chapter.

If you wish to learn more about Laravel, check our Laravel 5 Cookbook out!

Thank you again for your support! Have a great time!