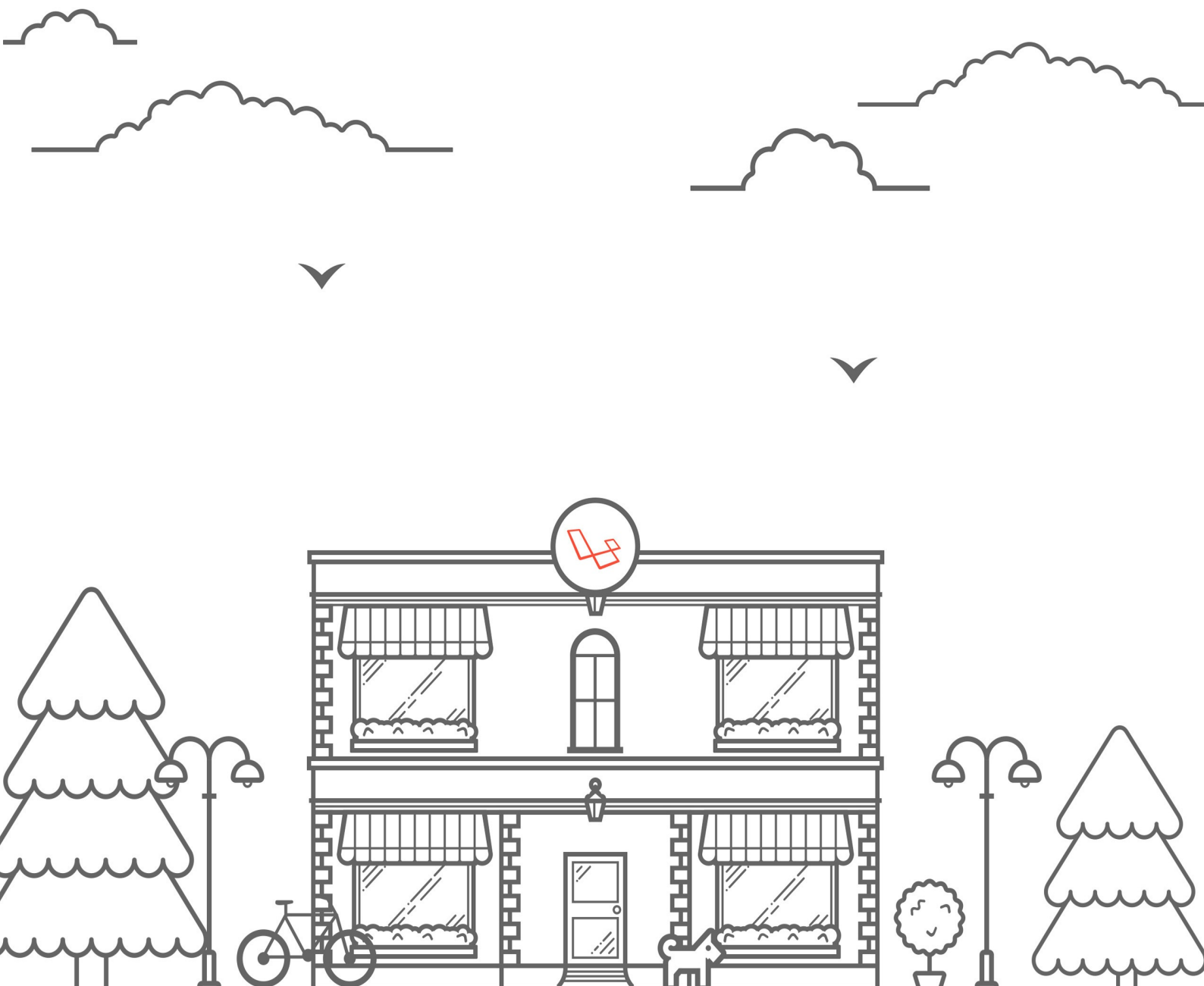
2020 EDITION

# Learning Laravel 6

## Building applications with Bootstrap 4

by Nathan Wu



# Table of Contents

## Part 1



## Part 2

# Part 3

# Part 4

# Part 5

# Part 6

# About This Book

Learning Laravel 6: Building Applications with Bootstrap 4 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 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 you'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, as 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 100000+ 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, which 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's status, changelog and view the list of contributors at:

Book Status

# Changelog

## Current Version

**Latest version the book:**

- Version: Laravel 6 (2020 Edition)
- Status: Complete. The book supports the latest version of Laravel and Bootstrap 4.
- Updated: October 20, 2019

# 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.

Alternatively, you may use **Cygwin**.

### 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 18.04
- Git
- PHP 7.3
- PHP 7.2

- PHP 7.1
- PHP 7.0
- PHP 5.6
- Nginx
- MySQL
- lmm for MySQL or MariaDB database snapshots
- Sqlite3
- PostgreSQL
- Composer
- Node (With Yarn, Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd
- Mailhog
- avahi
- ngrok
- Xdebug
- XHProf / Tideways / XHGui
- wp-cli

You may check out the **Homestead Documentation** at:

https://laravel.com/docs/master/homestead

## How to install Homestead?

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

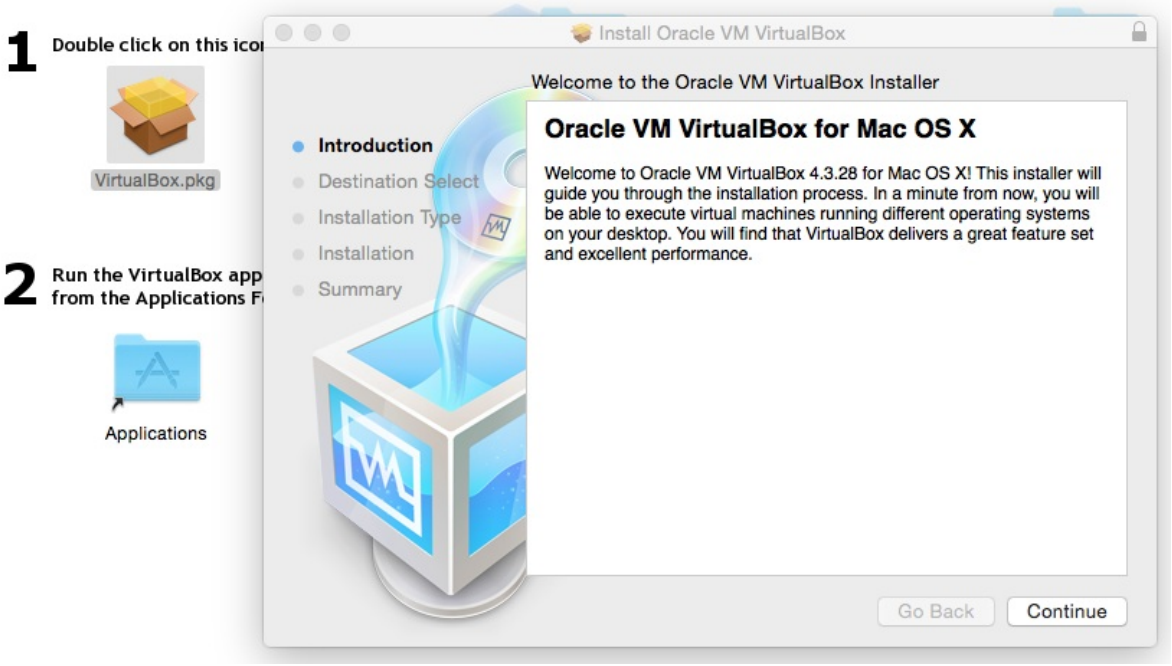
Choose a VirtualBox for your platform and install it.

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

The **stable release is version 6.0.12**. You can use a newer version if you want, but if you have any problems, try to use 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.



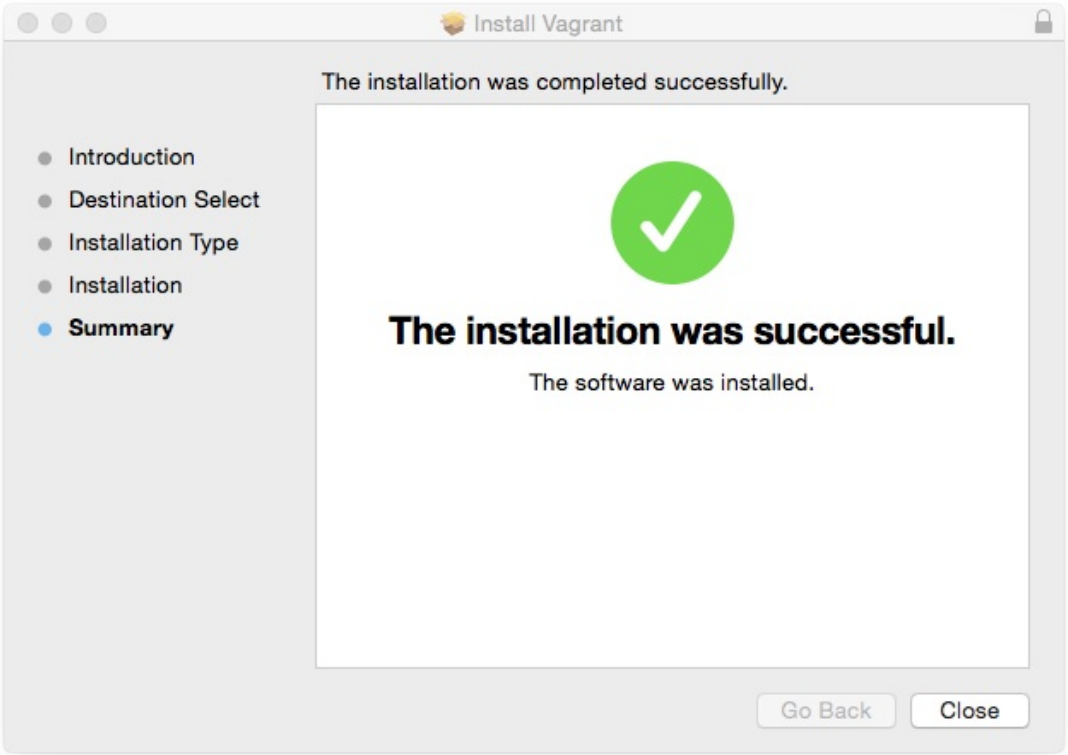
## Step 2 - Installing Vagrant

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

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

> **Note:** The stable release is **version 2.2.5**, which can be found at https://releases.hashicorp.com/vagrant/2.2.5/. You may use a newer version, but if you encounter any error, try to **reinstall version 2.2.5**.

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




If you still don't know how to install, there is an official guide on the 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 you're on a **Debian-based** distribution, use this:

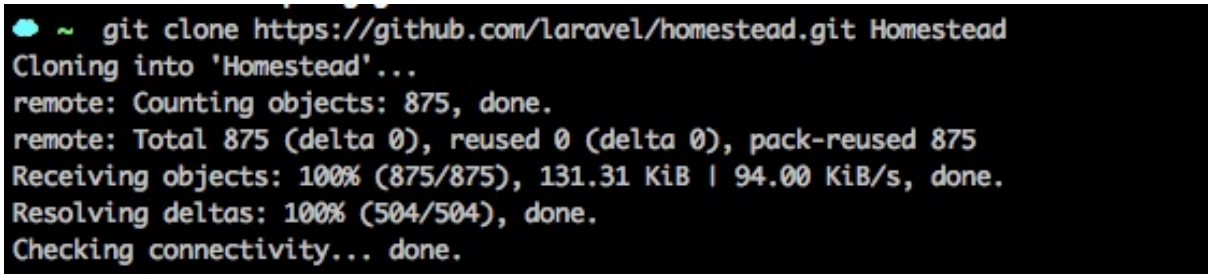
```
sudo apt-get install git
```

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
```




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

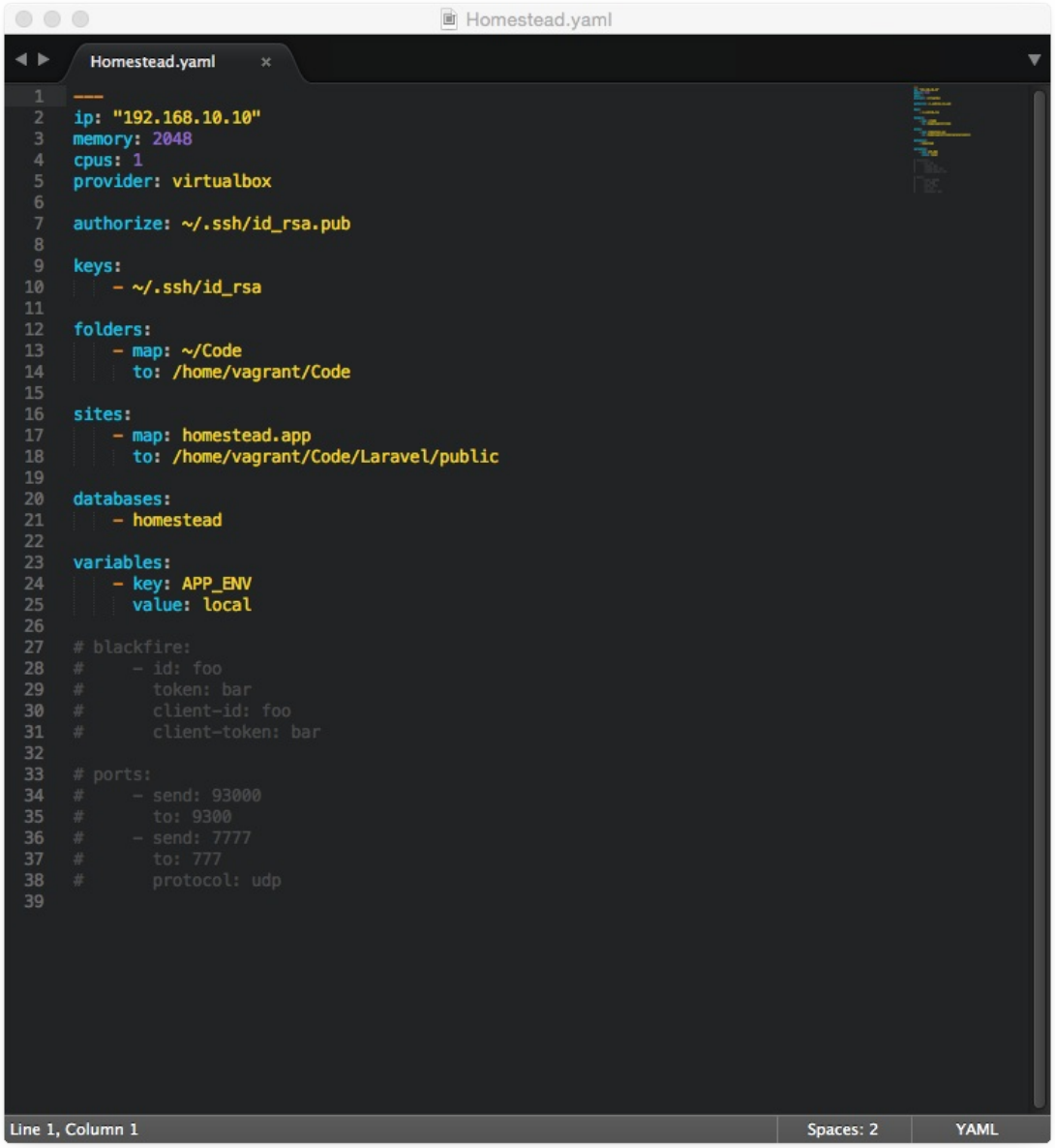
```
cd Homestead
```

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

```
bash init.sh
```

The `Homestead.yaml` file will be placed in the `Homestead` directory. Open it with a text editor to edit it.

> **Note:** In older versions of Homestead, the `Homestead.yaml` file will be placed in your `~/.homestead` directory. Please note that the `~/.homestead` directory **is hidden** by default, make sure that you can see hidden files.




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

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

Alternatively, you can use this command to open the file with your text editor:

```
open ~/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.

> **Note:** If you're using Windows, you may need to use a full path. For example:
> `- map: C:\Users\YourUSERNAME\Documents\Projects\Laravel\Homestead\Code`

Please note that everytime you change the path, you have to run these commands to reload and update Homestead:

```
vagrant halt
vagrant up --provision
```

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.test
      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.test` to the `public` folder of our Laravel project, and then we can easily access our Laravel app via this address: "http://homestead.test".

> **Note:** You can use the `.app` extension if you like (For example, http://homestead.app). However, if you're using the latest version of Google Chrome, you should change the `.app` extension to `.test` extension.

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.test
```

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

http://homestead.test

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

All sites will be accessible by HTTP via port `8000` and HTTPS via port `44300` by default (Homestead port).

## 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:

```
Bringing machine 'default' up with 'virtualbox' provider...
There are errors in the configuration of this machine. Please fix
the following errors and try again:

vm:
* The host path of the shared folder is missing: ~/Code
```

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

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

```
sudo mkdir ~/Code
```

To prevent possible errors when creating Laravel, try to **set right permissions** for the **Code folder** by running:

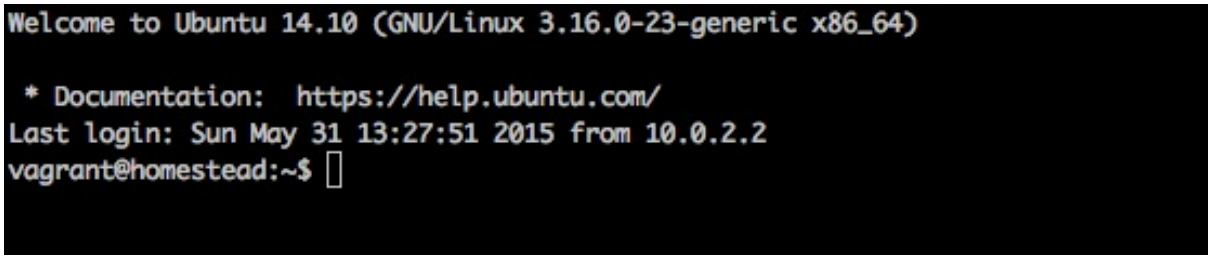
```
chmod -R 0777 ~/Code
```

If everything is going fine, we should see:

```
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'laravel/homestead'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'laravel/homestead' is up to date...
==> default: Setting the name of the VM: homestead
==> default: Fixed port collision for 22 => 2222. Now on port 2200.
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
    default: Adapter 2: hostonly
==> default: Forwarding ports...
```

Now we can access our VM using:

```
vagrant ssh
```




> **Note:** If it asks for a password, type `vagrant`.

To make sure that everything is ok, 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
```

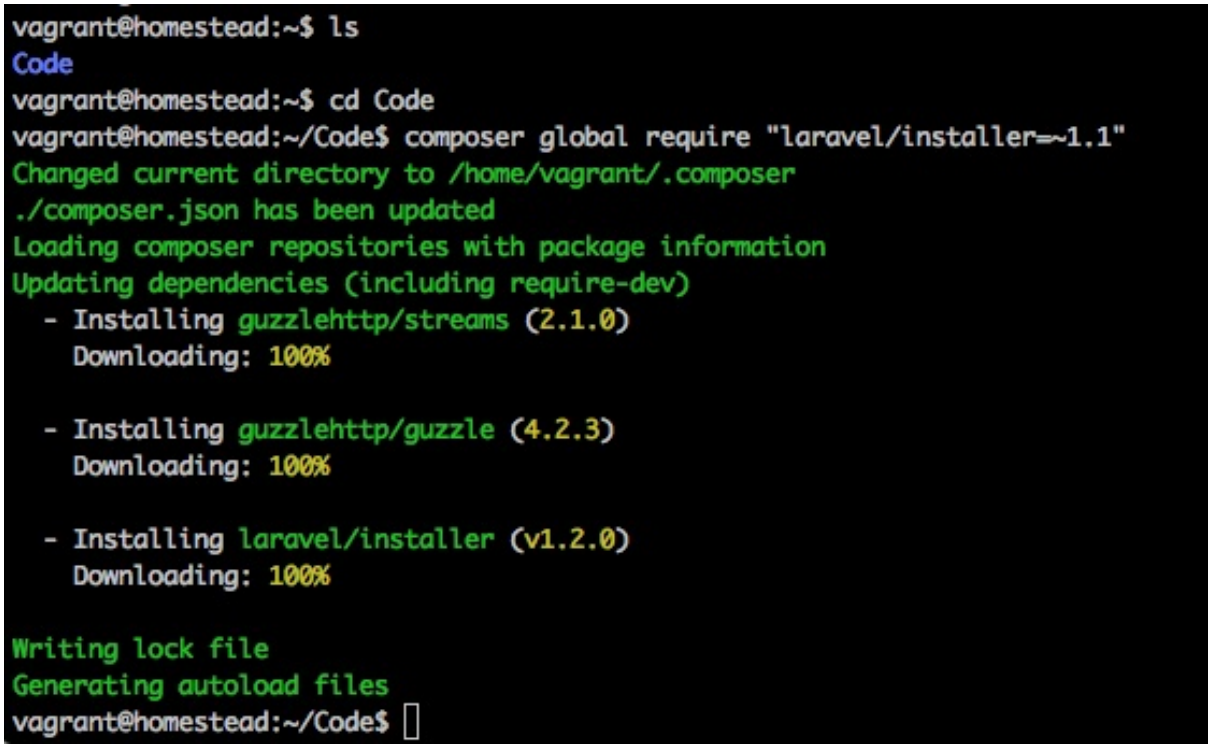
You should notice that the directory is empty. There are two methods to install Laravel.

### 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"
```




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

```
laravel new nameOfYourSite
```

**Laravel Installer** will download the **latest Laravel version** and install it. To install a **specific Laravel version**, you may use this command instead:

```
laravel new nameOfYourSite --6.2
```

This command is used to download **Laravel 6.2**.

> **Note:** If the latest version of Laravel is **6.2**, you can't run this command. You may install Laravel 6.1 instead, by using the `--6.1` flag. It is recommended to use Laravel 6.2 to learn the basics of the Laravel Framework. You can upgrade to a newer version later.

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.test
      to: /home/vagrant/Code/Laravel/public
```

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

```
laravel new Laravel
```

You should see something like this:


> **Note:** Please note that the name is **L**aravel, not **l**aravel.

If you see this error when using the `laravel new` command:

```
laravel: command not found
```

We have to edit the **.bashrc** file, type:

```
nano ~/.bashrc
```

Add this line at end of the file:

```
alias laravel='~/.config/composer/vendor/bin/laravel'
```

Press `Ctrl + X`, then `Y`, then `Enter` to **exit and save** the file.

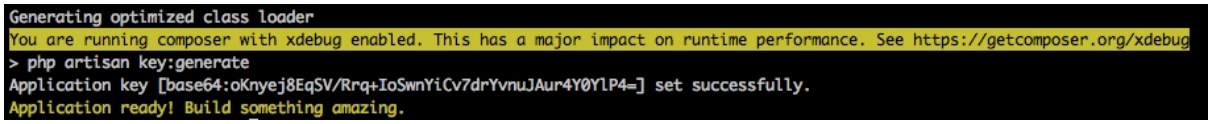
Lastly, run this command:

```
source ~/.bashrc
```

Now you should be able to create a new Laravel app using:

```
laravel new Laravel
```

You should see this:




Next, open your web browser and go to http://homestead.test


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

## Install Laravel using 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 "~5.7.7"
```

This command is used to download **Laravel 5.7.7**, which is a stable version. If you want to use the latest version, use:

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

> **Note:** It is recommended to use Laravel 5.7 to learn the basics of Laravel Framework. You can upgrade to a newer version later.

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.test
      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:

```
Writing lock file
Generating autoload files
> Illuminate\Foundation\ComposerScripts::postUpdate
> php artisan optimize
Generating optimized class loader
> php artisan key:generate
Application key [base64:BMFMKOvcEp0homAtj+uR5xwBLyR1IZf52Zo2HNnOrhM=] set successfully.
```

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


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

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

## Checking Laravel version

We can check what version of Laravel that we've installed by simply running this command **at the root of our application**:

```
php artisan --version
```

or

```
php artisan -V
```

A line will be printed out:

```
Laravel Framework 6.2.0
```

As you see, I'm using **Laravel 6.2.0**.

> **Important note**: If you're not using Laravel 6.2, please download and install Laravel 6.2 to avoid possible errors.

# Updating Homestead box

Sometimes, when we run the `vagrant up` command, we might see this message:

```
==> default: Checking if box 'laravel/homestead' is up to date...
==> default: A newer version of the box 'laravel/homestead' is available! You currentl
y
==> default: have version '0.4.4'. The latest is version '2.1.0'. Run
==> default: `vagrant box update` to update.
```

As you may have noticed, this means a newer version of Homestead box is available. We can update the box by running this command:

```
vagrant box update
```

You'll see something like this:

```
==> default: Updating 'laravel/homestead' with provider 'vmware_desktop' from version
==> default: '0.4.4' to '2.1.0'...
==> default: Loading metadata for box 'https://atlas.hashicorp.com/laravel/homestead'
==> default: Adding box 'laravel/homestead' (v2.1.0) for provider: vmware_desktop
    default: Downloading: https://atlas.hashicorp.com/laravel/boxes/homestead/versions
/2.1.0/providers/vmware_desktop.box
    default: Progress: 64% (Rate: 863k/s, Estimated time remaining: 0:06:47)
```

> **Note:** Be sure to backup your files and databases first. It may take a long time to complete.

After that, `vagrant ssh` into your Homestead and run:

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

If it asks anything, type `y`.

Please note that the **master branch** (the latest version) may not always be stable. If your app is running fine, you don't have to update Homestead.

# Updating Homestead using Git

If you want to upgrade your Homestead to the latest version or use another version of Homestead, you may use this method.

First, please backup the **Homestead.yaml** file and your database first to ensure that you won't lose any data.

Go to your homestead **root directory**:

```
cd Homestead
```

> **Note:** Your path could be different.

Next, run this command to get a list of Homestead versions:

```
git fetch origin
```

You'll see something like this:

```
remote: Counting objects: 19, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 19 (delta 11), reused 12 (delta 11), pack-reused 5
Unpacking objects: 100% (19/19), done.
From https://github.com/laravel/homestead
   393c4bd..74749a5  master     -> origin/master
 * [new tag]         v7.2.0     -> v7.2.0
 * [new tag]         v7.3.0     -> v7.3.0
```

After that, pick the version that you like.

```
git checkout v9.2.2
```

Currently, the stable version is `v9.2.2`.

Run these commands to destroy and update your vagrant:

```
vagrant destroy
rm -rf .vagrant
vagrant up
```

> **Note:** You might need to download and install the latest version of Vagrant.

Done! You're good to go!

> **Note:** In this book, we'll be using Homestead `v9.2.2` (which is a stable version), so please use the same version to avoid possible errors.

# Generate new application key

Sometimes, you might see this error when generating a new application:

```
"No application encryption key has been specified."
```

To fix this, you just need to update the Laravel Installer by running this command:

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

After that, simply generate a new **application key**:

```
php artisan key:generate
```

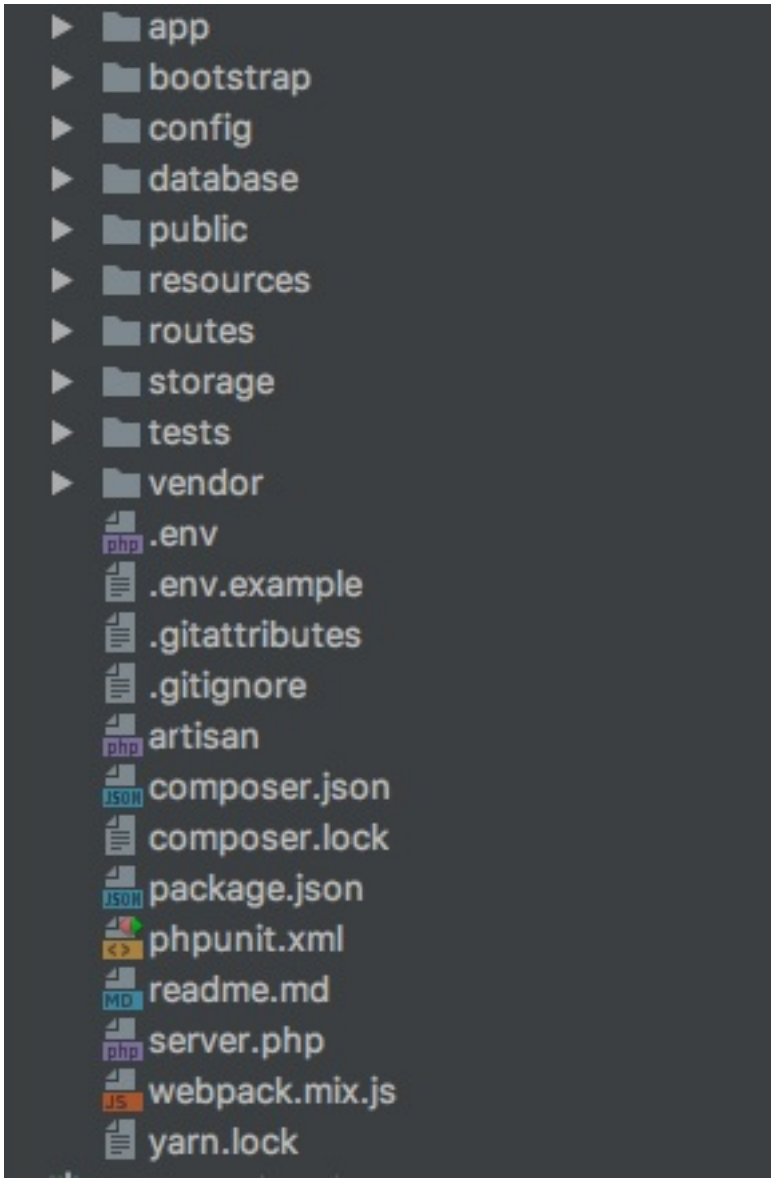
This should fix the bug.

# 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.




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 **these folders**.

1. app
2. bootstrap
3. config
4. database
5. public
6. resources

7. routes
8. storage
9. tests
10. 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.

**Routes**

All our **route files** will be stored here.

We'll learn about this **routes directory** in the next section.

**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:

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

# Understand the routes directory

One of the most important features of Laravel is **"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.test**, 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 something!"**

We usually register all routes in the **routes.php** file, but since Laravel 5.3, the **routes.php** file has been moved to the new **routes** directory. We now have three files: **web.php**, **console.php** and **api.php**.

# Changing Laravel home page

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

Let's see what's inside the file.

**routes/web.php**

```
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 **/**).

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:

**resources/views/welcome.blade.php**

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>Laravel</title>

        <!-- Fonts -->
        <link href="https://fonts.googleapis.com/css?family=Nunito:200,600" rel="style
sheet">

        <!-- Styles -->
        <style>
            html, body {
                background-color: #fff;
```

```
                color: #636b6f;
                font-family: 'Nunito', sans-serif;
                font-weight: 200;
                height: 100vh;
                margin: 0;
            }

            .full-height {
                height: 100vh;
            }

            .flex-center {
                align-items: center;
                display: flex;
                justify-content: center;
            }

            .position-ref {
                position: relative;
            }

            .top-right {
                position: absolute;
                right: 10px;
                top: 18px;
            }

            .content {
                text-align: center;
            }

            .title {
                font-size: 84px;
            }

            .links > a {
                color: #636b6f;
                padding: 0 25px;
                font-size: 13px;
                font-weight: 600;
                letter-spacing: .1rem;
                text-decoration: none;
                text-transform: uppercase;
            }

            .m-b-md {
                margin-bottom: 30px;
            }
        </style>
    </head>
    <body>
        <div class="flex-center position-ref full-height">
            @if (Route::has('login'))
```

```
                <div class="top-right links">
                    @auth
                        <a href="{{ url('/home') }}">Home</a>
                    @else
                        <a href="{{ route('login') }}">Login</a>

                        @if (Route::has('register'))
                            <a href="{{ route('register') }}">Register</a>
                        @endif
                    @endauth
                </div>
            @endif

            <div class="content">
                <div class="title m-b-md">
                    Laravel
                </div>

                <div class="links">
                    <a href="https://laravel.com/docs">Docs</a>
                    <a href="https://laracasts.com">Laracasts</a>
                    <a href="https://laravel-news.com">News</a>
                    <a href="https://blog.laravel.com">Blog</a>
                    <a href="https://nova.laravel.com">Nova</a>
                    <a href="https://forge.laravel.com">Forge</a>
                    <a href="https://vapor.laravel.com">Vapor</a>
                    <a href="https://github.com/laravel/laravel">GitHub</a>
                </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.

> **Tip:** 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 homepage!

What should we do?...

How about display a custom quote?

Let's replace these lines:

```
<div class="links">
    <a href="https://laravel.com/docs">Docs</a>
    <a href="https://laracasts.com">Laracasts</a>
    <a href="https://laravel-news.com">News</a>
    <a href="https://blog.laravel.com">Blog</a>
    <a href="https://nova.laravel.com">Nova</a>
    <a href="https://forge.laravel.com">Forge</a>
    <a href="https://vapor.laravel.com">Vapor</a>
    <a href="https://github.com/laravel/laravel">GitHub</a>
</div>
```

with:

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

Our new **welcome view** should look like this:

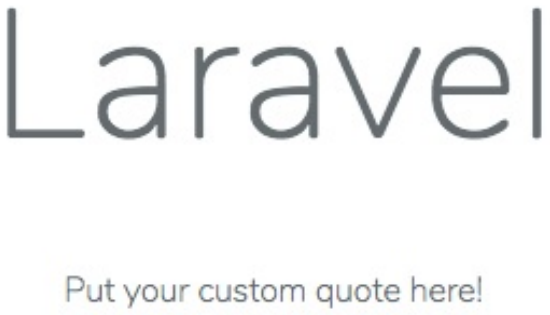
```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>Laravel</title>

        <!-- Fonts -->
        <link href="https://fonts.googleapis.com/css?family=Nunito:200,600" rel="style
sheet">

        <!-- Styles -->
        <style>
            html, body {
                background-color: #fff;
                color: #636b6f;
                font-family: 'Nunito', sans-serif;
                font-weight: 200;
                height: 100vh;
                margin: 0;
            }

            .full-height {
                height: 100vh;
            }

            .flex-center {
                align-items: center;
                display: flex;
                justify-content: center;
            }
```

```
            .position-ref {
                position: relative;
            }

            .top-right {
                position: absolute;
                right: 10px;
                top: 18px;
            }

            .content {
                text-align: center;
            }

            .title {
                font-size: 84px;
            }

            .links > a {
                color: #636b6f;
                padding: 0 25px;
                font-size: 13px;
                font-weight: 600;
                letter-spacing: .1rem;
                text-decoration: none;
                text-transform: uppercase;
            }

            .m-b-md {
                margin-bottom: 30px;
            }
        </style>
    </head>
    <body>
        <div class="flex-center position-ref full-height">
            @if (Route::has('login'))
                <div class="top-right links">
                    @auth
                        <a href="{{ url('/home') }}">Home</a>
                    @else
                        <a href="{{ route('login') }}">Login</a>

                        @if (Route::has('register'))
                            <a href="{{ route('register') }}">Register</a>
                        @endif
                    @endauth
                </div>
            @endif

            <div class="content">
                <div class="title m-b-md">
                    Laravel
```

```
                </div>

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

            </div>
        </div>
    </body>
</html>
```

> **Note:** Laravel welcome view is changed frequently. If your view is different, just copy the code above to **create a new welcome view**.

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




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

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

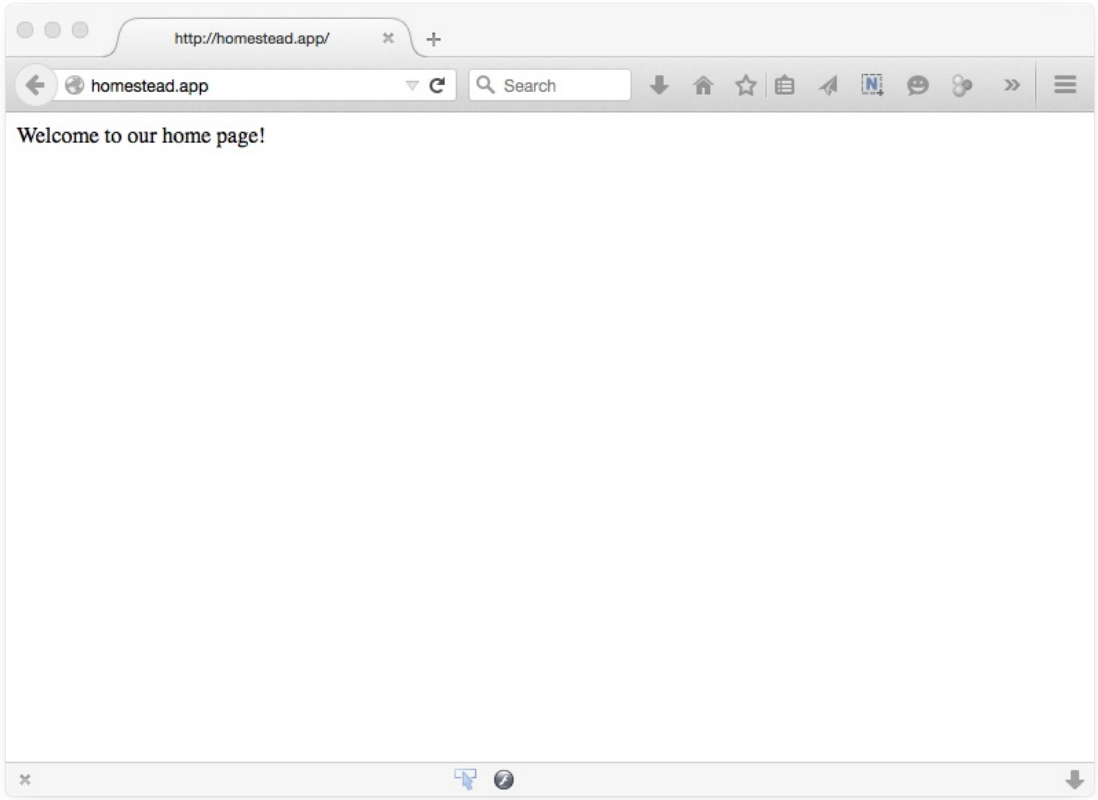
Modify the first route:

**routes/web.php**

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

to

```
Route::get('/', function()
{
return 'Welcome to our 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 the Learning Laravel book. Our 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 to 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 favorite text editor (PHPStorm, Sublime Text, etc.).

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

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

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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"**:

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

## Create a controller by using Artisan

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

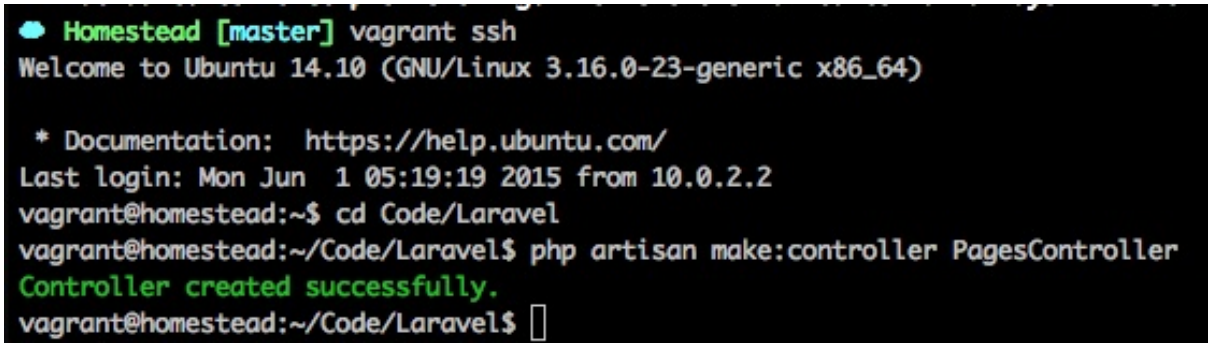
**Artisan** is Laravel command line tool that helps us to perform tasks that we hate to do manually. Using Artisan, we can create models, views, controllers, migrations and many other things.

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. Delete the old controller if you've created it.

You'll see:




By default, Laravel creates a plain controller:

**app/Http/Controllers/PagesController.php**

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class PagesController extends Controller
{
    //
}
```

Next, let's add a new a home action:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class PagesController extends Controller
{
    public function home()
    {
        return view('welcome');
    }
}
```

Alternatively, you can use this command to generate a new **RESTful PagesController**:

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

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

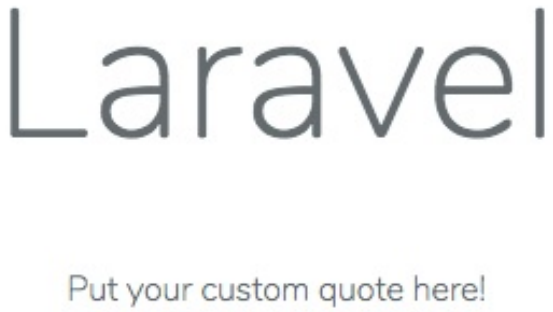
Cool! When you have **PagesController**, the next thing to do is modifying our **routes**!

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

**routes/web.php**

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

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




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.

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

Edit the **web.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');
}
```

Here is the updated **PagesController.php** file:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class PagesController extends Controller
{
    public function home()
    {
        return view('welcome');
    }

    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:

**resources/views/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>
```

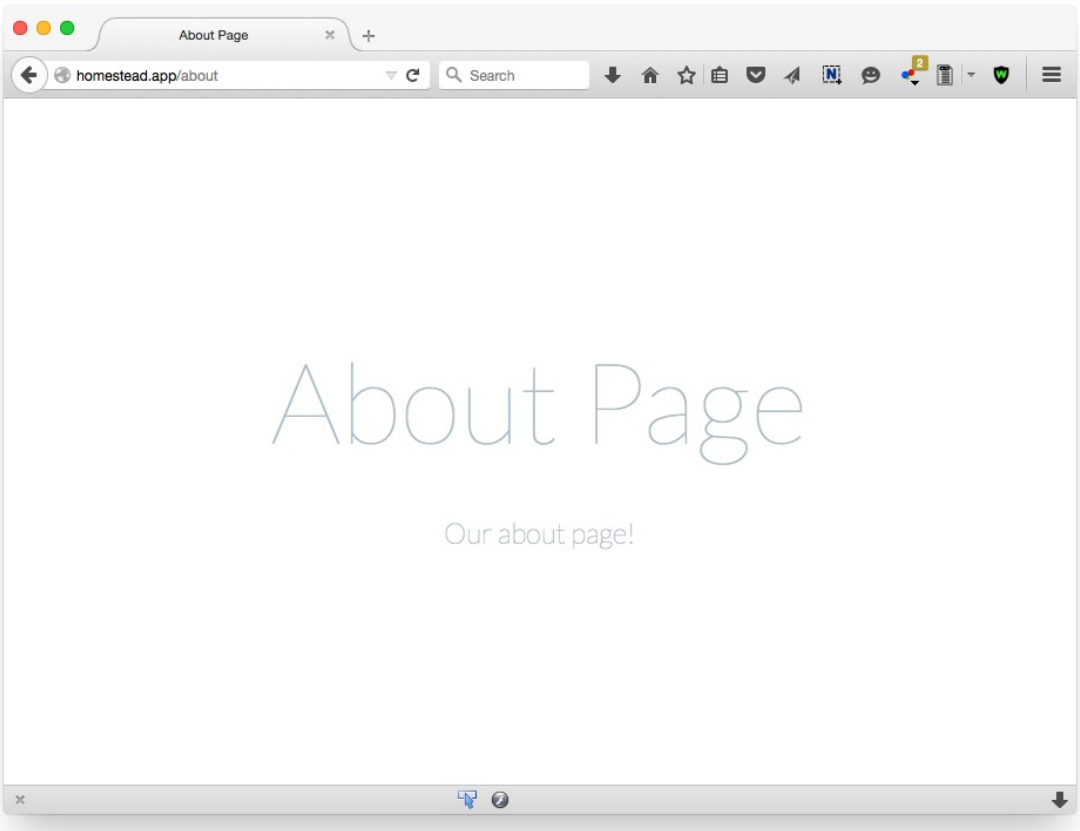
```
        <div class="container">
            <div class="content">
                <div class="title">About Page</div>
                <div class="quote">Our about page!</div>
            </div>
        </div>
    </body>
</html>
```

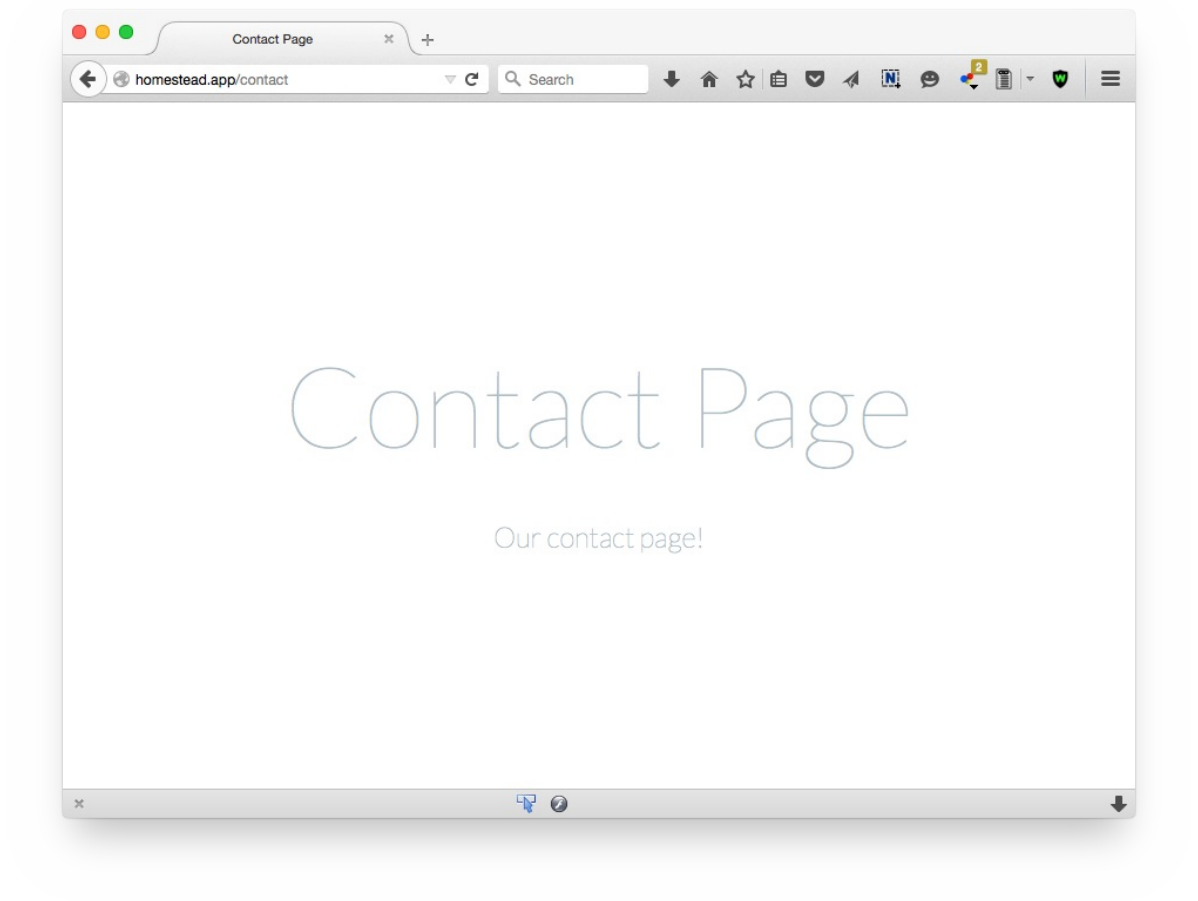
**resources/views/contact.blade.php**

```
<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.test/about** and **homestead.test/contact**:






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:

**resources/views/home.blade.php**

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

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

Find:

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

Replace with:

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

# Integrating Bootstrap

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

Using 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 Bootstrap into our Laravel application.

You can get Bootstrap and read its official documentation here:

https://getbootstrap.com/

> **Note:** We're using the new Bootstrap 4 in this book. If you want to use Bootstrap 3, check the old version of the book.

There are many ways to integrate 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 (Sass)

You can choose to use any method that you like.

By the way, before going to the next section, you may notice that we've added some CSS styles and Lato font into our **home.blade.php** file before. **Remove** those first:

```
<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='tex
t/css'>
```

## Using Bootstrap CDN

The fastest way to integrate Bootstrap is using CDN.

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

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bo
otstrap.min.css">
```

Find:

```
</body>
```

Add above:

```
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"></script>
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"></
script>
```

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

> **Note:** All the links can be found at https://getbootstrap.com/docs/4.1/getting-started/introduction/

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

```
<html>
<head>
    <title>Home Page</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/cs
s/bootstrap.min.css">
</head>
<body>
<div class="container">
    <div class="content">
        <div class="title">Home Page</div>
        <div class="quote">Our Home page!</div>
    </div>
</div>
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"></
script>
</body>
</html>
```

## Using Precompiled Bootstrap Files

Bootstrap 4 is built using Sass - a CSS pre-processor.

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

https://sass-lang.com/

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

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

https://getbootstrap.com/docs/4.3/getting-started/download/

> **Note:** I'm using **version 4.3** here. You may use a newer version if you want.

Find the **"Compiled CSS and JS"** section and click **Download** to download the latest compiled Bootstrap files.

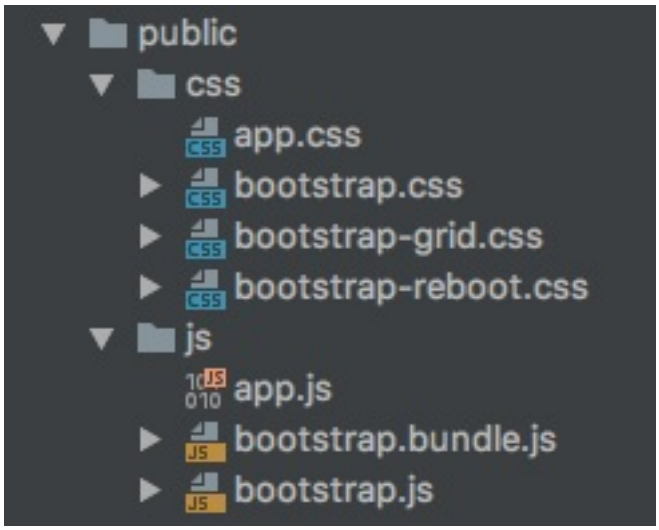
Uncompress the downloaded .zip file. We have three folders:

1. css
2. js

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

> Note: By default, Laravel has created **css** and **js** folder for us.

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




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

```
<link rel="stylesheet" href="{!! asset('css/bootstrap.min.css') !!}" >
```

Find:

```
</body>
```

Add above:

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

Twitter Bootstrap requires **jQuery** and **Popper.js** to work properly.

You can download **jQuery** here:

https://jquery.com/download

You can download **Popper.js** here:

https://unpkg.com/popper.js/dist/umd/popper.min.js

Put the downloaded files into the **public** directory as well, then use the following code to reference them:

```
<script src="{!! asset('js/jquery-3.3.1.min.js') !!}"></script>
<script src="{!! asset('js/popper.min.js') !!}"></script>
```

> **Note:** Your jQuery version may be different.

Here is our updated homepage:

**home.blade.php**

```
<html>
<head>
    <title>Home Page</title>
        <link rel="stylesheet" href="{!! asset('css/bootstrap.min.css') !!}">
</head>
<body>

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

<script src="{!! asset('js/jquery-3.3.1.min.js') !!}"></script>
<script src="{!! asset('js/popper.min.js') !!}"></script>
<script src="{!! asset('js/bootstrap.min.js') !!}"></script>

</body>
</html>
```

Done! You now have fully integrated **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" href="/css/bootstrap.min.css" >

<script src="/js/jquery-3.3.1.min.js"></script>
<script src="/js/popper.min.js"></script>
<script src="js/bootstrap.min.js"></script>
```

## Using Bootstrap Source Code (Sass)

The good news is, the latest version of Laravel has officially supported **Sass**. Sass files are placed at **resources/assets/sass**.

We'll use **Laravel Mix** to automatically compile Sass files to **app.css** file.

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

```
<link rel="stylesheet" href="/css/app.css">
```

Find:

```
</body>
```

Add above:

```
<script src="/js/app.js"></script>
```

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

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

```
<html>
<head>
    <title>Home Page</title>
    <link rel="stylesheet" href="/css/app.css">
</head>
```

```
<body>

<div class="container">
    <div class="content">
        <div class="title">Home Page</div>
        <div class="quote">Our Home page!</div>
    </div>
</div>
<script src="/js/app.js"></script>
</body>
</html>
```

Oh, I've mentioned **Laravel Mix**, what is it?

## Introducing Laravel Mix

One of the best Laravel features is **Laravel Mix**. We can use **Laravel Mix** to compile Sass files, Coffee Scripts or automate other manual tasks.

> **Note:** Laravel Mix is a new feature of Laravel 5.4+. In older versions of Laravel, we have **Elixir**, which is pretty similar.

Laravel Mix official documentation

Basically, Laravel Mix is based on **Webpack**. If you don't know about **Webpack** yet, you can find more information about it here:

Webpack Official Home Page

To use Laravel Mix, we must have **NPM**(Node Package Manager) and **Node.js** installed.

Luckily, Homestead has NPM and Node.js by default, we can use Laravel Mix right away.

If you don't use Homestead, you have to install Node.js and NPM.

> **Tip:** Actually, you should run Node.js/Laravel Mix directly on your local machine (Windows, Mac, etc.). It's faster and less buggy.

To install Node.js, visit:

https://nodejs.org

> **Note:** Be sure to use Node.js v10.11.0 or newer. The direct download links can also be found at: https://nodejs.org/en/blog/release/v10.11.0

Follow instructions on the site, you should have installed Node.js easily.

You may check the version of Node.js by running:

```
node -v
```

You may check the version of NPM by running:

```
npm -v
```

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

**package.json**

```
{
    "private": true,
    "scripts": {
        "dev": "npm run development",
        "development": "cross-env NODE_ENV=development node_modules/webpack/bin/webpac
k.js --progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config.
js",
        "watch": "npm run development -- --watch",
        "watch-poll": "npm run watch -- --watch-poll",
        "hot": "cross-env NODE_ENV=development node_modules/webpack-dev-server/bin/web
pack-dev-server.js --inline --hot --config=node_modules/laravel-mix/setup/webpack.conf
ig.js",
        "prod": "npm run production",
        "production": "cross-env NODE_ENV=production node_modules/webpack/bin/webpack.
js --no-progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config
.js"
    },
    "devDependencies": {
        "axios": "^0.19",
        "cross-env": "^5.1",
        "laravel-mix": "^4.0.7",
        "lodash": "^4.17.13",
        "resolve-url-loader": "^2.3.1",
        "sass": "^1.15.2",
        "sass-loader": "^7.1.0"
    }
}
```

Do you see the `laravel-mix` there?

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

```
npm install
```

> **Note:** If you install Node.js on your local machine, you should run this command directly on your machine (Windows, macOS, etc.).

Windows system users may need to run this command instead:

```
npm install --no-bin-links
```

Mac users may need to run this command instead (if the above command doesn't work):

```
sudo npm install
```


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 Laravel Mix and other Node.js packages here.

## Running first Laravel Mix task

You can write a new Mix task in **webpack.mix.js**. By default, you can find a Mix task that compiles **app.sass** file into **app.css**, and bundles all JS files there:

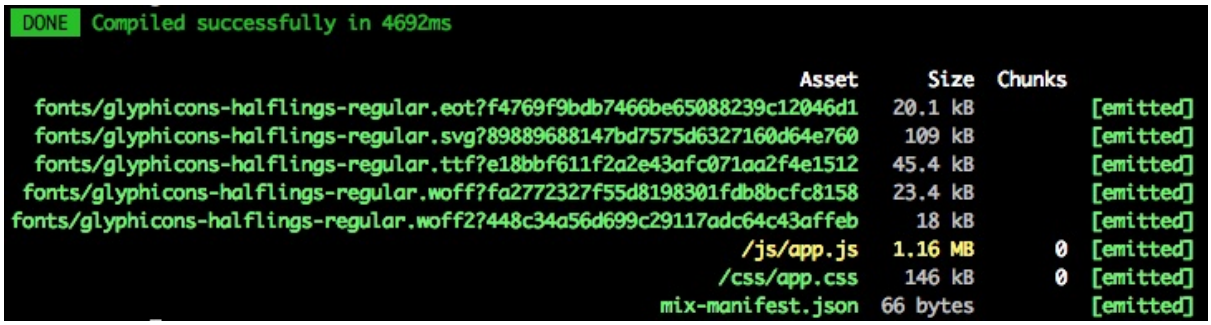
```
mix.js('resources/assets/js/app.js', 'public/js')
   .sass('resources/assets/sass/app.scss', 'public/css');
```

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

https://laravel.com/docs/master/mix

To execute your Mix task, run this command:

```
npm run dev
```




By running the task, Laravel will automatically compile the app.sass file and bundle all JS files. The output files can be found in your **public** directory.

# Adding Bootstrap components

Cool! Now we can add **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:

https://getbootstrap.com/docs/4.3/components/alerts/

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

Find:

```
<body>
```

Add below:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Navbar</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#na
vbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" ari
a-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>

  <div class="collapse navbar-collapse" id="navbarSupportedContent">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item active">
        <a class="nav-link" href="#">Home <span class="sr-only">(current)</span></a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Link</a>
      </li>
      <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button"
data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
          Dropdown
        </a>
        <div class="dropdown-menu" aria-labelledby="navbarDropdown">
          <a class="dropdown-item" href="#">Action</a>
          <a class="dropdown-item" href="#">Another action</a>
          <div class="dropdown-divider"></div>
          <a class="dropdown-item" href="#">Something else here</a>
        </div>
      </li>
      <li class="nav-item">
        <a class="nav-link disabled" href="#">Disabled</a>
      </li>
    </ul>
    <form class="form-inline my-2 my-lg-0">
      <input class="form-control mr-sm-2" type="search" placeholder="Search" aria-label
="Search">
      <button class="btn btn-outline-success my-2 my-sm-0" type="submit">Search</button
>
    </form>
  </div>
</nav>
```

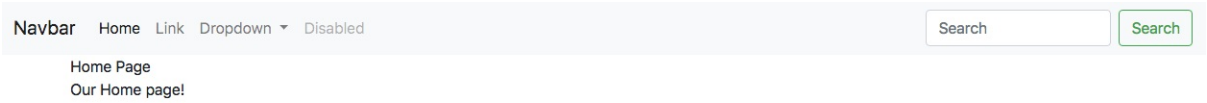
Here is our new **home.blade.php** after adding the navbar:

### home.blade.php

```
<html>
<head>
    <title>Home Page</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/cs
s/bootstrap.min.css">
</head>
<body>
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#
navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" a
ria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item active">
                <a class="nav-link" href="#">Home <span class="sr-only">(current)</span
></a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">Link</a>
            </li>
            <li class="nav-item dropdown">
                <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="
button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
                    Dropdown
                </a>
                <div class="dropdown-menu" aria-labelledby="navbarDropdown">
                    <a class="dropdown-item" href="#">Action</a>
                    <a class="dropdown-item" href="#">Another action</a>
                    <div class="dropdown-divider"></div>
                    <a class="dropdown-item" href="#">Something else here</a>
                </div>
            </li>
            <li class="nav-item">
                <a class="nav-link disabled" href="#">Disabled</a>
            </li>
        </ul>
        <form class="form-inline my-2 my-lg-0">
            <input class="form-control mr-sm-2" type="search" placeholder="Search" ari
a-label="Search">
            <button class="btn btn-outline-success my-2 my-sm-0" type="submit">Search</
button>
        </form>
    </div>
</nav>

<div class="container">
```

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

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"></
script>
</body>
</html>
```

It's time to refresh our browser:




Home Page
Our Home page!

# Responsive meta tag

To make our site more responsive to display well across all devices, we should put a responsive meta tag in the head section of our template:

Find:

```
<head>
```

Add below:

```
<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
```

> **Note:** this step is optional, but it's recommended to include the tag when using Bootstrap.

# 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 in 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**

**resources/views/master.blade.php**

```
<html>
<head>
    <title> @yield('title') </title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/cs
s/bootstrap.min.css">
</head>
<body>
@include('shared.navbar')

@yield('content')

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"></
script>
</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 embedded 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**:

**resources/views/shared/navbar.blade.php**

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#
navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" a
ria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item active">
                <a class="nav-link" href="#">Home <span class="sr-only">(current)</span
></a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">Link</a>
            </li>
            <li class="nav-item dropdown">
                <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="
button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
                    Dropdown
                </a>
                <div class="dropdown-menu" aria-labelledby="navbarDropdown">
                    <a class="dropdown-item" href="#">Action</a>
                    <a class="dropdown-item" href="#">Another action</a>
                    <div class="dropdown-divider"></div>
                    <a class="dropdown-item" href="#">Something else here</a>
                </div>
            </li>
            <li class="nav-item">
                <a class="nav-link disabled" href="#">Disabled</a>
            </li>
        </ul>
        <form class="form-inline my-2 my-lg-0">
            <input class="form-control mr-sm-2" type="search" placeholder="Search" ari
a-label="Search">
            <button class="btn btn-outline-success my-2 my-sm-0" type="submit">Search</
button>
        </form>
    </div>
</nav>
```

> **Note:** 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.

**resources/views/home.blade.php**

```
@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.

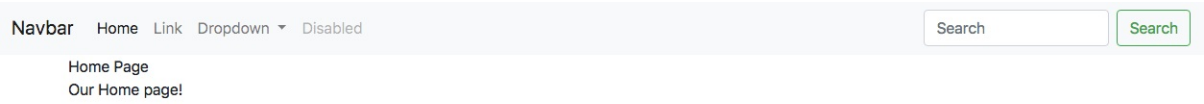
```
@section('title', 'Home')
```

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.




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

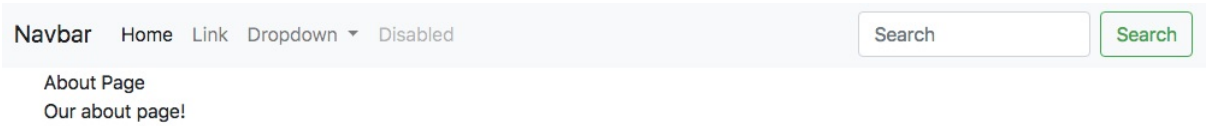
**about.blade.php**

```
@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
```




About Page
Our about page!

## Using other Bootstrap themes

The best thing about 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

If you like a theme, you can change the layout to that theme by following the instructions on its website.

> **Note:** In the old versions of this book, we've learned how to apply **Bootstrap Material Design** theme for our website. However, I think it's best to learn Laravel using the default Bootstrap 4 theme first, so we won't use another theme in this book.

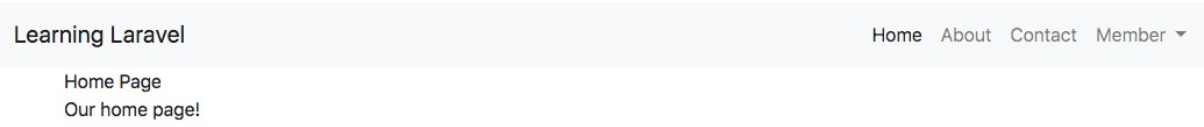
## Refine our website layouts

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**, and update it:

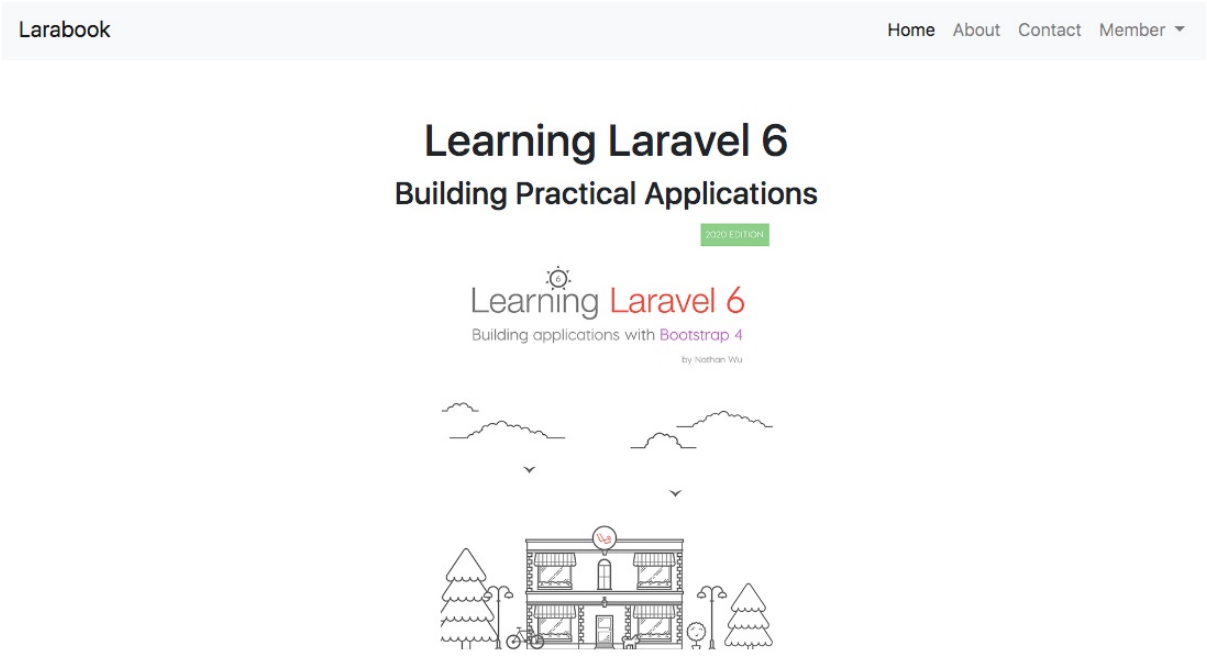
```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="#">Larabook</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#
navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" a
ria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav ml-auto">
            <li class="nav-item active">
                <a class="nav-link" href="/">Home <span class="sr-only">(current)</span
></a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="/about">About</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="/contact">Contact</a>
            </li>
            <li class="nav-item dropdown">
                <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="
button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
                    Member
                </a>
                <div class="dropdown-menu dropdown-menu-right" aria-labelledby="navbar
Dropdown">
                    <a class="dropdown-item" href="/users/register">Register</a>
                    <a class="dropdown-item" href="/users/login">Login</a>
                </div>
            </li>
        </ul>
    </div>
</nav>
```




## Changing the home page

Open **home.blade.php**, and 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 mt-5 editContent">
                    Learning Laravel 6
                </h1>

                <h3 class="text-center mt-2 editContent">Building Practical Applicatio
ns</h3>

                <div class="text-center">
                    <img src="https://learninglaravel.net/img/LearningLaravel5_cover7.
png" width="302" height="391" alt="">
                </div>

            </div>

        </div>
    </div>
@endsection
```




Try to resize your browser, you'll notice that we now have a cool responsive home page.


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 Laravel Mix, how to install Node.js and how to create a basic Mix 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 2 Source Code

You can view and download the source code at:

> Learning Laravel 6 Book: Chapter 2 Source Code

# 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
5. MariaDB

> **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 [

    /*
    |--------------------------------------------------------------------------
    | 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' => env('DB_DATABASE', database_path('database.sqlite')),
            'prefix' => '',
        ],

        'mysql' => [
            'driver' => 'mysql',
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '3306'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'unix_socket' => env('DB_SOCKET', ''),
            'charset' => 'utf8mb4',
            'collation' => 'utf8mb4_unicode_ci',
            'prefix' => '',
            'strict' => true,
            'engine' => null,
        ],

        'pgsql' => [
            'driver' => 'pgsql',
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '5432'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'charset' => 'utf8',
            'prefix' => '',
            'schema' => 'public',
            'sslmode' => 'prefer',
        ],

        'sqlsrv' => [
            'driver' => 'sqlsrv',
            'host' => env('DB_HOST', 'localhost'),
            'port' => env('DB_PORT', '1433'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'charset' => 'utf8',
```

```
            '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' => [

        'client' => 'predis',

        'default' => [
            'host' => env('REDIS_HOST', '127.0.0.1'),
            'password' => env('REDIS_PASSWORD', null),
            'port' => env('REDIS_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**.

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
```

# MySQL Strict Mode

In new versions of Laravel, **MySQL Strict Mode** is `enabled` by default.

> **Information**: If you want to know what **Strict Mode** is, check out this article: Upgrading to MySQL 5.7? Beware of the new STRICT mode

In some cases, you might want to turn the **Strict Mode** `off`. Here's how we can do it:

Find:

```
        'mysql' => [
            'driver' => 'mysql',
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '3306'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'unix_socket' => env('DB_SOCKET', ''),
            'charset' => 'utf8mb4',
            'collation' => 'utf8mb4_unicode_ci',
            'prefix' => '',
            'strict' => true,
            'engine' => null,
        ],
```

Change **strict** from `true` to `false`:

```
'strict' => false,
```

> **Note:** If you encounter some database errors, be sure to try to turn the **Strict Mode** off first. It's still safe to turn the **Strict Mode** off.

# 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 -u homestead -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
```

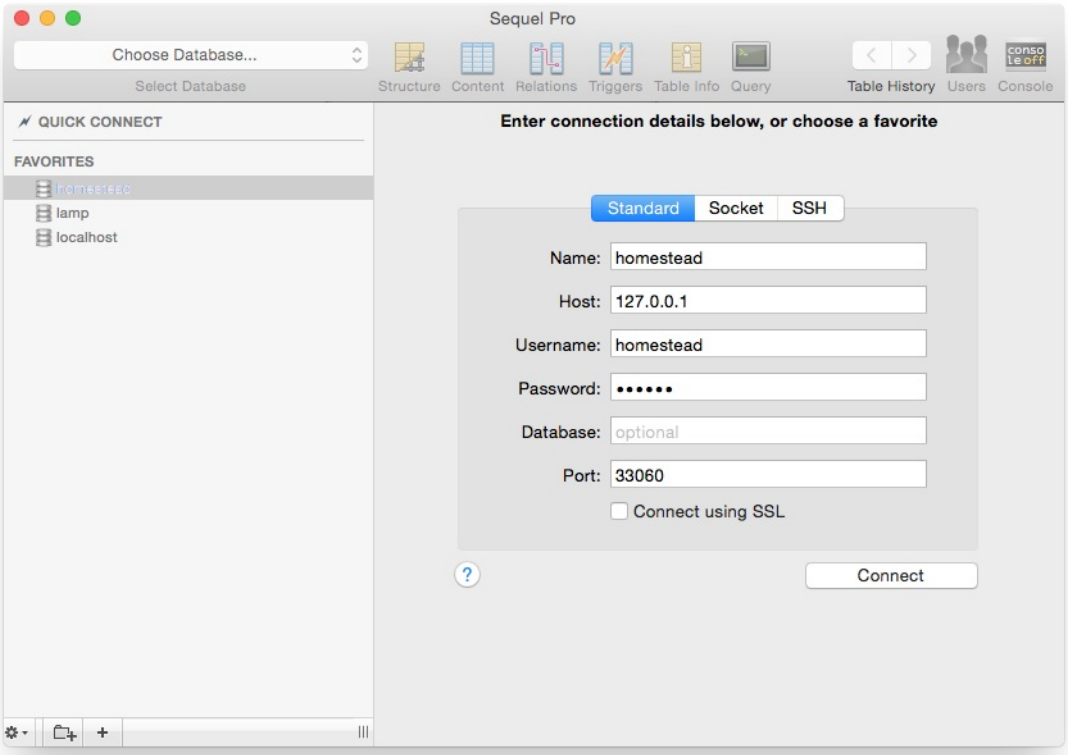
Even though we can easily create a new database via the **CLI**, we should use a **Graphical User Interface (GUI)** to manage databases easily.

## 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.




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, right?

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
```

You should see all commands:

```
Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
      --ansi            Force ANSI output
      --no-ansi         Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
      --env[=ENV]       The environment the command should run under
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output, 2 for
more verbose output and 3 for debug

Available commands:
  clear-compiled       Remove the compiled class file
  down                 Put the application into maintenance mode
  env                  Display the current framework environment
  help                 Displays help for a command
  inspire              Display an inspiring quote
  list                 Lists commands
  migrate              Run the database migrations
  optimize             Optimize the framework for better performance
  serve                Serve the application on the PHP development server
  tinker               Interact with your application
  up                   Bring the application out of maintenance mode
 app
  app:name             Set the application namespace
 auth
  auth:clear-resets    Flush expired password reset tokens
 cache
  cache:clear          Flush the application cache
  cache:forget         Remove an item from the cache
  cache:table          Create a migration for the cache database table
 config
  config:cache         Create a cache file for faster configuration loading
  config:clear         Remove the configuration cache file
 db
```

```
  db:seed               Seed the database with records
 event
  event:generate        Generate the missing events and listeners based on registration
 key
  key:generate          Set the application key
 make
  make:auth             Scaffold basic login and registration views and routes
  make:command          Create a new Artisan command
  make:controller       Create a new controller class
  make:event            Create a new event class
  make:job              Create a new job class
  make:listener         Create a new event listener class
  make:mail             Create a new email class
  make:middleware       Create a new middleware class
  make:migration        Create a new migration file
  make:model            Create a new Eloquent model class
  make:notification     Create a new notification class
  make:policy           Create a new policy class
  make:provider         Create a new service provider class
  make:request          Create a new form request class
  make:seeder           Create a new seeder class
  make:test             Create a new test class
 migrate
  migrate:install       Create the migration repository
  migrate:refresh       Reset and re-run all migrations
  migrate:reset         Rollback all database migrations
  migrate:rollback      Rollback the last database migration
  migrate:status        Show the status of each migration
 notifications
  notifications:table   Create a migration for the notifications table
 queue
  queue:failed          List all of the failed queue jobs
  queue:failed-table    Create a migration for the failed queue jobs database table
  queue:flush           Flush all of the failed queue jobs
  queue:forget          Delete a failed queue job
  queue:listen          Listen to a given queue
  queue:restart         Restart queue worker daemons after their current job
  queue:retry           Retry a failed queue job
  queue:table           Create a migration for the queue jobs database table
  queue:work            Start processing jobs on the queue as a daemon
 route
  route:cache           Create a route cache file for faster route registration
  route:clear           Remove the route cache file
  route:list            List all registered routes
 schedule
  schedule:run          Run the scheduled commands
 session
  session:table         Create a migration for the session database table
 storage
  storage:link          Create a symbolic link from "public/storage" to "storage/app/pu
blic"
 vendor
  vendor:publish        Publish any publishable assets from vendor packages
```

```
 view
  view:clear           Clear all compiled view files
```

## 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: 2017_09_09_095351_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:

**2018_10_01_180019_create_tickets_table**

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

use Illuminate\Support\Facades\Schema;
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->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('tickets');
    }
}
```

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.

## Errors when creating or deleting a migration

Before reading the next section, let's take a look at some common errors when creating or deleting a migration.

If you see this error:

```
      [ErrorException]
      include(/home/vagrant/Code/Laravel/database/migrations/2016_09_09_193947_create_
tickets_table.php):
      failed to open stream: No such file or directory
```

You will need to run this command to regenerate the list of all classes that need to be included in your app:

```
composer dump-autoload -o
```

> **Tip:** In the future, if you see this error again, you can run the command above to fix it.

If you see this error:

```
[Illuminate\Database\QueryException] SQLSTATE[42000]: Syntax error or access violation:
 1071 Specified key was too long;
```

You may be running a version of MySQL older than the **5.7.7** release. To fix the bug, you need to edit the **AppServiceProvider.php** file, and set a default string length inside the `boot` method:

```
use Illuminate\Support\Facades\Schema;

public function boot()
{
    Schema::defaultStringLength(191);
}
```

> **Note:** You can read more about this issue at https://laravel.com/docs/master/migrations#creating-indexes

## Understand Schema to write migrations

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

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**:

https://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:

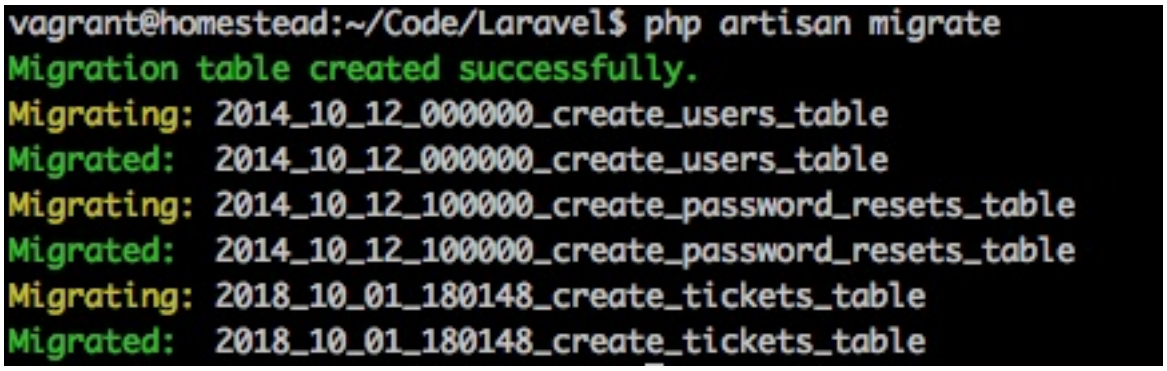
**create_tickets_table.php**

```
<?php

use Illuminate\Support\Facades\Schema;
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')->nullable();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('tickets');
    }
}
```

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

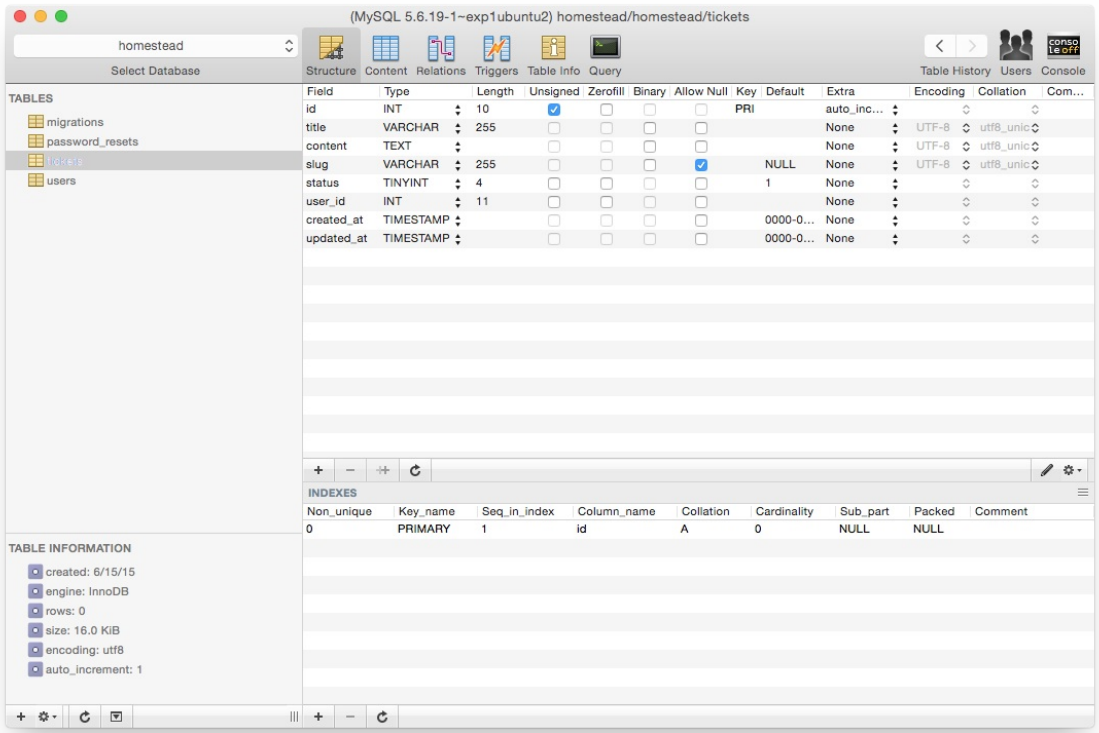
```
php artisan migrate
```




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_table** 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:




> **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:

**app/Ticket.php**

```
<?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**

**views/tickets/create.blade.php**

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

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Create a ticket</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                <form>
                    <fieldset>
                        <div class="form-group">
                            <label for="title" class="col-lg-12 control-label">Title</
label>
                            <div class="col-lg-12">
                                <input type="text" class="form-control" id="title" pla
ceholder="Title">
                            </div>
                        </div>
                        <div class="form-group">
                            <label for="content" class="col-lg-12 control-label">Conte
nt</label>
                            <div class="col-lg-12">
                                <textarea class="form-control" rows="3" id="content"></
textarea>
                                <span class="help-block">Feel free to ask us any quest
ion.</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>
    </div>
@endsection
```

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

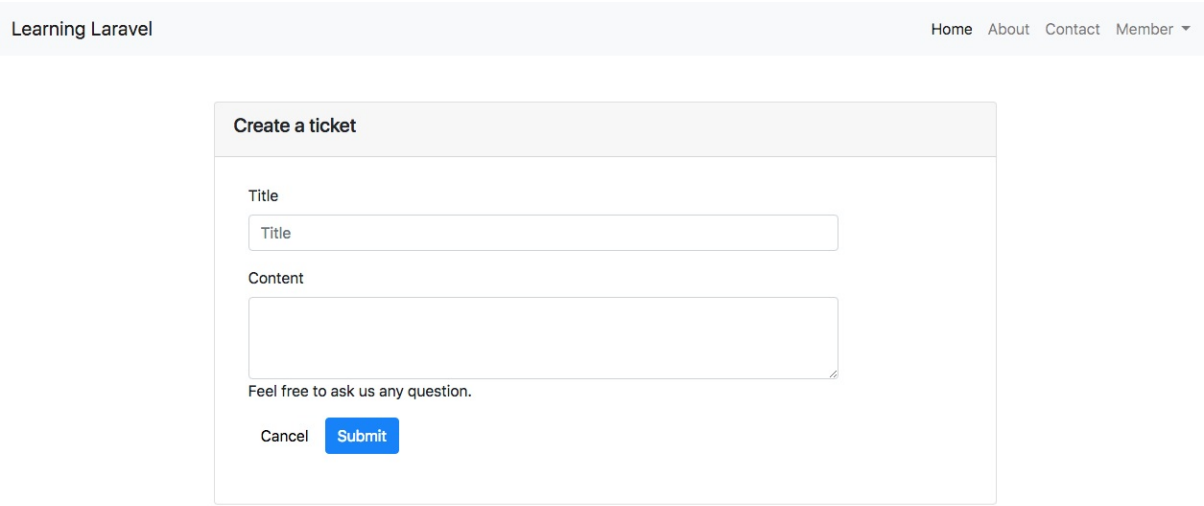
```
public function contact()
{
    return view('contact');
}
```

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:




# 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 --resource
```

As mentioned before, by using the **--resource** flag, Laravel creates some **RESTful actions** (create, edit, update, etc.) for us.

Update the **create** action as follows:

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

And don't forget to update the **web.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
```

```
vagrant@homestead:~/Code/Laravel$ php artisan make:request TicketFormRequest
Request created successfully.
```

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

> **Note:** This is an optional section, you may **SKIP THIS SECTION**. The package might not support the latest version of Laravel yet.

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!

https://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.

**composer.json**

```
{
    "name": "laravel/laravel",
    "description": "The Laravel Framework.",
    "keywords": ["framework", "laravel"],
    "license": "MIT",
    "type": "project",
    "require": {
        "php": ">=7.0.0",
        "fideloper/proxy": "~3.3",
        "laravel/framework": "5.5.*",
        "laravel/tinker": "~1.0"
    },
    "require-dev": {
        "filp/whoops": "~2.0",
        "fzaninotto/faker": "~1.4",
        "mockery/mockery": "0.9.*",
        "phpunit/phpunit": "~6.0"
    },
    "autoload": {
        "classmap": [
            "database/seeds",
            "database/factories"
        ],
        "psr-4": {
            "App\\": "app/"
        }
    },
    "autoload-dev": {
        "psr-4": {
            "Tests\\": "tests/"
        }
```

```
    },
    "extra": {
        "laravel": {
            "dont-discover": [
            ]
        }
    },
    "scripts": {
        "post-root-package-install": [
            "@php -r \"file_exists('.env') || copy('.env.example', '.env');\""
        ],
        "post-create-project-cmd": [
            "@php artisan key:generate"
        ],
        "post-autoload-dump": [
            "Illuminate\\Foundation\\ComposerScripts::postAutoloadDump",
            "@php artisan package:discover"
        ]
    },
    "config": {
        "preferred-install": "dist",
        "sort-packages": true,
        "optimize-autoloader": 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:

https://w3schools.com/json

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

https://laravelcollective.com/docs/master/html

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

find:

```
"require": {
    "php": ">=5.6.4",
    "laravel/framework": "5.5.*",
    "laravel/tinker": "~1.0"
```

```
},
```

add:

```
"require": {
    "php": ">=5.6.4",
    "laravel/framework": "5.5.*",
    "laravel/tinker": "~1.0",
    "laravelcollective/html": "5.5.*"
},
```

> **Note:** Your version could be different. For example, if you're using **Laravel 5.6**, the code should be "laravelcollective/html": "5.6.*". Check for the latest version here.

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 provider** to the **providers** array of the **config/app.php**.

Find:

```
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
    Illuminate\Bus\BusServiceProvider::class,
    Illuminate\Cache\CacheServiceProvider::class,
    Illuminate\Foundation\Providers\ConsoleSupportServiceProvider::class,
    Illuminate\Cookie\CookieServiceProvider::class,
    Illuminate\Database\DatabaseServiceProvider::class,
    Illuminate\Encryption\EncryptionServiceProvider::class,
```

```
        Illuminate\Filesystem\FilesystemServiceProvider::class,
        Illuminate\Foundation\Providers\FoundationServiceProvider::class,
        Illuminate\Hashing\HashServiceProvider::class,
        Illuminate\Mail\MailServiceProvider::class,
        Illuminate\Notifications\NotificationServiceProvider::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,

        /*
         * Package Service Providers...
         */
        Laravel\Tinker\TinkerServiceProvider::class,

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

    ],
```

Add the following line to the **$provider** array:

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

Your **$provider** array should look like this:

```
'providers' => [

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

```
        Illuminate\Encryption\EncryptionServiceProvider::class,
        Illuminate\Filesystem\FilesystemServiceProvider::class,
        Illuminate\Foundation\Providers\FoundationServiceProvider::class,
        Illuminate\Hashing\HashServiceProvider::class,
        Illuminate\Mail\MailServiceProvider::class,
        Illuminate\Notifications\NotificationServiceProvider::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,

        /*
         * Package Service Providers...
         */
        Laravel\Tinker\TinkerServiceProvider::class,
        Collective\Html\HtmlServiceProvider::class,

        /*
         * Application Service Providers...
         */
        App\Providers\AppServiceProvider::class,
        App\Providers\AuthServiceProvider::class,
        // App\Providers\BroadcastServiceProvider::class,
        App\Providers\EventServiceProvider::class,
        App\Providers\RouteServiceProvider::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,
        'Broadcast' => Illuminate\Support\Facades\Broadcast::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,
```

```
    'Gate' => Illuminate\Support\Facades\Gate::class,
    'Hash' => Illuminate\Support\Facades\Hash::class,
    'Lang' => Illuminate\Support\Facades\Lang::class,
    'Log' => Illuminate\Support\Facades\Log::class,
    'Mail' => Illuminate\Support\Facades\Mail::class,
    'Notification' => Illuminate\Support\Facades\Notification::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:

```
'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,
    'Broadcast' => Illuminate\Support\Facades\Broadcast::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,
    'Gate' => Illuminate\Support\Facades\Gate::class,
    'Hash' => Illuminate\Support\Facades\Hash::class,
    'Lang' => Illuminate\Support\Facades\Lang::class,
    'Log' => Illuminate\Support\Facades\Log::class,
```

```
    'Mail' => Illuminate\Support\Facades\Mail::class,
    'Notification' => Illuminate\Support\Facades\Notification::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,
    '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 some Laravel developers are still using it these days. It would be better if you know both methods.

# 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 **web.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 **Illuminate\Http\Request** 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>
```

Update to:

```
<form method="post">
```

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

Find:

```
<input type="text" class="form-control" id="title" placeholder="Title">
```

Update to:

```
<input type="text" class="form-control" id="title" placeholder="Title" name="title">
```

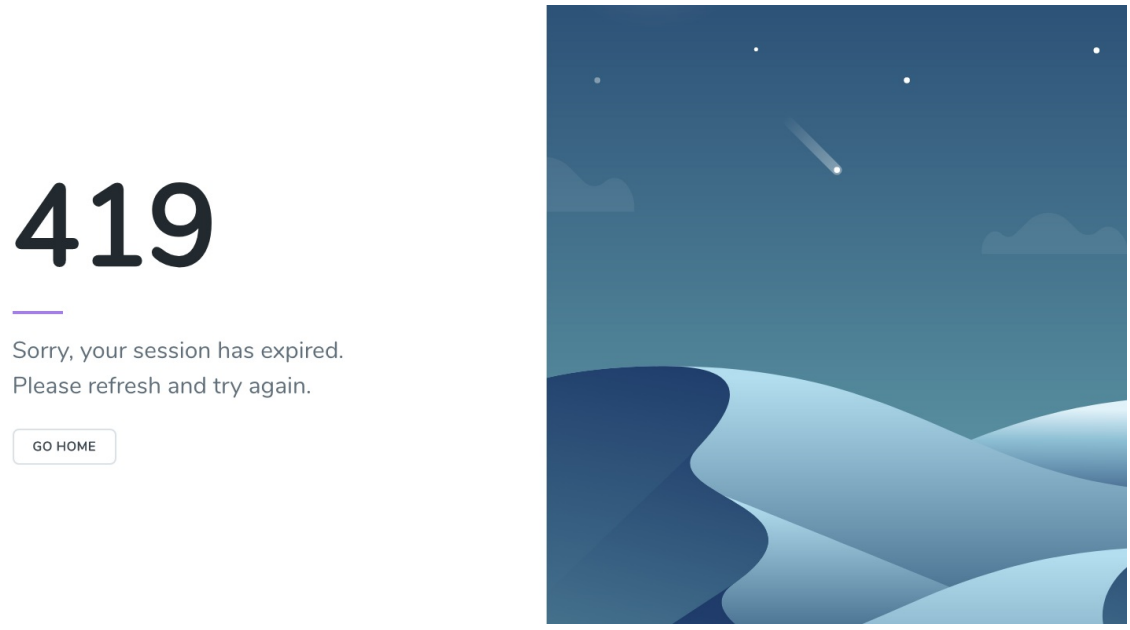
Find:

```
<textarea class="form-control" rows="3" id="content"></textarea>
```

Update to:

```
<textarea class="form-control" rows="3" id="content" name="content"></textarea>
```

Now, try to click the submit button!



Oops! There is an error - known as **TokenMismatchException**.

What is it?

For security purposes, Laravel 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 method="post">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
```

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


```
{
    "_token": "I5plO4YxBShS3UBl46AgWtrDZUPxlDimtTrBF8L5",
    "title": "My first ticket",
    "content": "This is a test"
}
```


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

> **Note:** If the page is refreshed and you don't see the ticket data, that means the validation rules are working. You have to follow the rules and fill the title and the content correctly.

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

Find:

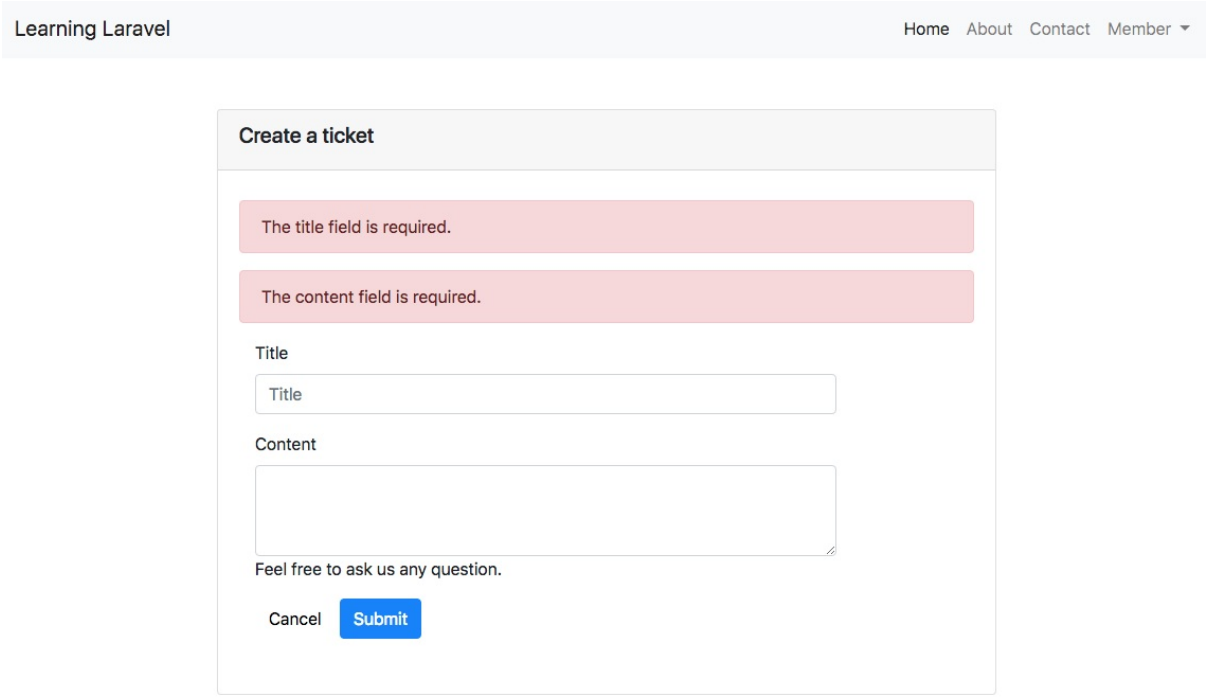
```
<form method="post">
```

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:




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

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

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Create a ticket</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                <form 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>
                        <div class="form-group">
```

```
                            <label for="title" class="col-lg-12 control-label">Title</
label>
                            <div class="col-lg-12">
                                <input type="text" class="form-control" id="title" pla
ceholder="Title" name="title">
                            </div>
                        </div>
                        <div class="form-group">
                            <label for="content" class="col-lg-12 control-label">Conte
nt</label>
                            <div class="col-lg-12">
                                <textarea class="form-control" rows="3" id="content" n
ame="content"></textarea>
                                <span class="help-block">Feel free to ask us any quest
ion.</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>
    </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_NAME=Laravel
APP_ENV=local
APP_KEY=base64:7mE0BuvaO41HhPKztH+ZGhXval8SMbABs7wEwI7/w10=
APP_DEBUG=true
APP_URL=http://localhost

LOG_CHANNEL=stack

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

BROADCAST_DRIVER=log
CACHE_DRIVER=file
QUEUE_CONNECTION=sync
SESSION_DRIVER=file
SESSION_LIFETIME=120

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

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

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
PUSHER_APP_CLUSTER=mt1

MIX_PUSHER_APP_KEY="${PUSHER_APP_KEY}"
MIX_PUSHER_APP_CLUSTER="${PUSHER_APP_CLUSTER}"
```

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 Sendinblue to send emails, replace these lines with your **Sendinblue 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-relay.sendinblue.com
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 as follows:

```
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! Its uni
que 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.

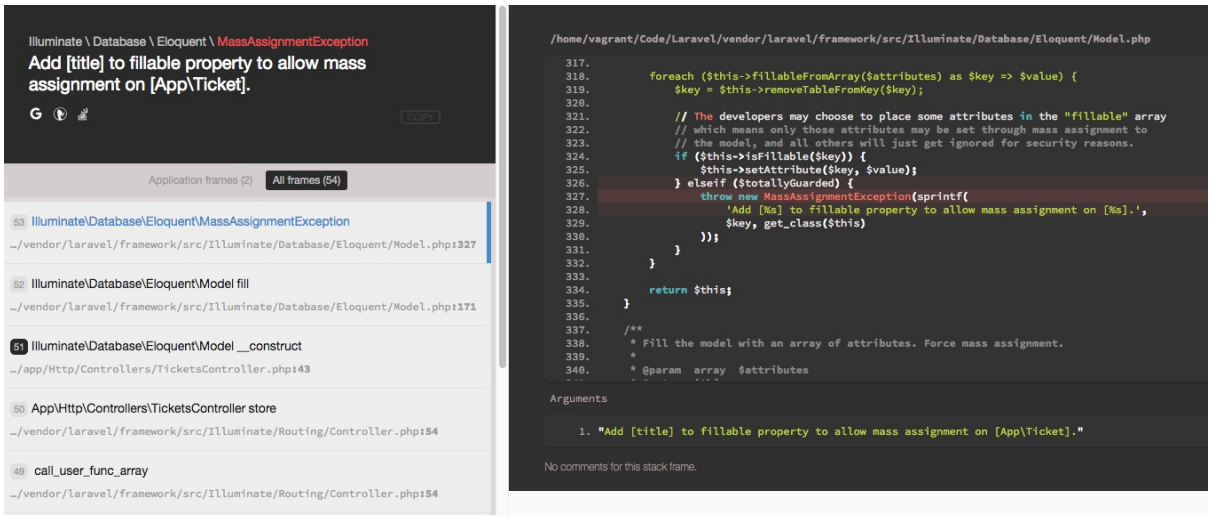
```
$ticket->save();
```

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

```
return redirect('/contact')->with('status', 'Your ticket has been created! Its unique
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.




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:

https://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:

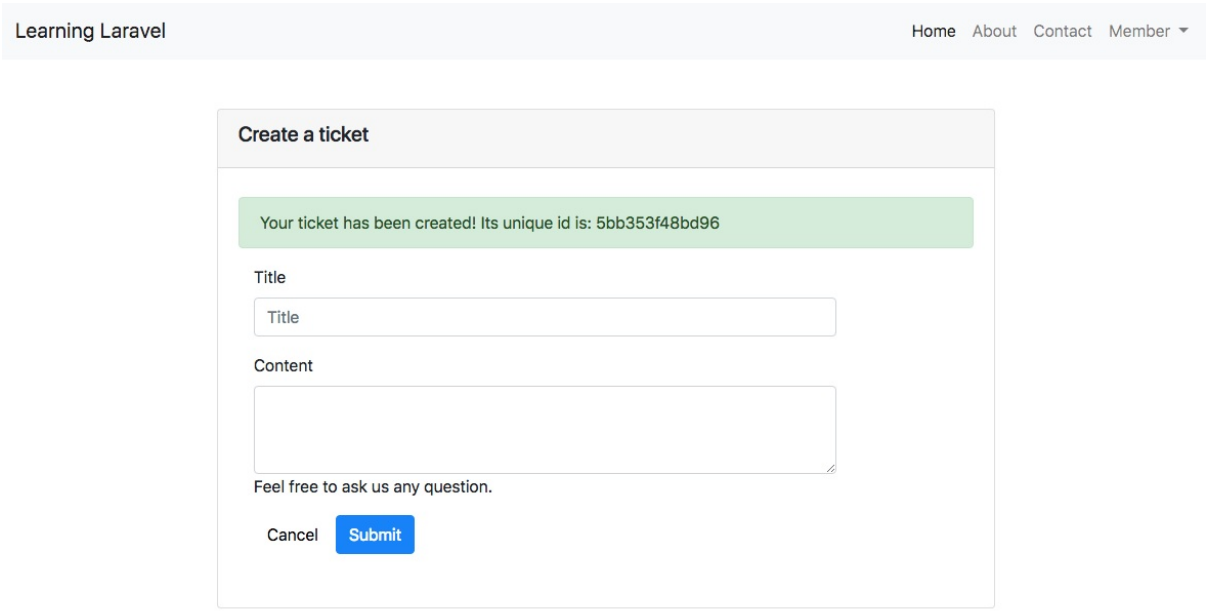
Find:

```
<form method="post">

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

Add Below:

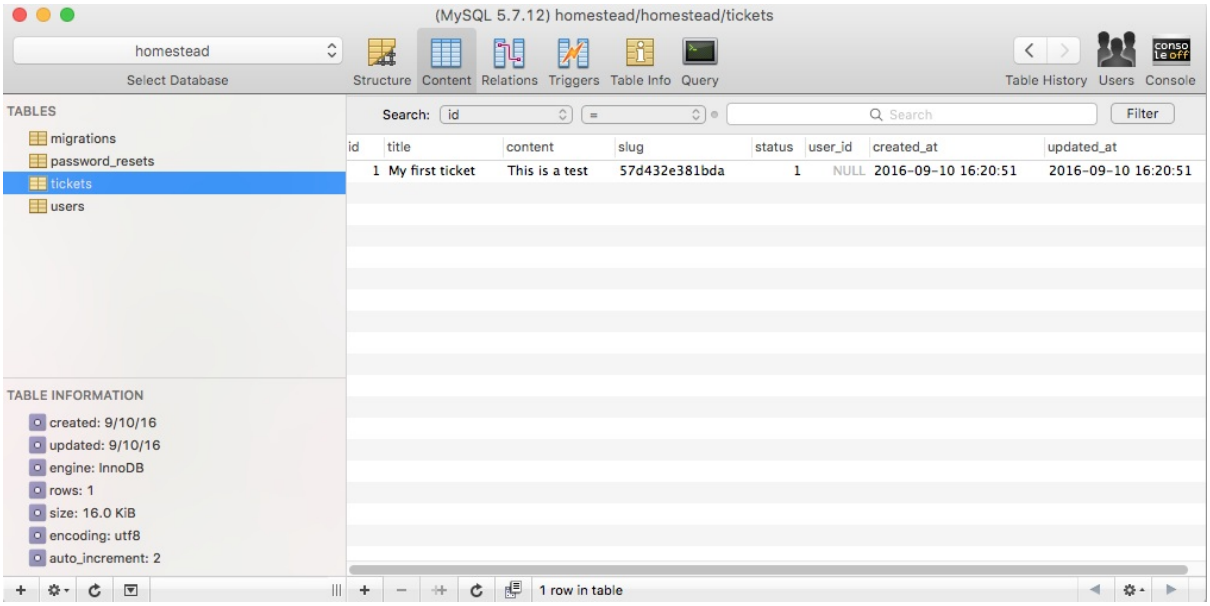
```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

Good! Try to create a new ticket again.




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:



# 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 **web.php** file:

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

When users access **homestead.test/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, and 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:

**views/tickets/index.blade.php**

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

    <div class="container col-md-8 col-md-offset-2 mt-5">
        <div class="card">
            <div class="card-header ">
                <h5 class="float-left">Tickets</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                @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' : 'Answered' }}</td>

                            </tr>
                        @endforeach
                        </tbody>
                    </table>
                @endif
            </div>
        </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
```

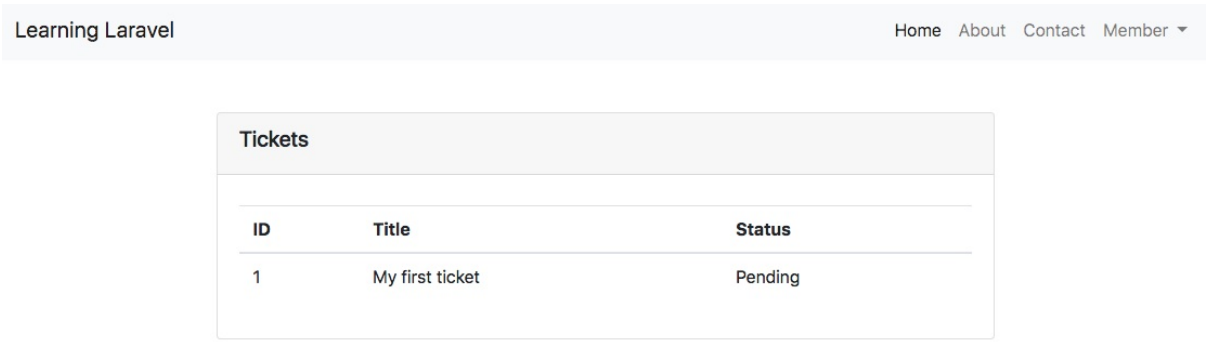
If the **$tickets** is not empty, we use **foreach() loop** to display all tickets.

```
<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.test/tickets**, you should be able to view all the tickets that you've created!




# 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 **web.php** file, and 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:

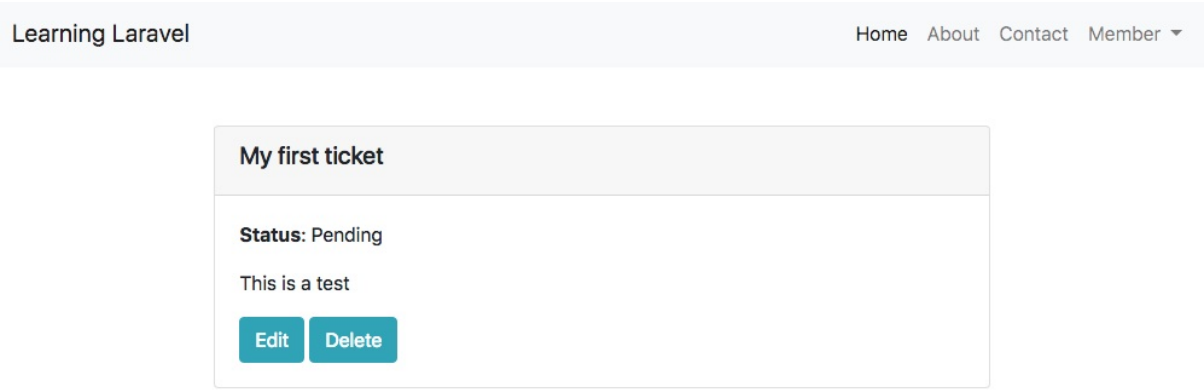
**views/tickets/show.blade.php**

```
@extends('master')
@section('title', 'View a ticket')
@section('content')

    <div class="container col-md-8 col-md-offset-2 mt-5">
        <div class="card">
            <div class="card-header ">
                <h5 class="float-left">{{ $ticket->title }}</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body">
                <p> <strong>Status</strong>: {{ $ticket->status ? 'Pending' : 'Answere
d' }}</p>
                <p> {{ $ticket->content }} </p>
                <a href="#" class="btn btn-info">Edit</a>
                <a href="#" class="btn btn-info">Delete</a>
            </div>
        </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.test/ticket/yourSlug, you'll see:




> **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:

https://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:

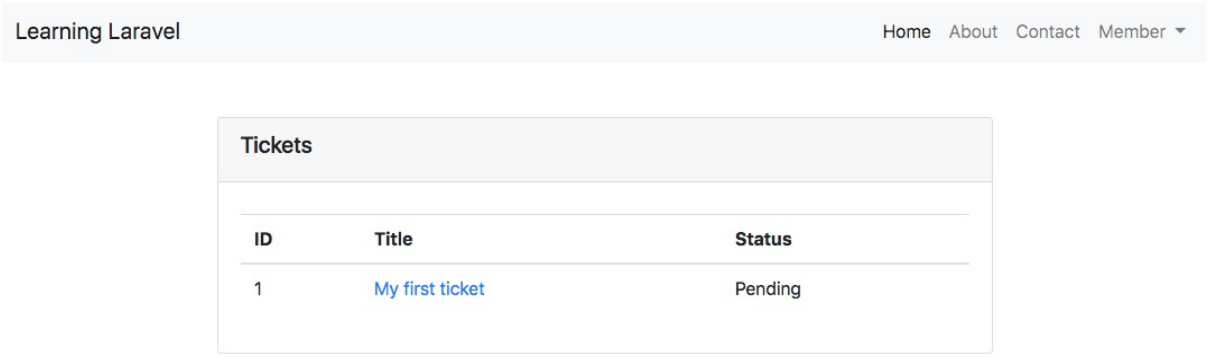
```
action('TicketsController@show', $ticket->slug)
```

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.test/tickets**, you can click on the title to view the ticket.




# Edit a ticket

It's time to move on to create our ticket edit form.

Open **web.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="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Edit ticket</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                <form 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>
                        <div class="form-group">
                            <label for="title" class="col-lg-12 control-label">Title</
label>
                            <div class="col-lg-12">
                                <input type="text" class="form-control" id="title" pla
ceholder="Title" name="title" value="{{ $ticket->title }}">
                            </div>
```

```
                        </div>
                        <div class="form-group">
                            <label for="content" class="col-lg-12 control-label">Conte
nt</label>
                            <div class="col-lg-12">
                                <textarea class="form-control" rows="3" id="content" n
ame="content">{{ $ticket->content }}</textarea>
                                <span class="help-block">Feel free to ask us any quest
ion.</span>
                            </div>
                        </div>
                        <div class="form-group">
                            <label>
                                <input type="checkbox" name="status" {{ $ticket->status
?"":"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>
    </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:

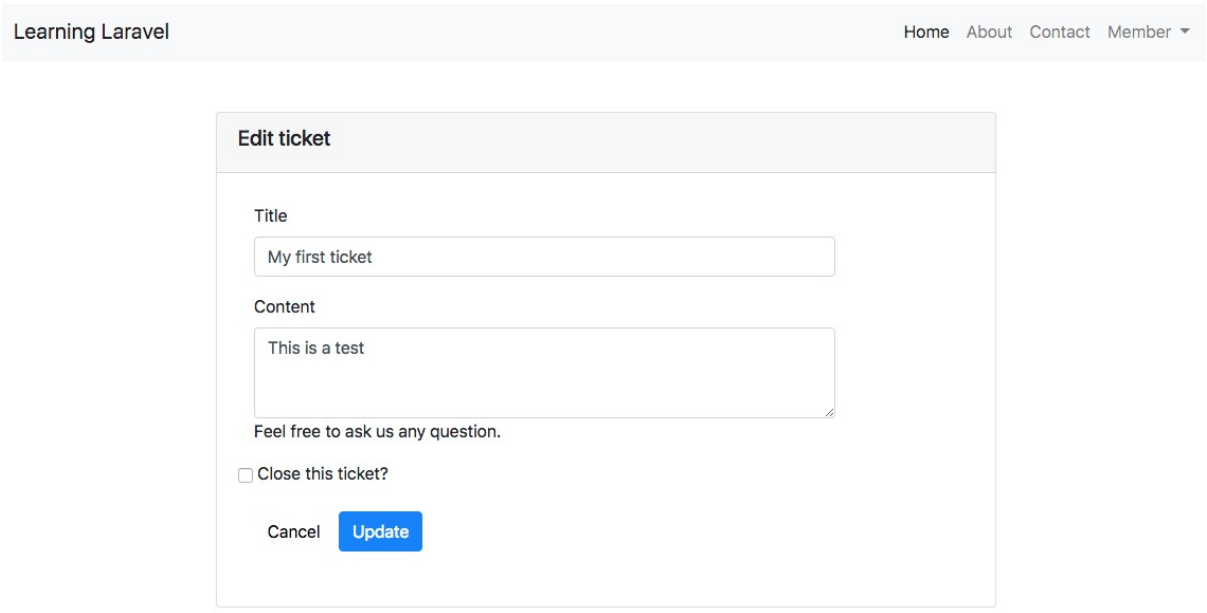
Find:

```
<a href="#" class="btn btn-info">Edit</a>
```

Update to:

```
<a href="{{ action('TicketsController@edit', $ticket->slug) }}" class="btn btn-info">E
dit</a>
```

We use the **action helper** again! When you click on the edit button, you should be able to access the 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 **web.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', $ticket->slug))->with('status', '
The ticket '.$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();
```

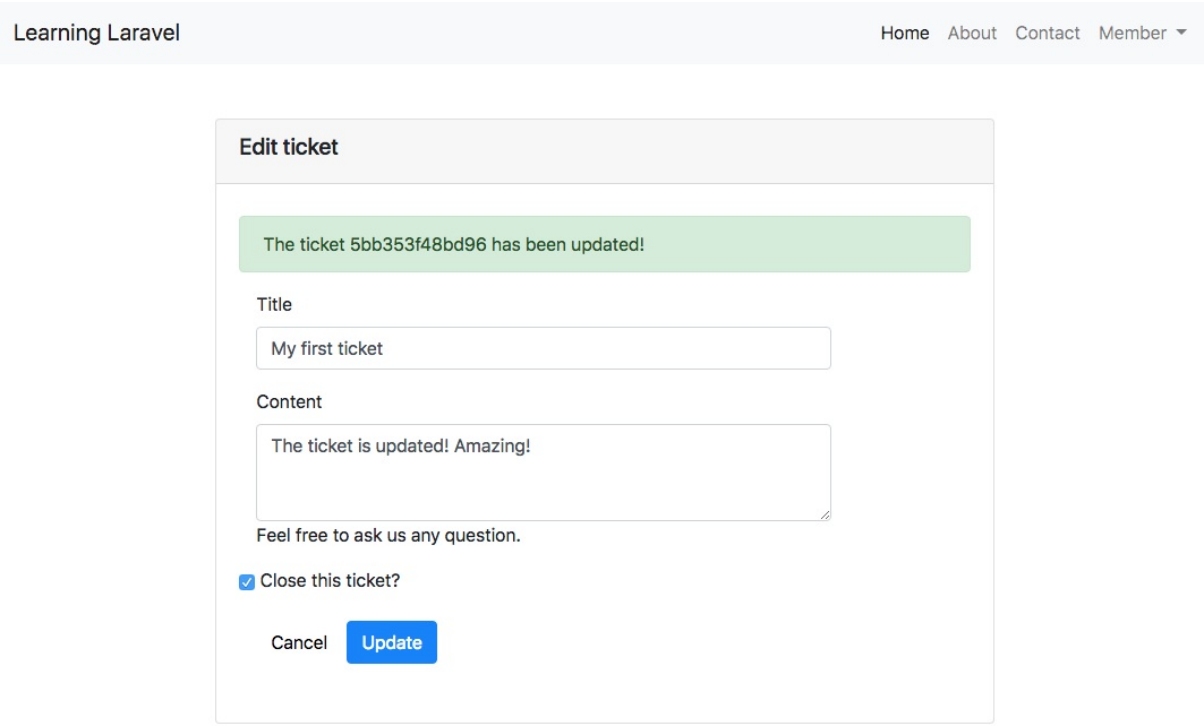
This is how we can check if the users select the status checkbox or not:

```
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**!




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 **web.php** file and add a new route:

```
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 deleted
!');
```

```
}
```

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="card-body mt-2">
```

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.

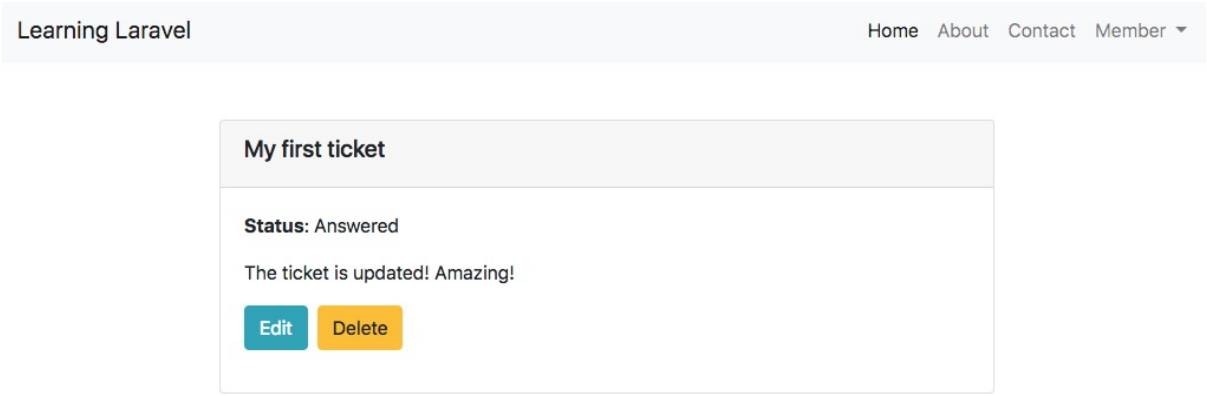
Open **show.blade.php** and find:

```
<a href="{{ action('TicketsController@edit', $ticket->slug) }}" class="btn btn-info">E
dit</a>
<a href="#" class="btn btn-info">Delete</a>
```

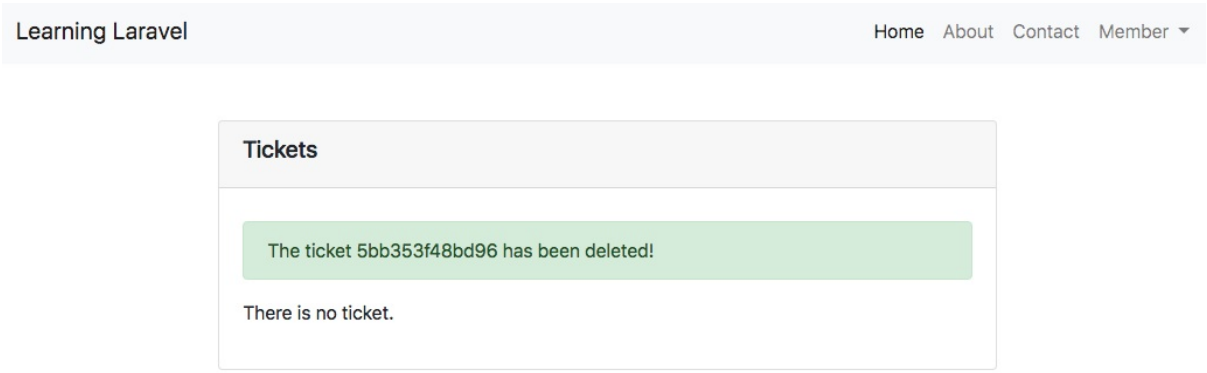
Update to:

```
<a href="{{ action('TicketsController@edit', $ticket->slug) }}" class="btn btn-info fl
oat-left mr-2">Edit</a>
<form method="post" action="{{ action('TicketsController@destroy', $ticket->slug) }}"
class="float-left">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
    <div>
        <button type="submit" class="btn btn-warning">Delete</button>
    </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:




Now, click the **delete button**, you should be able to remove the ticket!




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, Sendinblue, 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' => env('MAIL_FROM_ADDRESS', 'hello@example.com'),
    'name' => env('MAIL_FROM_NAME', 'Example'),
     ],

    'encryption' => env('MAIL_ENCRYPTION', 'tls'),

    'username' => env('MAIL_USERNAME'),

    'password' => env('MAIL_PASSWORD'),

    'sendmail' => '/usr/sbin/sendmail -bs',

    'markdown' => [
    'theme' => 'default',

    'paths' => [
        resource_path('views/vendor/mail'),
    ],
],
];
```

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:

https://laravel.com/docs/master/mail

Because working on Homestead, we will learn how to send emails on Homestead using **Gmail** and Sendinblue for FREE!

## Sending emails using Gmail

> **Note:** Even though we can use Gmail, it's recommended to use a transactional email service (Sendinblue, Mandrill, etc.) to send emails. You may **SKIP THIS SECTION**.

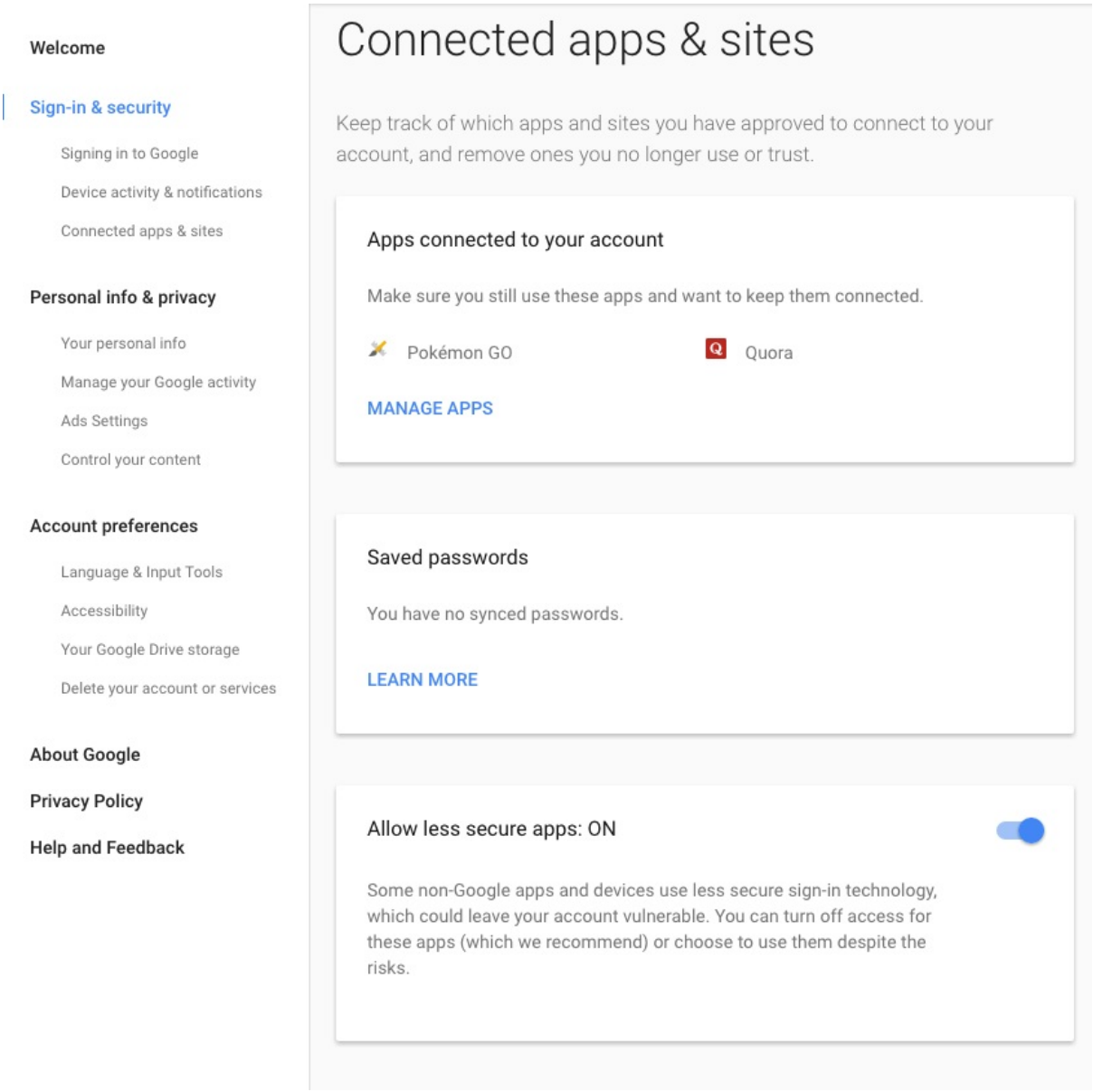
If you have a Gmail account, it's very easy to send emails using Laravel 5!

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**.




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 Sendinblue or other mail service providers

Go to **Sendinblue**, register a new account:

https://www.sendinblue.com/?ae=484

> **Note:** If you don't want to use Sendinblue, you can try other email service providers, such as Sendgrid.

When your account is activated, edit the **.env** file:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp-relay.sendinblue.com
MAIL_PORT=587
MAIL_USERNAME=yourSendinblueUsername
MAIL_PASSWORD=yourPassword
```

Good job! You're now ready to send emails using Sendinblue!

## Sending a test email

To send a test email, open **web.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 email');

    });
```

```
    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.test/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.

**views/emails/welcome.blade.php**

```
<!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.test/sendemail, you should see:

```
Your email has been sent 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 unique
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!');
});
```

> **Note:** Be sure to replace **yourEmail@domain.com** with your **real email address**. Don't add the code above if you don't want to send an email.

And don't forget to tell Laravel that you want to use the **Mail** facade here:

Find:

```
class TicketsController extends Controller
```

Add above:

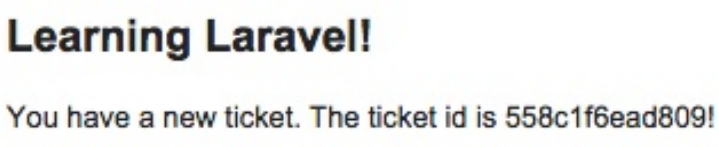
```
use Illuminate\Support\Facades\Mail;
```

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:




Laravel 5.6+ allows us to send emails using Markdown. If you love Markdown, you may read the documentation to learn how to use this feature:

https://laravel.com/docs/master/mail#markdown-mailables

# 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
```

Then, open **yourTimestamps_create_comments_table.php**, use **Schema** to create some columns:

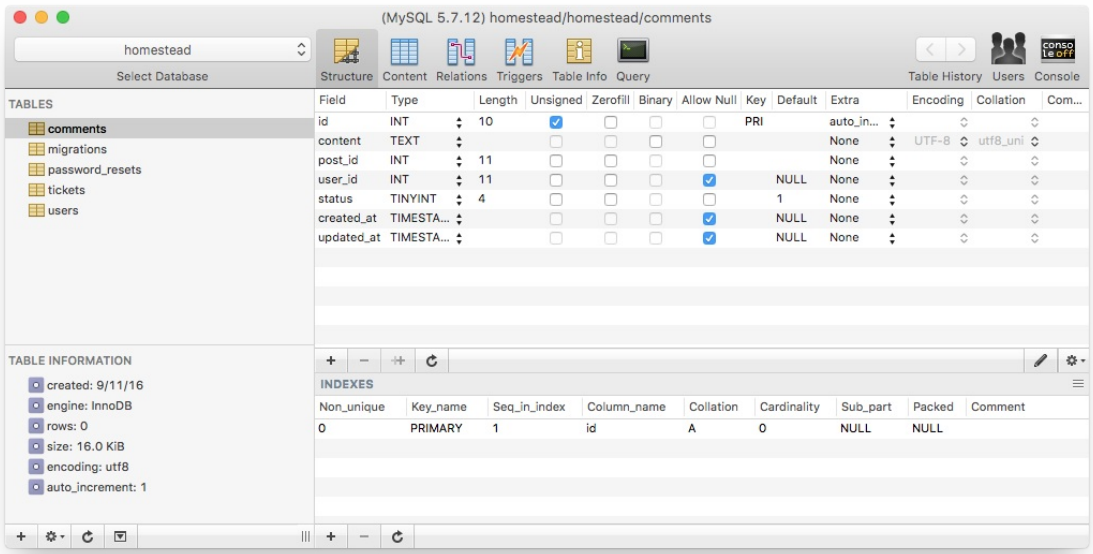
```
<?php

use Illuminate\Support\Facades\Schema;
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')->nullable();
            $table->tinyInteger('status')->default(1);
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('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.




## 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:

https://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:

**app/Comment.php**

```
<?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');
}
```

You now have:

**app/Ticket.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->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 **web.php** first:

```
Route::post('/comment', 'CommentsController@newComment');
```

When we send a **POST** request to this route, Laravel will execute the **CommentsController's newComment** action.

It's time to run this command to generate our controller:

```
php artisan make:controller CommentsController
```

Open the new **CommentsController** and add a new action:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
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!');
    }
}
```

Don't forget to add:

```
use App\Http\Requests\CommentFormRequest;
use App\Comment;
```

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
```

```
vagrant@homestead:~/Code/Laravel$ php artisan make:migration create_comments_table --create=comments
Created Migration: 2015_06_26_145222_create_comments_table
vagrant@homestead:~/Code/Laravel$ php artisan migrate
Migrated: 2015_06_26_145222_create_comments_table
vagrant@homestead:~/Code/Laravel$ php artisan make:model Comment
Model created successfully.
vagrant@homestead:~/Code/Laravel$ php artisan make:controller CommentsController
Controller created successfully.
vagrant@homestead:~/Code/Laravel$ php artisan make:request CommentFormRequest
Request created successfully.
vagrant@homestead:~/Code/Laravel$
```

Now, define our rules:

**app/Requests/CommentFormRequest.php**

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class CommentFormRequest extends FormRequest
{
    /**
     * 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',
        ];
    }
}
```

> **Note:** Check the **authorize()** function, don't forget to update it to **return true**

## Create a new reply form

Good job! Now open the **tickets.show** view and find:

```
        </form>
        <div class="clearfix"></div>
    </div>
</div>
```

Add this form below:

```
<div class="card mt-3">
    <form 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 class="ml-3">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>
```

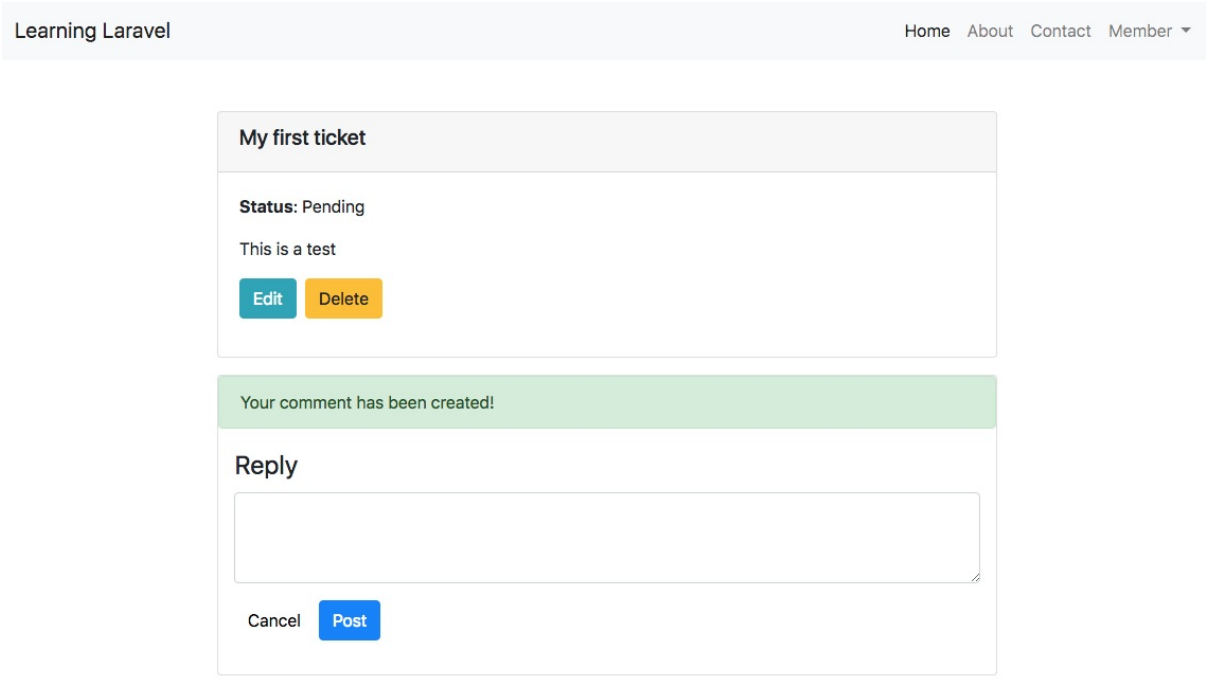
Here is the new **tickets.show** view:

```
@extends('master')
@section('title', 'View a ticket')
@section('content')

    <div class="container col-md-8 col-md-offset-2 mt-5">
        <div class="card">
            <div class="card-header ">
                <h5 class="float-left">{{ $ticket->title }}</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body">
                <p> <strong>Status</strong>: {{ $ticket->status ? 'Pending' : 'Answere
d' }}</p>
                <p> {{ $ticket->content }} </p>
                <a href="{{ action('TicketsController@edit', $ticket->slug) }}" class="
btn btn-info float-left mr-2">Edit</a>
                <form method="post" action="{{ action('TicketsController@destroy', $ti
cket->slug) }}" class="float-left">
                    <input type="hidden" name="_token" value="{{ csrf_token() }}">
                    <div>
                        <button type="submit" class="btn btn-warning">Delete</button>
                    </div>
                </form>
                <div class="clearfix"></div>
            </div>
        </div>
        <div class="card mt-3">
            <form 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 class="ml-3">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>

    </div>

@endsection
```

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.




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, and add this code above the reply form:

```
@foreach($comments as $comment)
```

```
    <div class="card mt-3">
        <div class="card-body">
            {{ $comment->content }}
        </div>
    </div>
@endforeach
```

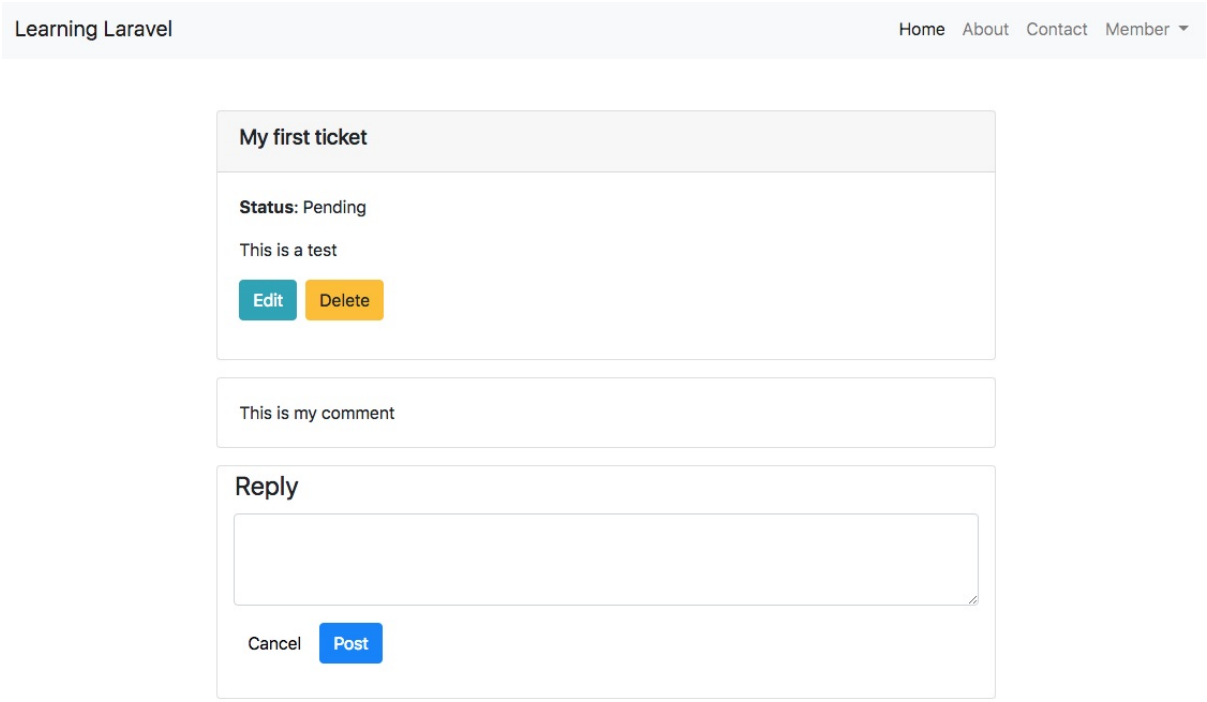
Here is the new **show.blade.php**:

**views/tickets/show.blade.php**

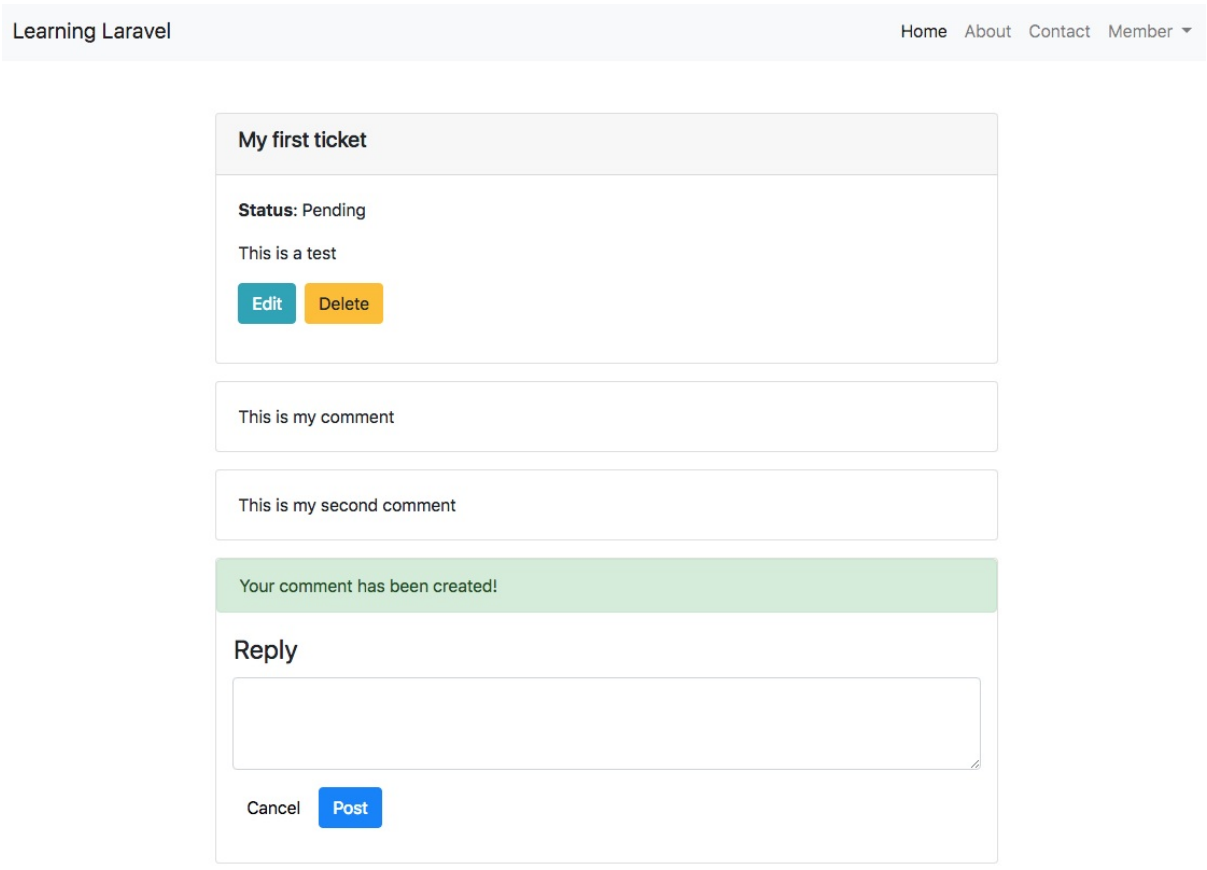
```
@extends('master')
@section('title', 'View a ticket')
@section('content')

    <div class="container col-md-8 col-md-offset-2 mt-5">
        <div class="card">
            <div class="card-header ">
                <h5 class="float-left">{{ $ticket->title }}</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body">
                <p> <strong>Status</strong>: {{ $ticket->status ? 'Pending' : 'Answere
d' }}</p>
                <p> {{ $ticket->content }} </p>
                <a href="{{ action('TicketsController@edit', $ticket->slug) }}" class="
btn btn-info float-left mr-2">Edit</a>
                <form method="post" action="{{ action('TicketsController@destroy', $ti
cket->slug) }}" class="float-left">
                    <input type="hidden" name="_token" value="{{ csrf_token() }}">
                    <div>
                        <button type="submit" class="btn btn-warning">Delete</button>
                    </div>
                </form>
                <div class="clearfix"></div>
            </div>
        </div>
        @foreach($comments as $comment)
            <div class="card mt-3">
                <div class="card-body">
                    {{ $comment->content }}
                </div>
            </div>
        @endforeach

        <div class="card mt-3">
            <form 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 class="ml-3">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>

    </div>

@endsection
```

Refresh your browser now!




To make sure that everything is working, reply again!




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 Sendinblue 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 3 Source Code

You can view and download the source code at:

> Learning Laravel 5 Book: Chapter 3 Source Code

# 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 of 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.

By default, Laravel has created some **authentication controllers** for us:

- **RegisterController**: handles new user registration.
- **LoginController**: handles authentication (login).
- **ForgotPasswordController**: sends emails to users with a link to reset their password.
- **ResetPasswordController**: reset users' password.

You can find them in the **app/Http/Controllers/Auth** directory.

> **Note:** If you don't see these controllers, you may use a different version of Laravel. In **Laravel 5.1** and **5.2**, we only have one controller (the **AuthController**) that handles all the functionalities.

First, let's take a look at our database, we should see that the **users table** has been created.

As mentioned before, Laravel comes with a **default user migration**, which is placed in the **database/migrations** directory:

```
<?php

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

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
```

```
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('users');
    }
}
```

If you want to **customize** your **users table**, you can modify this migration.

We also have the **User model** (app/User.php) already.

Since Laravel 5.2, we can generate all routes and views for our authentication system using **Laravel Auth Scaffold**. We just need to run this command:

```
php artisan make:auth
```

However, for learning purposes, we will create our routes and views manually to understand how the system works.

Once you know how the system works, you can easily customize it to meet your needs.

Next, we would need some routes for our registration form. We can open **web.php** and add these routes:

```
Route::get('users/register', 'Auth\RegisterController@showRegistrationForm');
Route::post('users/register', 'Auth\RegisterController@register');
```

The first route will provide the **registration form**. The second route will **process the form**. Both routes are handled by the **RegisterController**'s actions: **showRegistrationForm** and **register**.

Let's open the **RegisterController** 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|min:6|confirmed',
    ]);
}
```

As you may notice, there are **three fields** here: **name**, **email** and **password**.

When users visit **users/register**, this RegisterController 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:

**resources/views/auth/register.blade.php**

```
@extends('master')
@section('name', 'Register')

@section('content')
    <div class="container col-md-6 col-md-offset-3">
        <div class="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Register an account</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body">
                <form method="post">
                    @foreach ($errors->all() as $error)
                        <p class="alert alert-danger">{{ $error }}</p>
                    @endforeach

                    {{ csrf_field() }}
                    <div class="form-group">
                        <label for="name" class="col-lg-12 control-label">Name</label>
                        <div class="col-lg-12">
                            <input type="text" class="form-control" id="name" placehol
der="Name" name="name" value="{{ old('name') }}">
                        </div>
                    </div>

                    <div class="form-group">
```

```
                        <label for="email" class="col-lg-12 control-label">Email</label
>
                        <div class="col-lg-12">
                            <input type="email" class="form-control" id="email" placeh
older="Email" name="email" value="{{ old('email') }}">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="password" class="col-lg-12 control-label">Password
</label>
                        <div class="col-lg-12">
                            <input type="password" class="form-control"  name="password
">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="password" class="col-lg-12 control-label">Confirm
password</label>
                        <div class="col-lg-12">
                            <input type="password" class="form-control"  name="passwor
d_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</butt
on>
                        </div>
                    </div>
                </form>
            </div>
        </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:

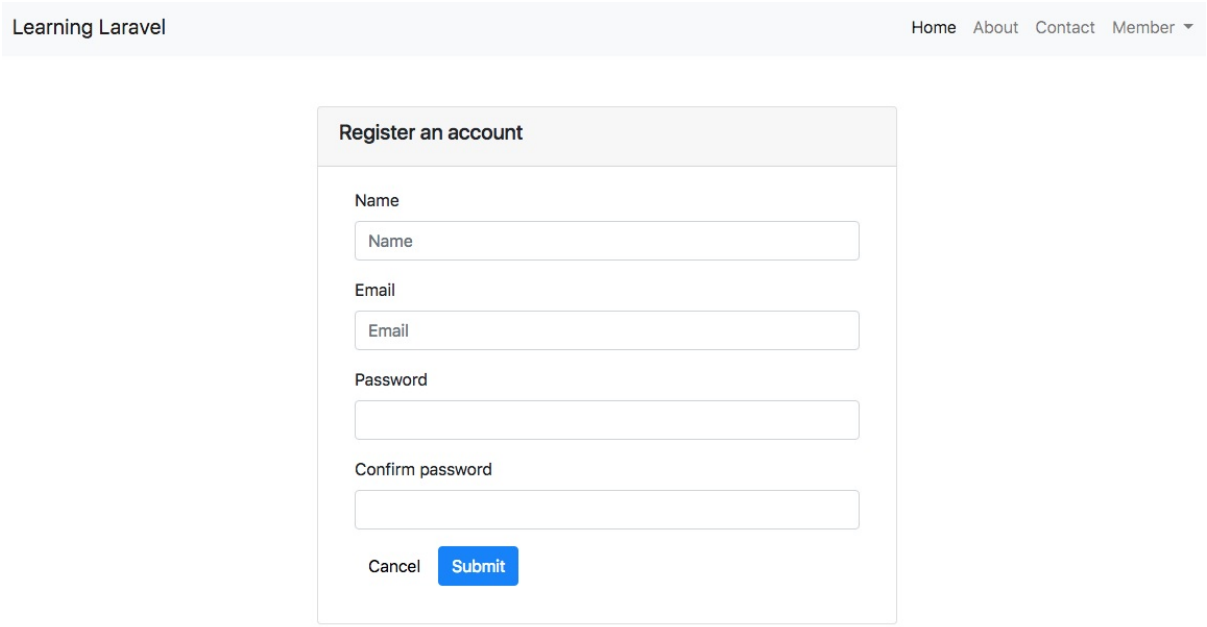
```
<input type="hidden" name="_token" value="{{ csrf_token() }}">
```

You can simply use this helper function to generate the token!

```
{{ csrf_field() }}
```

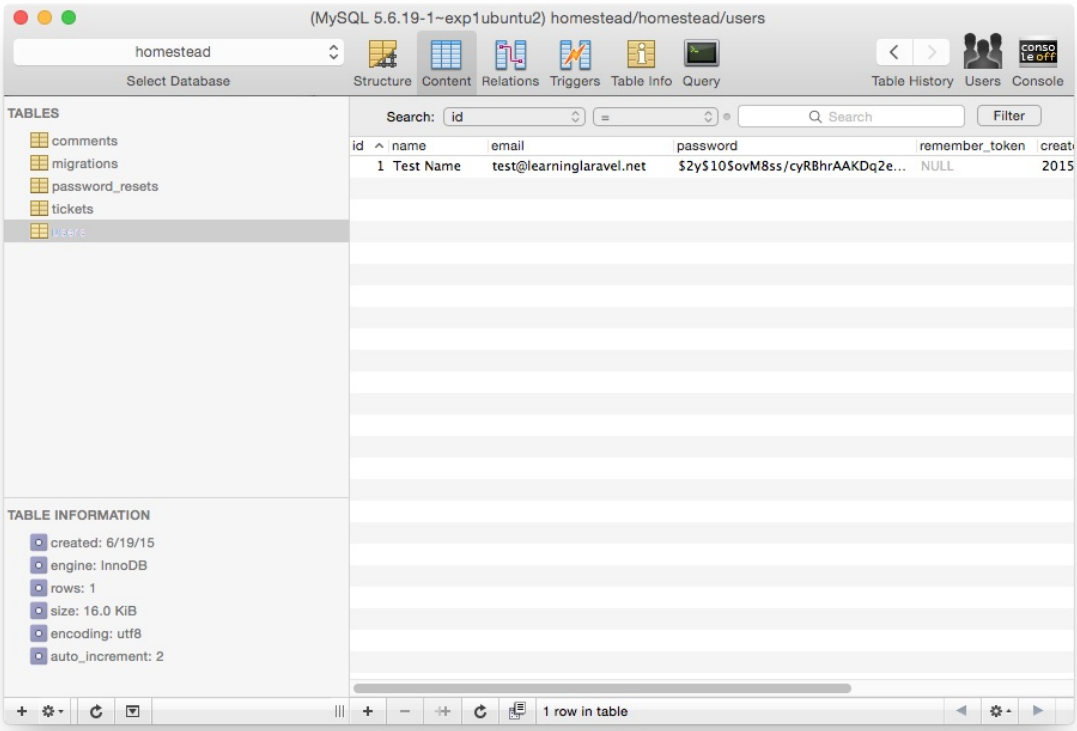
You also may notice that there is a new **old()** method. When the form's validation fails, the users will be redirected back to the form. We use this method to display the old users' input, so they don't have to fill in all the fields again.

Now, go to http://homestead.test/users/register, you should see a nice user registration form.




Fill in all the fields, and hit submit! You've registered a new user!

Check your database now, you should see:




By default, Laravel automatically redirects you to the **/home** URI. If you see this error when you're at http://homestead.test/home:

```
NotFoundHttpException in RouteCollection.php line 161:
```

or

```
Sorry, the page you are looking for could not be found.
```

Then that means your **web.php** file doesn't have the **home** route:

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

You can fix the error by adding the **home route** into your app or you can open the **RegisterController** and take a look at:

```
/**
 * Where to redirect users after login / registration.
 *
 * @var string
```

```
     */
    protected $redirectTo = '/home';
```

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).

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.

To redirect users to other locations, we simply change this **$redirectTo** variable:

```
protected $redirectTo = '/';
```

# Creating a logout functionality

We don't have the **logout** functionality yet, but don't worry, it's very easy to implement.

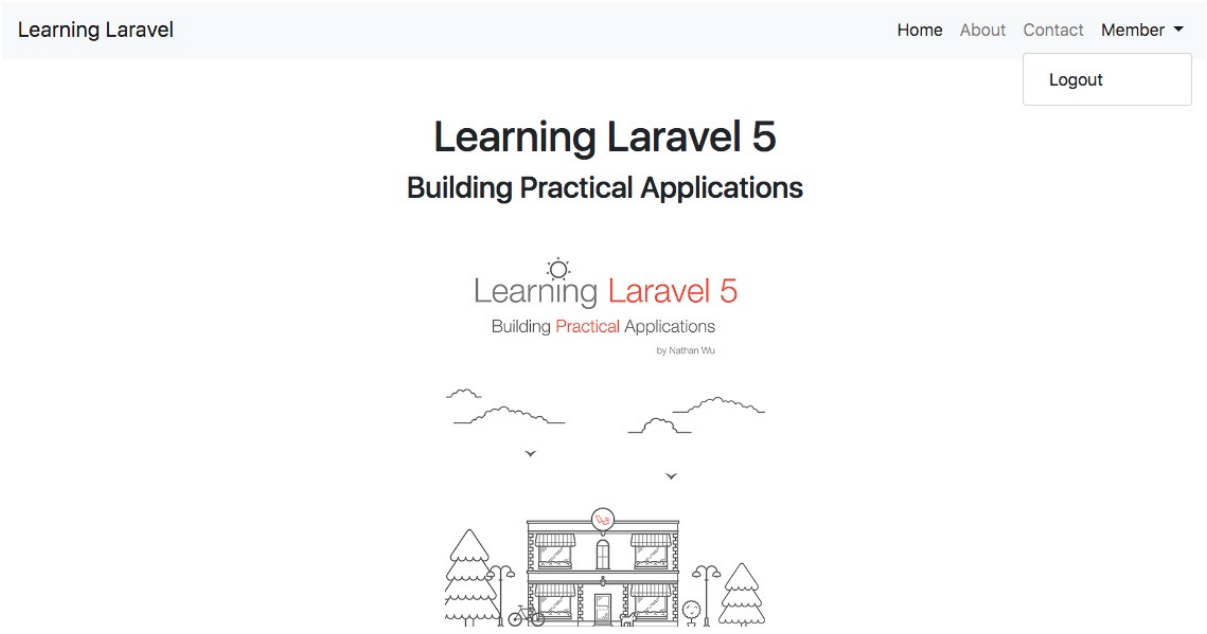
Open the **shared/navbar.blade.php** view, find:

```
<a class="dropdown-item" href="/users/register">Register</a>
<a class="dropdown-item" href="/users/login">Login</a>
```

Replace with the following code:

```
@if (Auth::check())
    <a class="dropdown-item" href="/users/logout">Logout</a>
@else
    <a class="dropdown-item" href="/users/register">Register</a>
    <a class="dropdown-item" href="/users/login">Login</a>
@endif
```

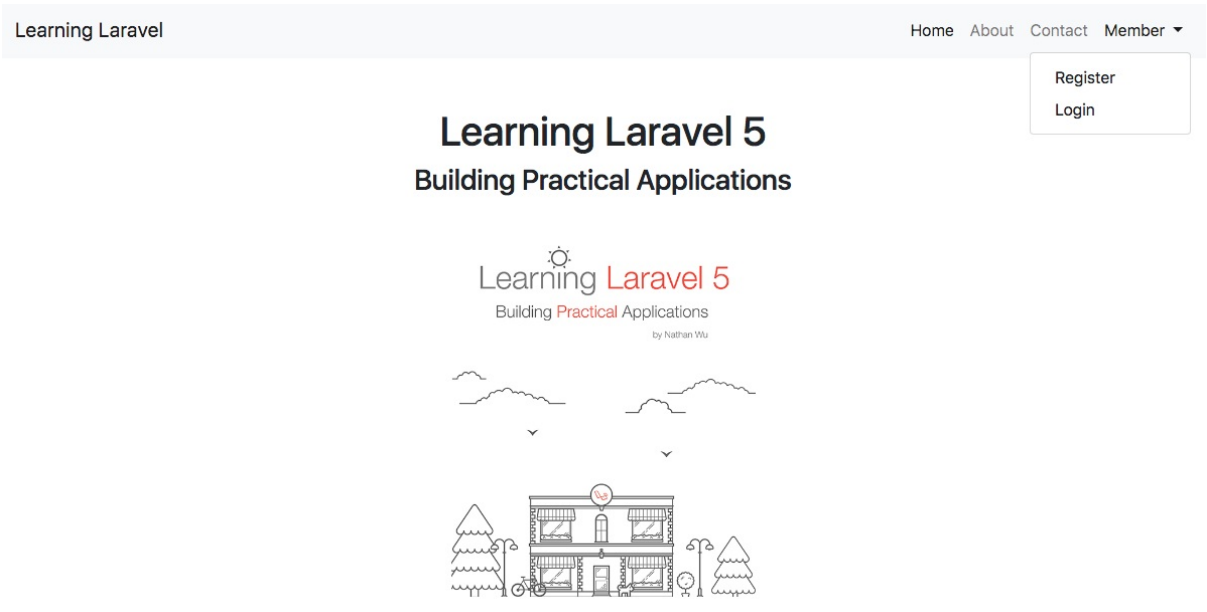
To check whether a user is logged in, we can use the **Auth::check()** method. In the code above, if users are logged in, we will display a logout link.




To make the link work, open **web.php**, add:

```
Route::get('users/logout', 'Auth\LoginController@logout');
```

As you see, when users visit the **users/logout** link, we will use **LoginController's getLogout** action to log the users out.




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\LoginController@showLoginForm');
Route::post('users/login', 'Auth\LoginController@login');
```

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="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Login</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body">
                <form class="form-horizontal" method="post">

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

                    {{ csrf_field() }}

                    <div class="form-group">
                        <label for="email" class="col-lg-12 control-label">Email</label
>
                        <div class="col-lg-12">
                            <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-12 control-label">Password
</label>
                        <div class="col-lg-12">
                            <input type="password" class="form-control"  name="password
">
                        </div>
```

```
                    </div>

                    <div class="form-group">
                        <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>
                </form>
            </div>
        </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.

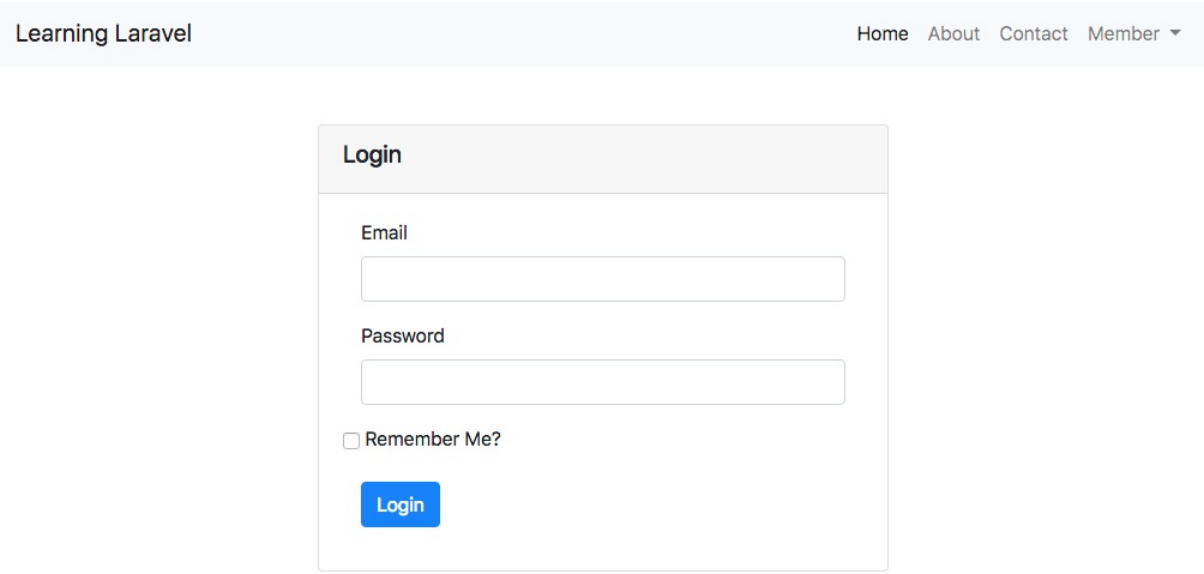
> **Note**: The name of the checkbox should be **"remember"**.

One last step, if you haven't added the **home** route yet, don't forget to open the **LoginController** and change the **$redirectTo** variable:

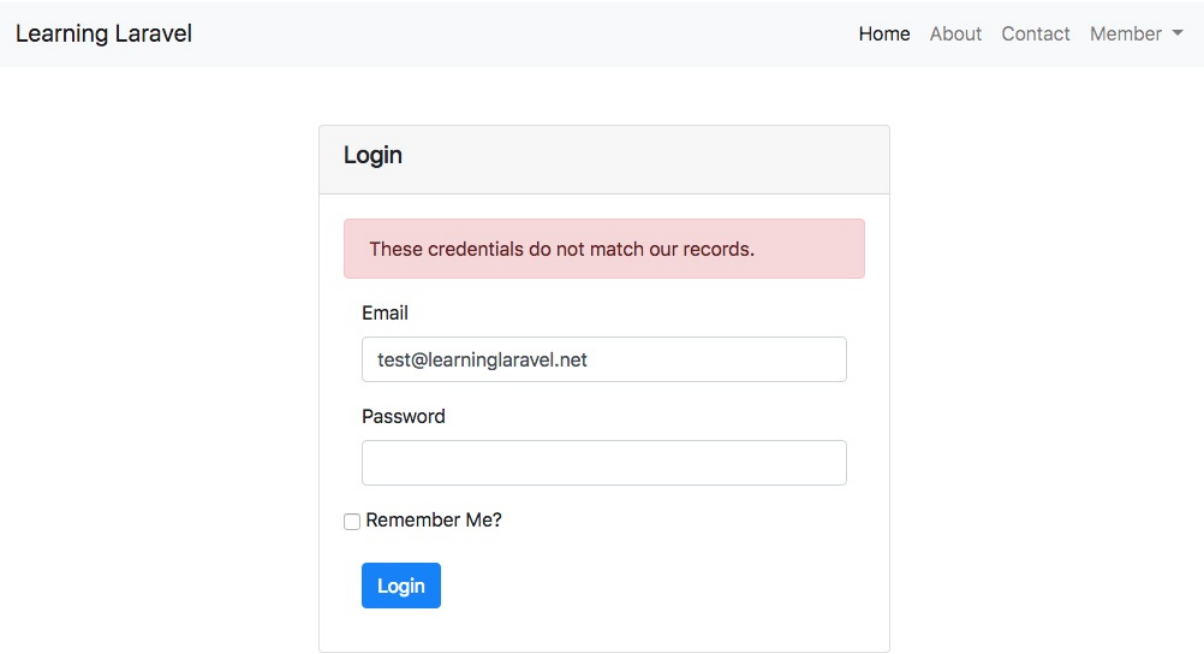
```
protected $redirectTo = '/';
```

Now everything should be working fine.

Let's go to http://homestead.test/users/login and try to login with your email and password!




If you try to login with wrong credentials, you should see this message:

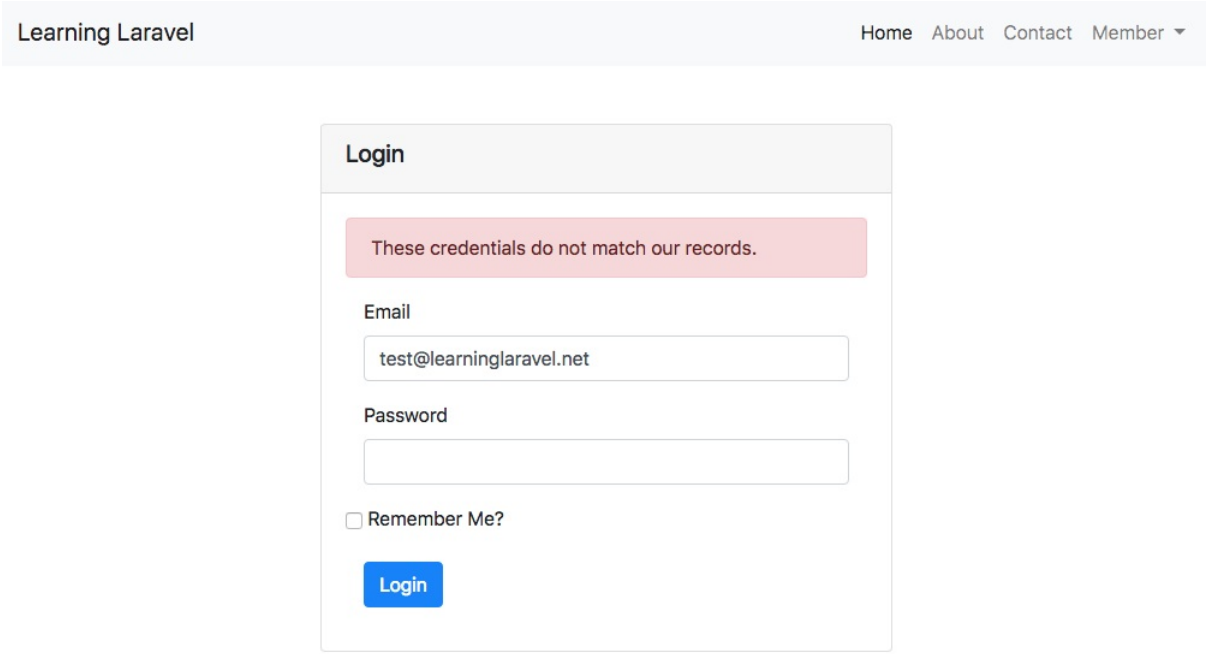



# Add authentication throttling to your application

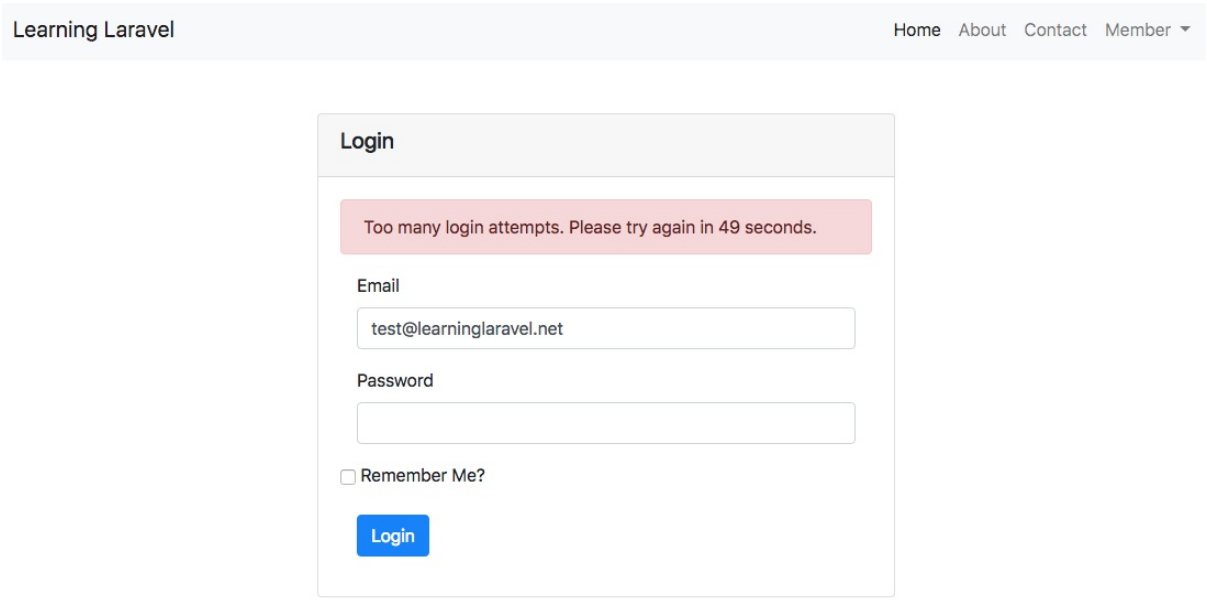
Since **Laravel 5.1.4**, we have a new feature: **"Authentication Throttling"**. This feature is used to throttle login attempts to your application. If users try to log in many times, they can't be able to log in for one minute.

Because we're using the latest version of Laravel and the Laravel's built-in **LoginController** class for authentication, this feature has been implemented already.

Now, let's try to log in with wrong credentials:




When you try to log in many times, an error message would appear:




> **Documentation:** Learn more about Login Throttling at https://laravel.com/docs/master/authentication#login-throttling.

# 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 **web.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.test/admin or any routes that contain the admin prefix, Laravel will understand that we want to access the admin area.

Laravel 5 uses PSR-4 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 **RegisterController.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 middleware** 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.

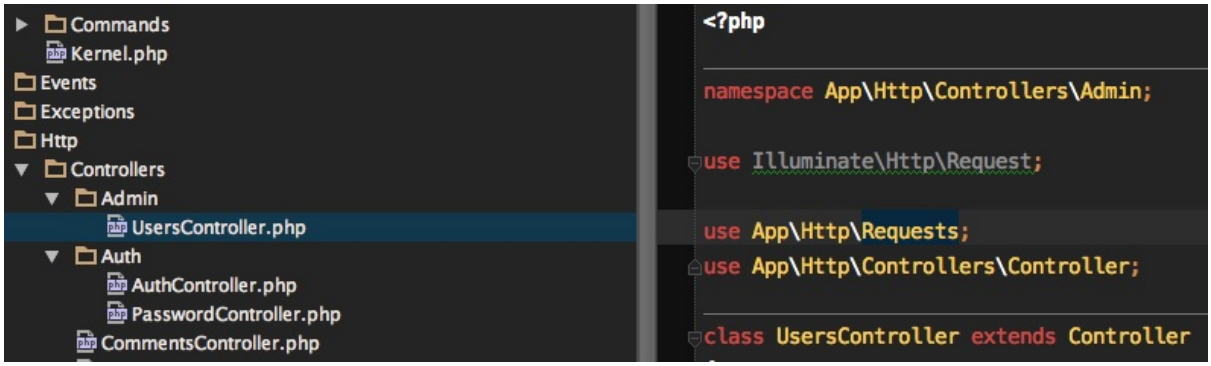
As you know, we have defined a route here:

```
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!




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:

```
class UsersController extends Controller
```

Add above:

```
use App\User;
```

Now, add a new **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.

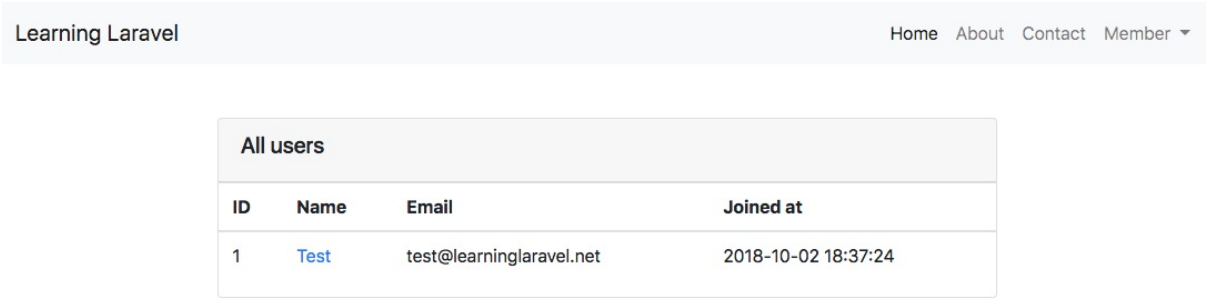
So the index view will be placed at **views/backend/users/index.blade.php**.

Here is the code:

```
@extends('master')
@section('title', 'All users')
@section('content')

    <div class="container col-md-10 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header">
                <h5 class="float-left">All users</h5>
                <div class="clearfix"></div>
            </div>
            <div class="content">
                @if (session('status'))
                    <div class="alert alert-success">
                        {{ session('status') }}
                    </div>
                @endif
                @if ($users->isEmpty())
                    <p> There is no user.</p>
                @else
                    <div class="table-responsive">
                        <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>
                    </div>
```

```
                @endif
            </div>
        </div>
    </div>

@endsection
```

Now, be sure to login to our application and visit http://homestead.test/admin/users




Cool! We can be able to view all the users!

If you don't log in and visit the page, you will get an error:

```
Route [login] not defined.
```

Don't worry, it's because our **middleware** is working fine.

To fix this issue, we'll use named route.

# Using named route

In Laravel 5, we can use the named route feature to generate URLs or redirects for specific routes. Simply put, we can name a route by just using the `name` method:

```
Route::get('yourRoute', 'yourController@yourAction')->name('yourCustomNamedRoute');
```

To fix the login issue, we just need to open **routes/web.php**:

Find:

```
Route::get('users/login', 'Auth\LoginController@showLoginForm')
```

Replace with:

```
Route::get('users/login', 'Auth\LoginController@showLoginForm')->name('login');
```

Everything should be good to go.

Why? Because if you try to access the admin routes when you're not authenticated, you will be redirected to the **login** route.

However, we use **users/login** route to access our login page. That's why Laravel doesn't understand where the **login** route is, and it throws an error. To fix the bug, we just need to update the route and tell Laravel that the `login` route is now `/users/login`.

Simple, isn't it?

> **Documentation:** Learn more about named routes at https://laravel.com/docs/5.7/routing#named-routes

In the next section, we will learn how to use **Middleware**.

# 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:

https://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 some middleware in the **app/Http/Middleware** directory. Open the directory, you'll see these middleware:

1. **EncryptCookies** middleware: Used to encrypt the application's cookies.

2. **RedirectIfAuthenticated** middleware: Redirect users to a page if they're not authenticated.
3. **VerifyCsrfToken** middleware: Used to manage RSRF tokens.
4. **TrimStrings** middleware: Used to automatically trim all the request data.
5. **TrustProxies** middleware: Used to quickly customize the load balancers or proxies that should be trusted by our application.
6. **Authenticate** middleware: Get that path that the users should be redirected to when they're not authenticated
7. **CheckForMaintenanceMode** middleware: When the maintenance mode is enabled, get the URIs that should be reachable.

Let's open the **RedirectIfAuthenticated** middleware:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\Auth;

class RedirectIfAuthenticated
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @param  string|null  $guard
     * @return mixed
     */
    public function handle($request, Closure $next, $guard = null)
    {
        if (Auth::guard($guard)->check()) {
            return redirect('/home');
        }

        return $next($request);
    }
}
```

As you see, this **RedirectIfAuthenticated middleware** is just a class. We use the **handle method** to process all requests and define request filters.

By default, if users are not authenticated, the middleware automatically redirects users to the **/home** URI:

```
return redirect('/home');
```

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
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($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.

**app/Http/Kernel.php**

```php
<?php

namespace App\Http;

use Illuminate\Foundation\Http\Kernel as HttpKernel;

class Kernel extends HttpKernel
{
    /**
     * The application's global HTTP middleware stack.
     *
     * These middleware are run during every request to your application.
     *
     * @var array
     */
    protected $middleware = [
        \App\Http\Middleware\CheckForMaintenanceMode::class,
        \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
        \App\Http\Middleware\TrimStrings::class,
        \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
        \App\Http\Middleware\TrustProxies::class,
    ];

    /**
     * The application's route middleware groups.
     *
     * @var array
     */
    protected $middlewareGroups = [
        'web' => [
            \App\Http\Middleware\EncryptCookies::class,
            \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
            \Illuminate\Session\Middleware\StartSession::class,
            // \Illuminate\Session\Middleware\AuthenticateSession::class,
            \Illuminate\View\Middleware\ShareErrorsFromSession::class,
            \App\Http\Middleware\VerifyCsrfToken::class,
            \Illuminate\Routing\Middleware\SubstituteBindings::class,
        ],

        'api' => [
            'throttle:60,1',
            'bindings',
        ],
    ];

    /**
     * The application's route middleware.
     *
     * These middleware may be assigned to groups or used individually.
     *
     * @var array
     */
```

```
    protected $routeMiddleware = [
        'auth' => \App\Http\Middleware\Authenticate::class,
        'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
        'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
        'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
        'can' => \Illuminate\Auth\Middleware\Authorize::class,
        'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
        'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
        'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
        'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
    ];
}
```

There are three properties: **$middleware**, **$middlewareGroups** 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 = [
    \App\Http\Middleware\Manager::class,
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
    \App\Http\Middleware\TrimStrings::class,
    \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
    \App\Http\Middleware\TrustProxies::class,
];
```

If you want to enable a middleware for just the **web group** (the **routes/web.php**), you can append it to the **$middlewareGroups** property:

```
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        // \Illuminate\Session\Middleware\AuthenticateSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
        \App\Http\Middleware\Manager::class,
    ],

    'api' => [
        'throttle:60,1',
        'bindings',
    ],
];
```

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' => \Illuminate\Auth\Middleware\Authenticate::class,
        'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
        'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
        'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
        'can' => \Illuminate\Auth\Middleware\Authorize::class,
        'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
        'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
        'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::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 **$middleware** property or the **$middlewareGroups** property. Only append the middleware to the **$routeMiddleware** property, because we only want to use it for our admin route group.

Next, open **web.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' => 'manag
er'), function () {
    Route::get('users', 'UsersController@index');
});
```

Well done! Our **Manager** middleware is now active.

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 permissions to our users.

Fortunately, Laravel has many packages that we can use to implement the features easily:

- Entrust: This is the most popular package for adding Role-based Permissions to Laravel 5. However, it's not updated frequently.
- laravel-permission: This is a new package but it's powerful and very easy to use. It is built upon Laravel's authorization functionality.

Because **laravel-permission** is well maintained, we'll be using it.

> **Note:** If you want to learn more about Entrust, please read the first edition of the Learning Laravel 5 book.

# Adding roles and permission to our app using laravel-permission

You can install laravel-permission by running this command:

```
composer require spatie/laravel-permission
```

> **Note:** `vagrant ssh` to your Homestead, go to the **root directory** of your application and run the command there.

```
vagrant@homestead:~/Code/Laravel$ composer require spatie/laravel-permission
Using version ^2.6 for spatie/laravel-permission
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing spatie/laravel-permission (2.6.0): Downloading (100%)
Writing lock file
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover
Discovered Package: fideloper/proxy
Discovered Package: laravel/tinker
Discovered Package: spatie/laravel-permission
Package manifest generated successfully.
```

In Laravel 5.5 or higher, the service provider will automatically get registered. If you're using an older version of Laravel, you might need to open the **config/app.php** file and find the **providers** array, add:

```
'providers' => [
    ...
    Spatie\Permission\PermissionServiceProvider::class,
];
```

> **Note:** that means you can skip the step above because we're using Laravel 5.7.

Next, we can **publish the migration** using this command:

```
php artisan vendor:publish --provider="Spatie\Permission\PermissionServiceProvider" --
tag="migrations"
```

A new **timestamp_create_permission_tables** migration file will be created.

> **Note:** If your **users table** is different, you have to manually update the migration file.

After that, we can create the **role and permission tables** using:

```
php artisan migrate
```

Check your database, you should see some new tables.

We can publish **laravel-permission**'s config file by running this command:

```
php artisan vendor:publish --provider="Spatie\Permission\PermissionServiceProvider" --
tag="config"
```

A new **permission.php** file will be created in the config directory.

Last step, open our **User model** (app/User.php) and update as follows:

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Spatie\Permission\Traits\HasRoles;

class User extends Authenticatable
{
    use HasRoles, Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];
}
```

Good job! You've installed **laravel-permission**!

## Create a new role

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 **web.php** file and update the admin route group as follows:

```
Route::group(array('prefix' => 'admin', 'namespace' => 'Admin', 'middleware' => 'manag
er'), 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**, add a new **create action** as follows:

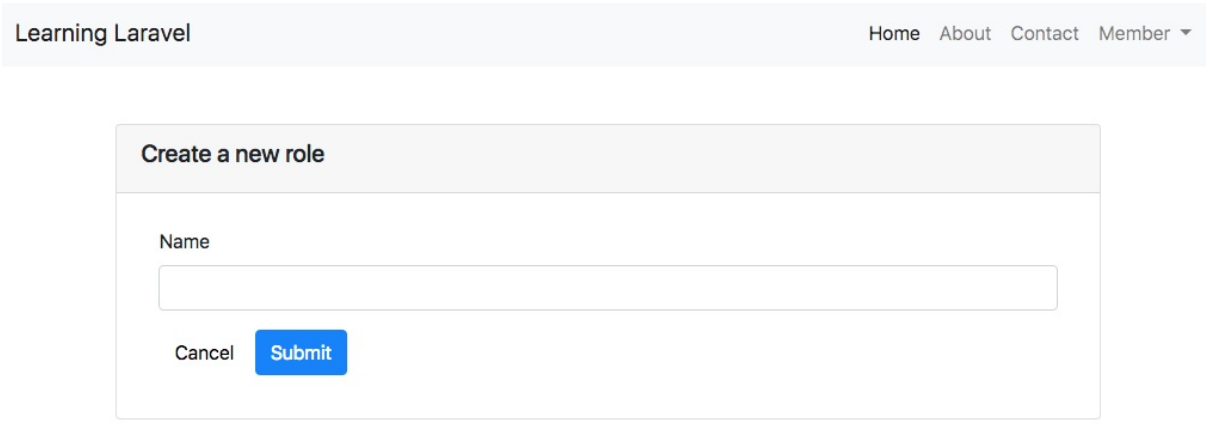
```
class RolesController extends Controller
{
    public function create()
    {
        return view('backend.roles.create');
    }
}
```

Create a new **roles** directory, place it inside the **views/backend** directory. Then create a new view called **create**:

**views/backend/roles/create.blade.php**

```
@extends('master')
@section('title', 'Create A New Role')

@section('content')
    <div class="container col-md-10 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Create a new role</h5>
                <div class="clearfix"></div>
            </div>
```

```
            <div class="card-body mt-2">
                <form 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() }}">

                    <div class="form-group">
                        <label for="name" class="col-lg-12 control-label">Name</label>
                        <div class="col-lg-12">
                            <input type="text" class="form-control" id="name" name="na
me">
                        </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</butt
on>
                        </div>
                    </div>
                </form>
            </div>
        </div>
    </div>
@endsection
```



We should have a nice role creation form at http://homestead.test/admin/roles/create.

Next, let's add a new **store** action into the **RolesController** to save the data:

```
public function store(RoleFormRequest $request)
{
    Role::create(['name' => $request->get('name')]);

    return redirect('/admin/roles/create')->with('status', 'A new role has been create
d!');
}
```

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',
    ];
}
```

We only need to require users to enter the **role's name**.

Here is the updated **RoleFormRequest**:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class RoleFormRequest extends FormRequest
{
    /**
     * 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 [
            'name' => 'required',
        ];
    }
}
```

Be sure that you have told Laravel that you want to use **Role**, **Permission** and **RoleFormRequest** in the **RolesController**:

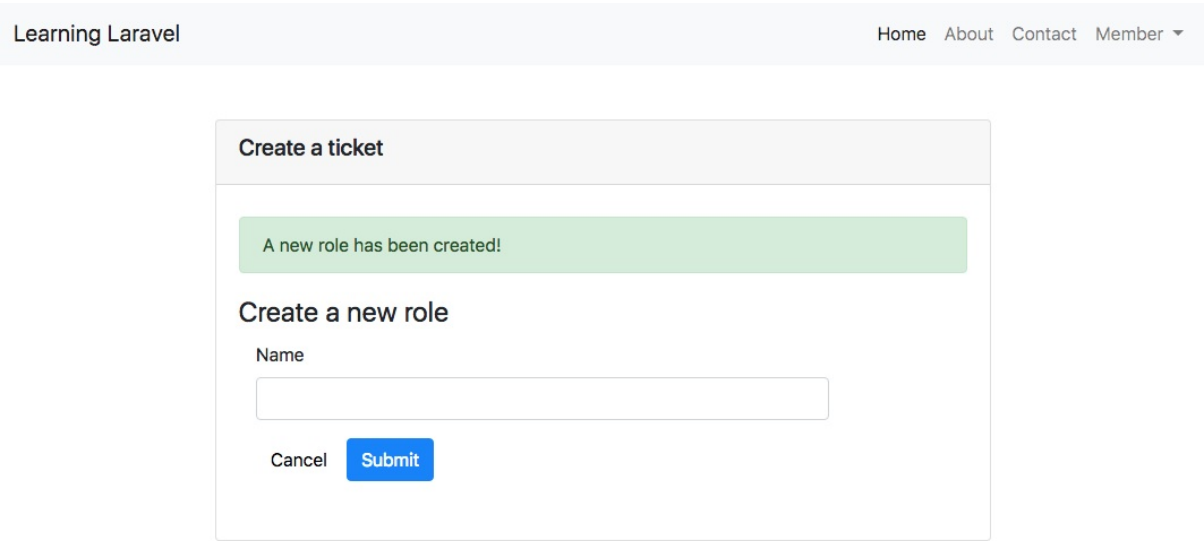
**app/Http/Controllers/Admin/RolesController**

```
<?php

namespace App\Http\Controllers\Admin;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use App\Http\Requests\RoleFormRequest;
use Spatie\Permission\Models\Role;
use Spatie\Permission\Models\Permission;

class RolesController extends Controller
```

Here is the updated **RolesController**:

```
<?php

namespace App\Http\Controllers\Admin;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use App\Http\Requests\RoleFormRequest;
use Spatie\Permission\Models\Role;
use Spatie\Permission\Models\Permission;

class RolesController extends Controller
{
    public function create()
    {
        return view('backend.roles.create');
    }

    public function store(RoleFormRequest $request)
    {
        Role::create(['name' => $request->get('name')]);

        return redirect('/admin/roles/create')->with('status', 'A new role has been cr
eated!');
    }
}
```

Now, go to http://homestead.test/admin/roles/create and create two new roles: **manager** and **member**.




> **Note:** Please note that names are case-sensitive, which means that you have to use "**manager**" instead of "Manager".

To view all roles, add a new `index` action of our **RolesController** as follows:

```
public function index()
{
    $roles = Role::all();
    return view('backend.roles.index', compact('roles'));
}
```

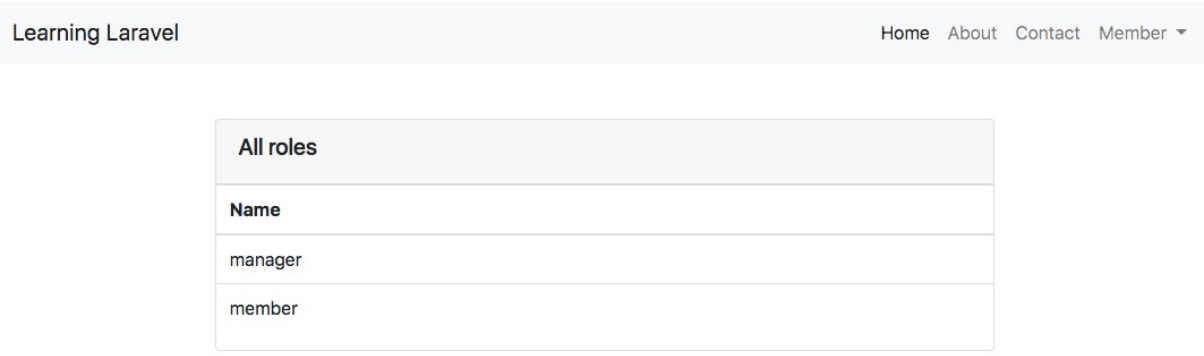
Then create a new **index** view at **views/backend/roles/index.blade.php**:

**views/backend/roles/index.blade.php**

```
@extends('master')
@section('title', 'All roles')
@section('content')

    <div class="container col-md-10 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header">
                <h5 class="float-left">All roles</h5>
                <div class="clearfix"></div>
            </div>
            <div class="content">
                @if (session('status'))
                    <div class="alert alert-success">
                        {{ session('status') }}
                    </div>
                @endif
                @if ($roles->isEmpty())
                    <p> There is no role.</p>
                @else
                    <div class="table-responsive">
                        <table class="table">
                            <thead>
                            <tr>
                                <th>Name</th>
                            </tr>
                            </thead>
                            <tbody>
                            @foreach($roles as $role)
                                <tr>
                                    <td>{{ $role->name }}</td>
                                </tr>
                            @endforeach
                            </tbody>
                        </table>
                    </div>
                @endif
            </div>
        </div>
```

```
    </div>

@endsection
```

Go to http://homestead.test/admin/roles. You should see a list of roles.




## 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 **backend/users/index** view and find:

```
<a href="#">{{ $user->name }} </a>
```

Update the link to:

```
<a href="{{ action('Admin\UsersController@edit', $user->id) }}">{{ $user->name }} </a>
```

Open **Admin/UsersController**, add a new **edit** action:

```
public function edit($id)
{
    $user = User::whereId($id)->firstOrFail();
    $roles = Role::all();
    $selectedRoles = $user->roles()->pluck('name')->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 names 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 Spatie\Permission\Models\Role;
```

Here is the **backend/users/edit** view:

**views/backend/users/edit.blade.php**

```
@extends('master')
@section('name', 'Edit a user')

@section('content')
    <div class="container col-md-6 col-md-offset-3">
        <div class="card mt-5">
            <div class="card-header">
                <h5 class="float-left">Edit user</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                <form 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() }}

                    <div class="form-group">
                        <label for="name" class="col-lg-12 control-label">Name</label>

                        <div class="col-lg-12">
                            <input type="text" class="form-control" id="name" placehol
der="Name" name="name"
                                   value="{{ $user->name }}">
                        </div>
                    </div>
```

```
                    <div class="form-group">
                        <label for="email" class="col-lg-12 control-label">Email</label
>

                        <div class="col-lg-12">
                            <input type="email" class="form-control" id="email" placeh
older="Email" name="email"
                                   value="{{ $user->email }}">
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="select" class="col-lg-12 control-label">Role</label
>

                        <div class="col-lg-12">
                            <select class="form-control" id="role" name="role[]" multi
ple>
                                @foreach($roles as $role)
                                    <option value="{{ $role->name }}"  @if(in_array($r
ole->name, $selectedRoles))
                                    selected="selected" @endif >{{ $role->name }}
                                    </option>
                                @endforeach
                            </select>
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="password" class="col-lg-12 control-label">Password
</label>

                        <div class="col-lg-12">
                            <input type="password" class="form-control" name="password"
>
                        </div>
                    </div>

                    <div class="form-group">
                        <label for="password" class="col-lg-12 control-label">Confirm
password</label>

                        <div class="col-lg-12">
                            <input type="password" class="form-control" name="password
_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</butt
```

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

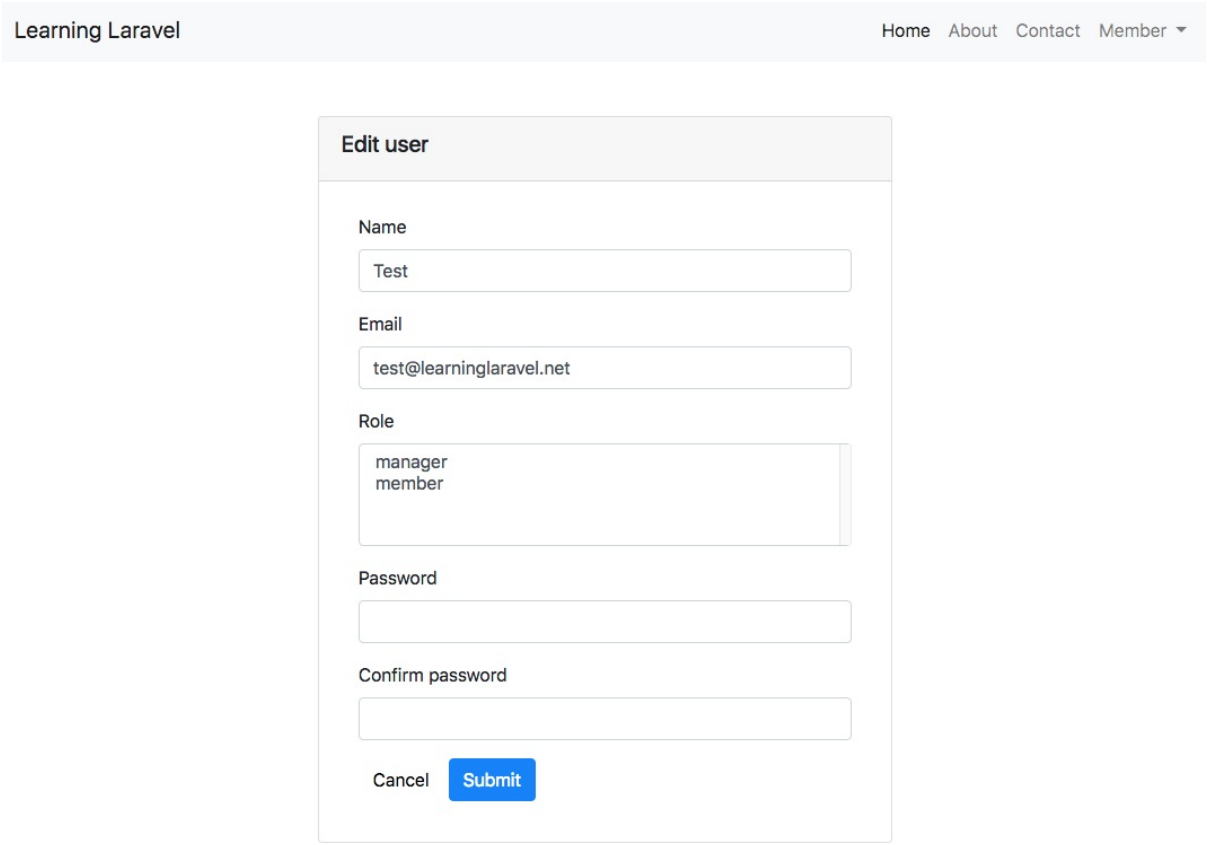
Let's look at this code:

```
<select class="form-control" id="role" name="role[]" multiple>
    @foreach($roles as $role)
        <option value="{{ $role->name }}"  @if(in_array($role->name, $selectedRoles))
        selected="selected" @endif >{{ $role->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 name.

Save the changes and go to http://homestead.test/admin/users. Click on the name of a user that you want to edit.




If you click the **Submit button** now, nothing will happen. We have to add a new **UsersController**'s **update** action 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();

    $user->syncRoles($request->get('role'));

    return redirect(action('Admin\UsersController@edit', $user->id))->with('status', '
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:

https://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 **syncRoles()** method here:

```
$user->syncRoles($request->get('role'));
```

**syncRoles** is a **laravel-permission** method that we can use to automatically **sync** (attach and detach) multiple roles.

Basically, this method will retrieve the **$role** 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.

Here is our updated **UsersController** file:

```
<?php
```

```
namespace App\Http\Controllers\Admin;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use App\User;
use Spatie\Permission\Models\Role;
use App\Http\Requests\UserEditFormRequest;
use Illuminate\Support\Facades\Hash;

class UsersController extends Controller
{
    public function index()
    {
        $users = User::all();
        return view('backend.users.index', compact('users'));
    }

    public function edit($id)
    {
        $user = User::whereId($id)->firstOrFail();
        $roles = Role::all();
        $selectedRoles = $user->roles()->pluck('name')->toArray();
        return view('backend.users.edit', compact('user', 'roles', 'selectedRoles'));
    }

    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();

        $user->syncRoles($request->get('role'));

        return redirect(action('Admin\UsersController@edit', $user->id))->with('status'
, 'The user has been updated!');
    }
}
```

As always, generate the **UserEditFormRequest** by running this command:

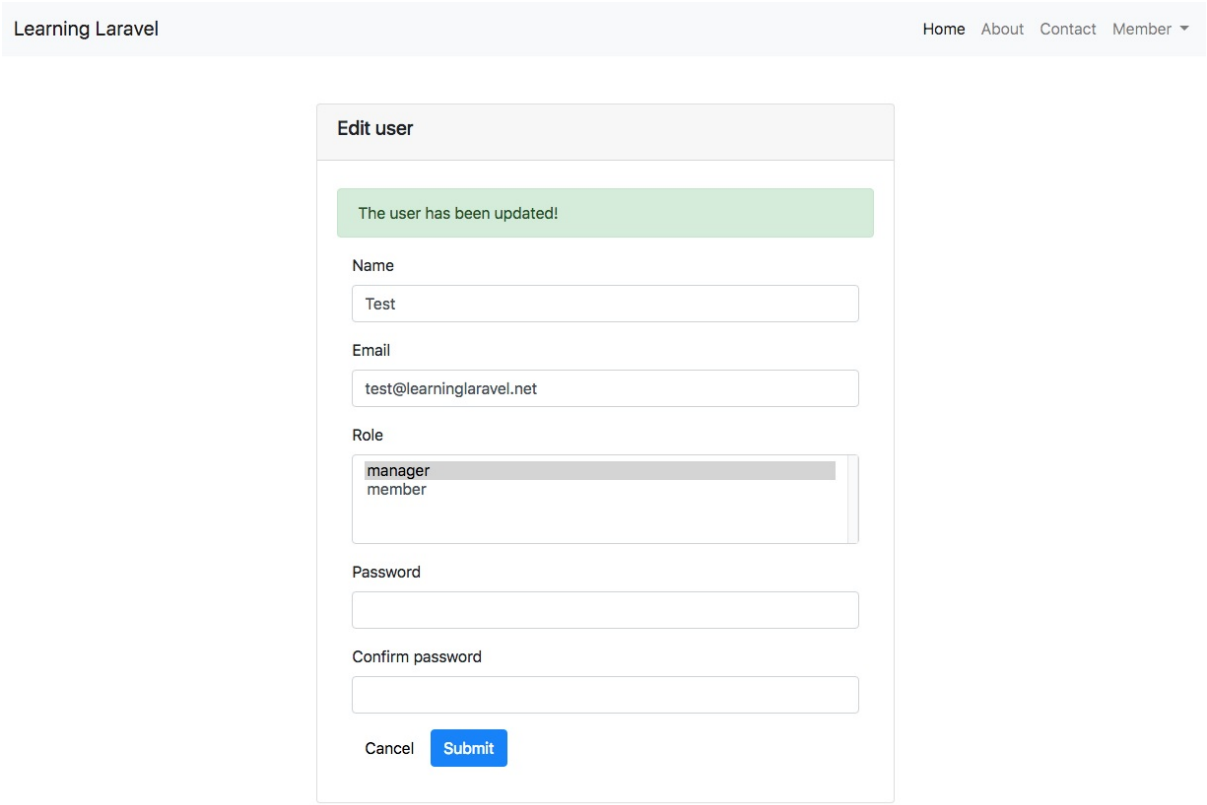
```
php artisan make:request UserEditFormRequest
```

Be sure to change **return false** to **true**. Here are the rules:

### UserEditFormRequest.php

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class UserEditFormRequest extends FormRequest
{
    /**
     * 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 [
            'name' => 'required',
            'email'=> 'required',
            'role'=> 'required',
        ];
    }
}
```

Sweet! Now let's try to assign a role to a user!




> **Note:** If you see any error, make sure that **laravel-permission** is installed properly, and you've added the `HasRoles` trait to your **User** model.

# 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:

**Middleware/Manager.php**

```
<?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('/');
            }
        }
    }
}
```

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('/');
```

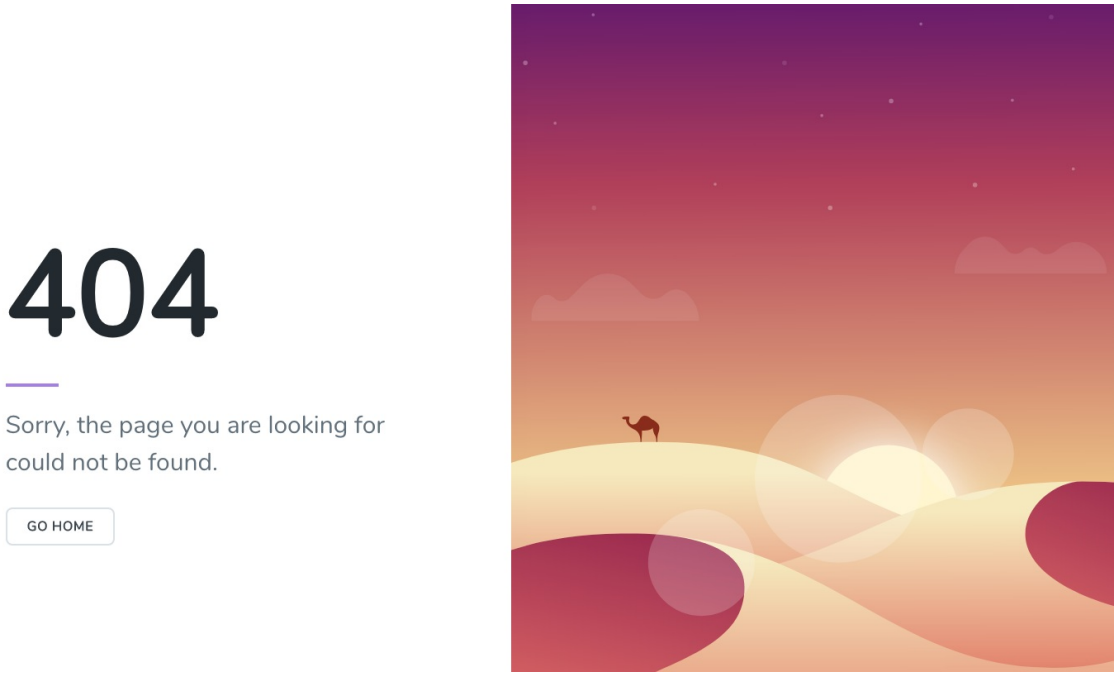
```
}
```

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.

> **Tip:** If you can't access http://homestead.test/admin/users anymore, you can open the **web.php** file, remove the **manager** middleware or change it to **auth**. When the manager middleware is active, only managers (users who have the manager role) can be able to access the admin routes.

## Create an admin dashboard page

Since we haven't created the admin home page, when we access http://homestead.test/admin, there is an error:




> **Note:** You may see a different error message if you're using a different Laravel version. It's normal.

Let's move onto creating the admin home page now so that we can access the admin area easily.

Update our **web.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
```

**Admin/PagesController.php** file will be created. Let's add a new **home action**:

**Admin/PagesController.php**

```
<?php

namespace App\Http\Controllers\Admin;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PagesController extends Controller
{
    public function home()
    {
        return view('backend.home');
    }
}
```

Finally, create a new **home** view in the **backend** directory:

**views/backend/home.blade.php**

```
@extends('master')
@section('title', 'Admin Control Panel')

@section('content')

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

            <div class="col-md-12 mt-5">

                <div class="list-group">
                    <div class="list-group-item">
                        <div class="row-content">
                            <h5><i class="far fa-grin-tongue mb-3"></i> Manage User</h5
>
                            <a href="/admin/users" class="btn btn-warning">All Users</a
>
                        </div>
                    </div>
                    <div class="list-group-separator"></div>
                    <div class="list-group-item">
```

```
                        <div class="row-content">
                            <h5><i class="fa fa-users mb-3"></i> Manage Roles</h5>
                            <a href="/admin/roles" class="btn btn-info">All Roles</a>
                            <a href="/admin/roles/create" class="btn btn-success">Crea
te A Role</a>
                        </div>
                    </div>
                    <div class="list-group-separator"></div>
                    <div class="list-group-item">
                        <div class="row-content">
                            <h5><i class="fas fa-file-signature mb-3"></i> Manage Posts
</h5>
                            <a href="/admin/posts" class="btn btn-info">All Posts</a>
                            <a href="/admin/posts/create" class="btn btn-success">Crea
te A Post</a>
                        </div>
                    </div>
                    <div class="list-group-separator"></div>
                </div>

            </div>

        </div>
    </div>

@endsection
```

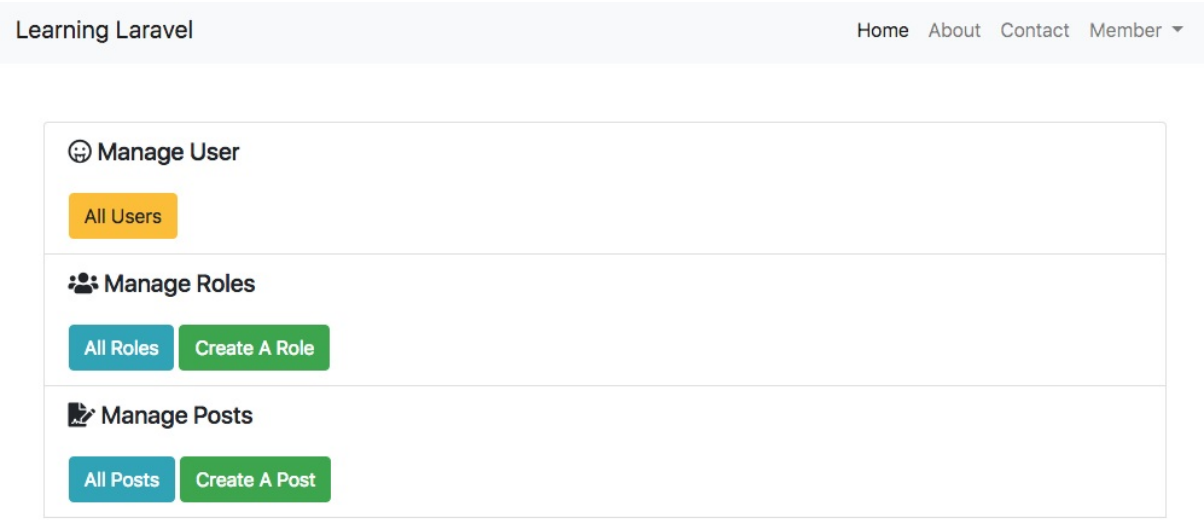
We're using Font Awesome here for displaying some icons, so let's open the **master.blade.php** file, and find:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bo
otstrap.min.css">
```

Add this line below to integrate **Font Awesome**:

```
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.3.1/css/all.css">
```

Great! Let's visit http://homestead.test/admin:




Awesome! This is our admin home page.

I just add some buttons to access other admin pages here. Feel free to change the layout to your liking.

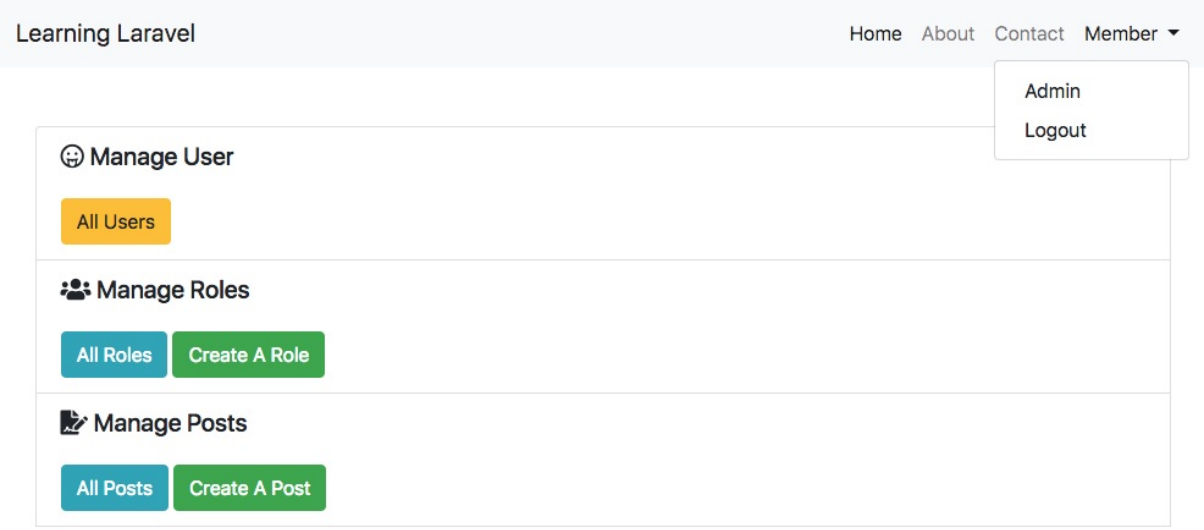
One more thing, how about adding a link to our navigation bar to access the admin area easier? Open the **shared/navbar** view and find:

```
@if (Auth::check())
    <a class="dropdown-item" href="/users/logout">Logout</a>
@else
```

Update to:

```
@if (Auth::check())
    @role('manager')
        <a class="dropdown-item" href="/admin">Admin</a>
    @endrole
    <a class="dropdown-item" href="/users/logout">Logout</a>
@else
```

As you see, we can use the **@role** Blade directive to check if the user is a manager in **views** as well.




# 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')->nullable();
        $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 **web.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 --resource
```

Open **Admin/PostsController**, update the **create** action as follows:

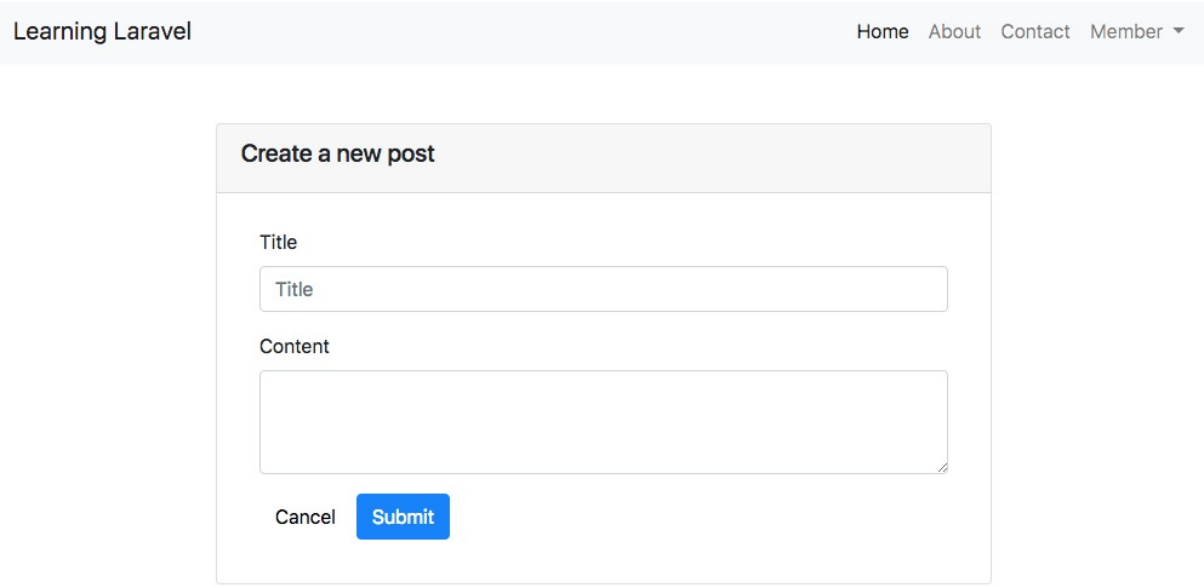
```
public function create()
{
    return view('backend.posts.create');
}
```

Create a new **posts** folder in the **backend** directory. Place a new **create** view there:

**views/backend/posts/create.blade.php**

```
@extends('master')
@section('title', 'Create A New Post')

@section('content')
    <div class="container col-md-10 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Create a new post</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                <form 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() }}">

                <div class="form-group">
                    <label for="title" class="col-lg-12 control-label">Title</label>
                    <div class="col-lg-12">
                        <input type="text" class="form-control" id="title" placeholder=
"Title" name="title">
                    </div>
                </div>
                <div class="form-group">
                    <label for="content" class="col-lg-12 control-label">Content</label
>
                    <div class="col-lg-12">
                        <textarea class="form-control" rows="3" id="content" name="con
tent"></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>
            </form>
            </div>
        </div>
    </div>
@endsection
```




When we visit http://homestead.test/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). We have to create our **category_post** manually.

Run this Artisan command to generate the **category_post** migration:

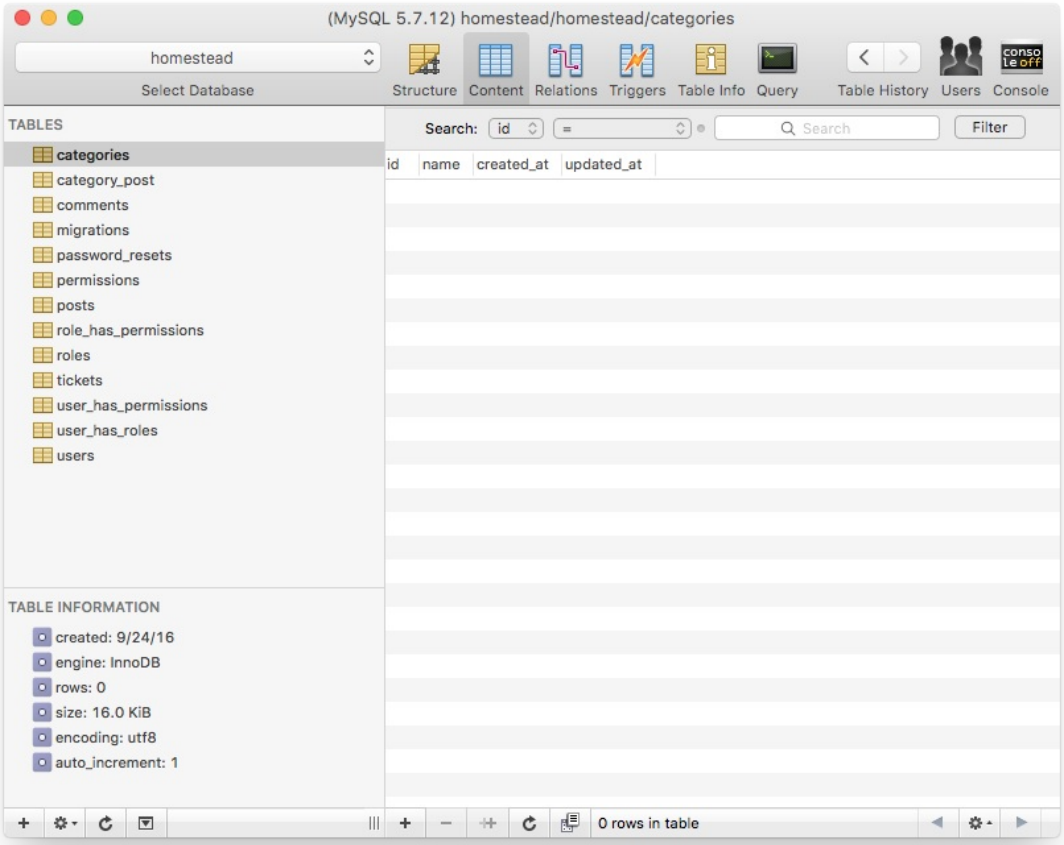
```
php artisan make:migration create_category_post_table
```

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
```




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:

**app/Post.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();
    }

}
```

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:

**app/Category.php**

```
<?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 **web.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 --resource
```

Open the newly created **CategoriesController** and update the **create** action:

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

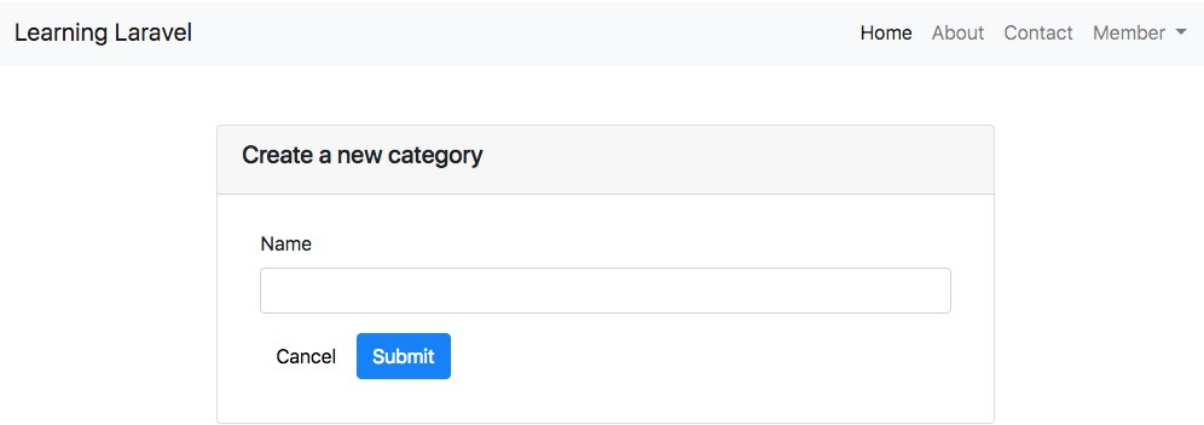
As usual, create a new **categories** folder inside the **backend** directory. Put a new **create** view in the **categories** directory:

**backend/categories/create.blade.php**

```
@extends('master')
@section('title', 'Create A New Category')

@section('content')
    <div class="container col-md-10 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Create a new category</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                <form 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() }}">

                    <div class="form-group">
                        <label for="name" class="col-lg-12 control-label">Name</label>
                        <div class="col-lg-12">
                            <input type="text" class="form-control" id="name" name="na
me">
                        </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</butt
on>
                        </div>
                    </div>
                </form>
            </div>
        </div>
    </div>
@endsection
```

Visit http://homestead.test/admin/categories/create, you should have a form to 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 be
en 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:

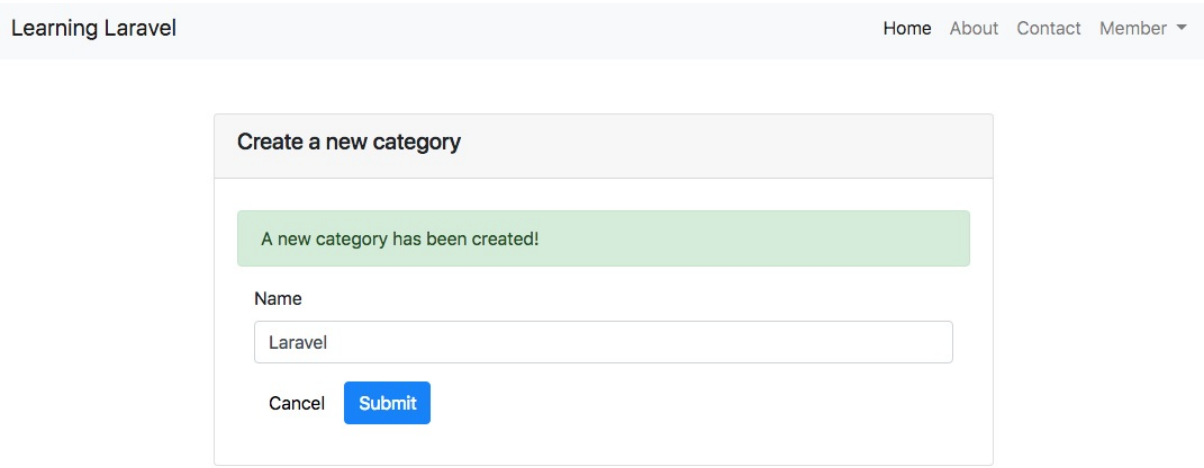
```
public function authorize()
{
    return true;
}

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

Here is the updated **CategoryFormRequest**:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class CategoryFormRequest extends FormRequest
{
    /**
     * 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 [
            'name'=> 'required|min:3',
        ];
    }
}
```

Save the changes and visit the category form again. Create some categories (News, Laravel, Announcement, etc.) :




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:

**backend/categories/index.blade.php**

```
@extends('master')
@section('title', 'All categories')
@section('content')

    <div class="container col-md-10 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header">
                <h5 class="float-left">All categories</h5>
```

```
                <div class="clearfix"></div>
            </div>
            <div class="content">
            @if (session('status'))
                <div class="alert alert-success">
                    {{ session('status') }}
                </div>
            @endif
            @if ($categories->isEmpty())
                <p> There is no category.</p>
            @else
            <div class="table-responsive">
                <table class="table">
                    <tbody>
                    @foreach($categories as $category)
                        <tr>
                            <td>{{ $category->name }}</td>
                        </tr>
                    @endforeach
                    </tbody>
                </table>
            </div>
            @endif
        </div>
    </div>

@endsection
```

Now you can be able to view all categories at http://homestead.test/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):

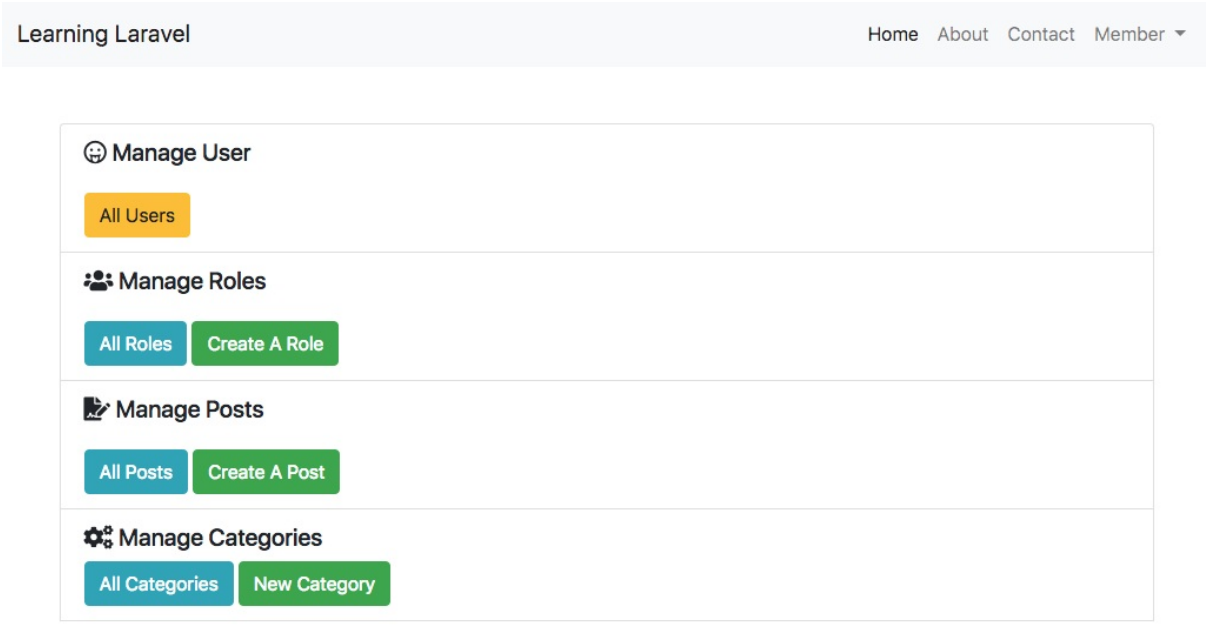
```
        <a href="/admin/posts/create" class="btn btn-primary">Create A Post</a>
    </div>
```

```
</div>
<div class="list-group-separator"></div>
```

Add below:

```
<div class="list-group-item">
    <div class="row-content">
        <h5><i class="fas fa-cogs"></i> Manage Categories</h5>
        <a href="/admin/categories" class="btn btn-info">All Categories</a>
        <a href="/admin/categories/create" class="btn btn-success">New Category</a>
    </div>
</div>
<div class="list-group-separator"></div>
```

Here is our new admin homepage:




## Select categories when creating a post

Now we can create a new post with categories.

Open **Admin/PostsController** and find:

```
class PostsController extends Controller
```

Add above:

```
use App\Category;
use App\Post;
use Illuminate\Support\Str;
use Illuminate\Support\Facades\Auth;
```

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-12 control-label">Content</label>
    <div class="col-lg-12">
        <textarea class="form-control" rows="3" id="content" name="content"></textarea>

    </div>
</div>
```

Add this code below to allow users to select the categories:

```
<div class="form-group">
    <label for="categories" class="col-lg-12 control-label">Categories</label>

    <div class="col-lg-12">
        <select class="form-control" id="category" name="categories[]" multiple>
            @foreach($categories as $category)
                <option value="{{ $category->id }}">
                    {{ $category->name }}
                </option>
            @endforeach
        </select>
    </div>
</div>
```

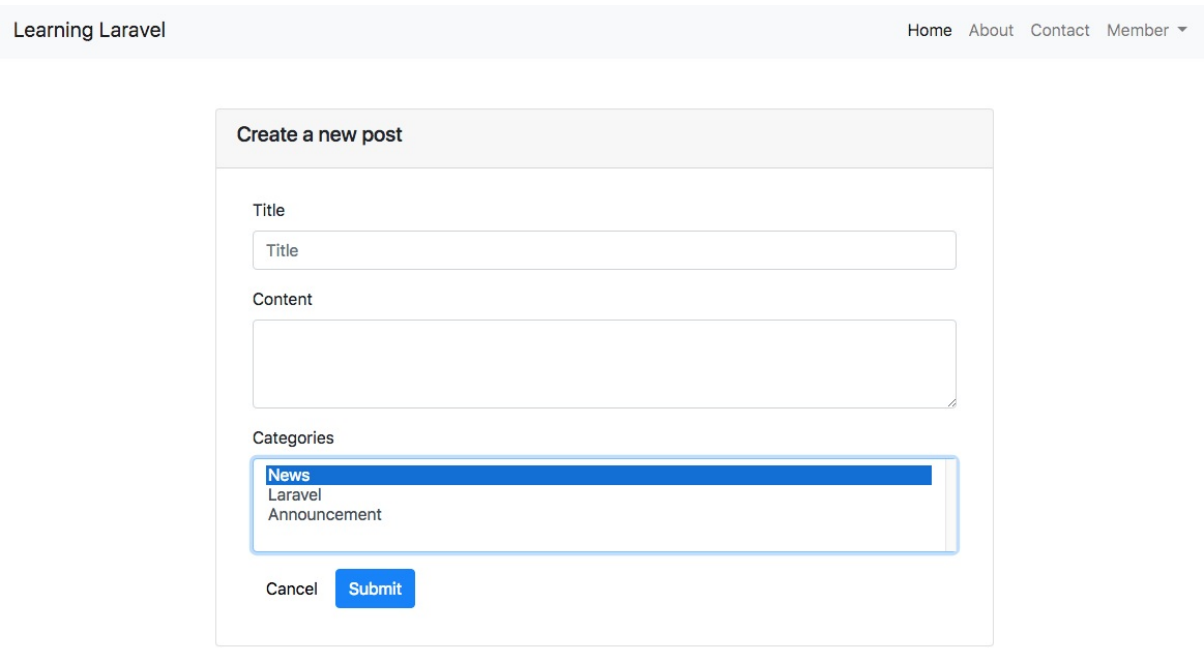
Here is the updated **create** view:

**backend/posts/create.blade.php**

```
@extends('master')
```

```
@section('title', 'Create A New Post')

@section('content')
    <div class="container col-md-10 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Create a new post</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                <form 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() }}">

                <div class="form-group">
                    <label for="title" class="col-lg-12 control-label">Title</label>
                    <div class="col-lg-12">
                        <input type="text" class="form-control" id="title" placeholder=
"Title" name="title">
                    </div>
                </div>
                <div class="form-group">
                    <label for="content" class="col-lg-12 control-label">Content</label
>
                    <div class="col-lg-12">
                        <textarea class="form-control" rows="3" id="content" name="con
tent"></textarea>
                    </div>
                </div>
                <div class="form-group">
                    <label for="categories" class="col-lg-12 control-label">Categories
</label>

                    <div class="col-lg-12">
                        <select class="form-control" id="category" name="categories[]"
multiple>
                            @foreach($categories as $category)
                                <option value="{{ $category->id }}">
                                    {{ $category->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>
            </form>
            </div>
        </div>
    </div>
@endsection
```




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 **Admin/PostsController** 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)
{
    $user_id = Auth::user()->id;
    $post= new Post(array(
        'title' => $request->get('title'),
        'content' => $request->get('content'),
        'slug' => Str::slug($request->get('title'), '-'),
        'user_id' => $user_id
    ));

    $post->save();
    $post->categories()->sync($request->get('categories'));

    return redirect('/admin/posts/create')->with('status', 'The post has been created!'
);
}
```

To create the post's slug, we can use **Str::slug** method to automatically generate a **friendly slug** from the given **post's title**.

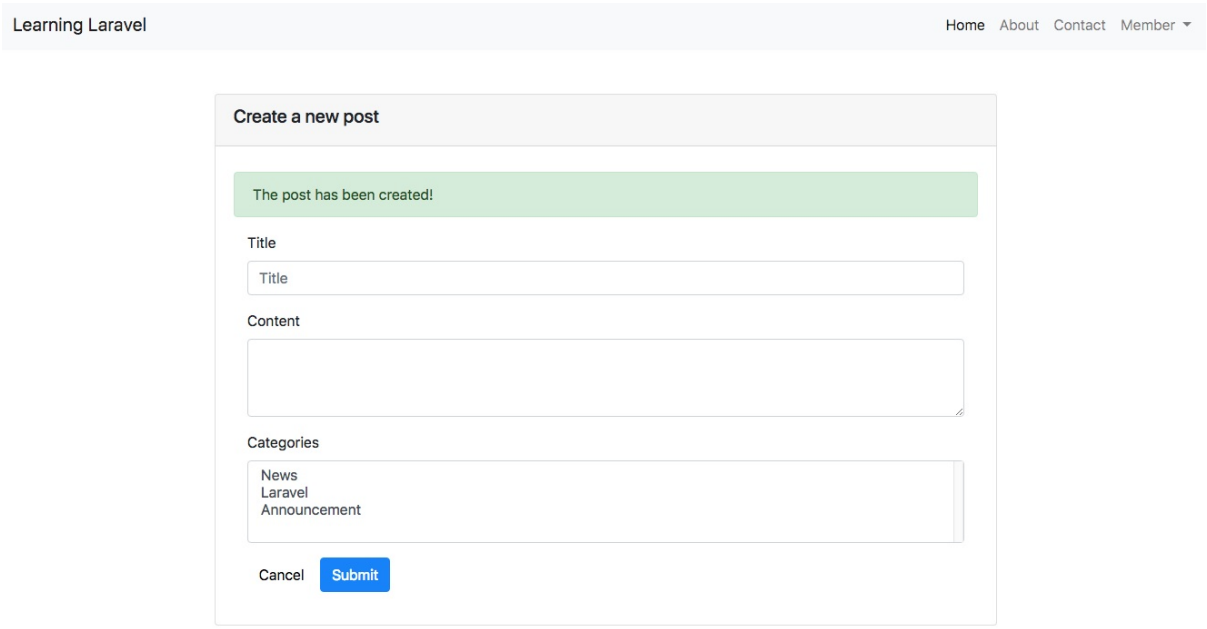
As you see, we also use the **Auth** facade to get the **current user id**.

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.




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 **Admin/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**:

**backend/posts/index.blade.php**

```
@extends('master')
@section('title', 'All posts')
@section('content')

    <div class="container col-md-10 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header">
                <h5 class="float-left">All posts</h5>
                <div class="clearfix"></div>
            </div>
            <div class="content">
                @if (session('status'))
                    <div class="alert alert-success">
                        {{ session('status') }}
                    </div>
                @endif
                @if ($posts->isEmpty())
                    <p> There is no post.</p>
                @else
                    <div class="table-responsive">
                        <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>
                    </div>
                @endif
            </div>
```

```
        </div>
    </div>

@endsection
```

Go to http://homestead.test/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 of the **Admin/PostsController** to display a post edit form:

```
public function edit($id)
{
    $post = Post::whereId($id)->firstOrFail();
    $categories = Category::all();
    $selectedCategories = $post->categories->pluck('id')->toArray();
    return view('backend.posts.edit', compact('post', 'categories', 'selectedCategorie
s'));
}
```

Create a new **edit** view at **backend/posts/edit.blade.php**:

**backend/posts/edit.blade.php**

```
@extends('master')
@section('title', 'Edit a post')

@section('content')
    <div class="container col-md-10 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Edit a post</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                <form 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() }}">

                <div class="form-group">
                    <label for="title" class="col-lg-12 control-label">Title</label>
                    <div class="col-lg-12">
                        <input type="text" class="form-control" id="title" placeholder
="Title" name="title" value="{{ $post->title }}">
                    </div>
                </div>
                <div class="form-group">
                    <label for="content" class="col-lg-12 control-label">Content</label
>
                    <div class="col-lg-12">
                        <textarea class="form-control" rows="3" id="content" name="con
tent">{{ $post->content }}</textarea>
                    </div>
                </div>
                <div class="form-group">
                    <label for="categories" class="col-lg-12 control-label">Categories
</label>

                    <div class="col-lg-12">
                        <select class="form-control" id="category" name="categories[]"
multiple>
                            @foreach($categories as $category)
                                <option value="{{ $category->id }}"  @if(in_array($cat
egory->id, $selectedCategories))
                                selected="selected" @endif >{{ $category->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>
            </form>
            </div>
        </div>
    </div>
@endsection
```

After creating the form, you'll need to modify the **posts/index** view to add links to the posts.

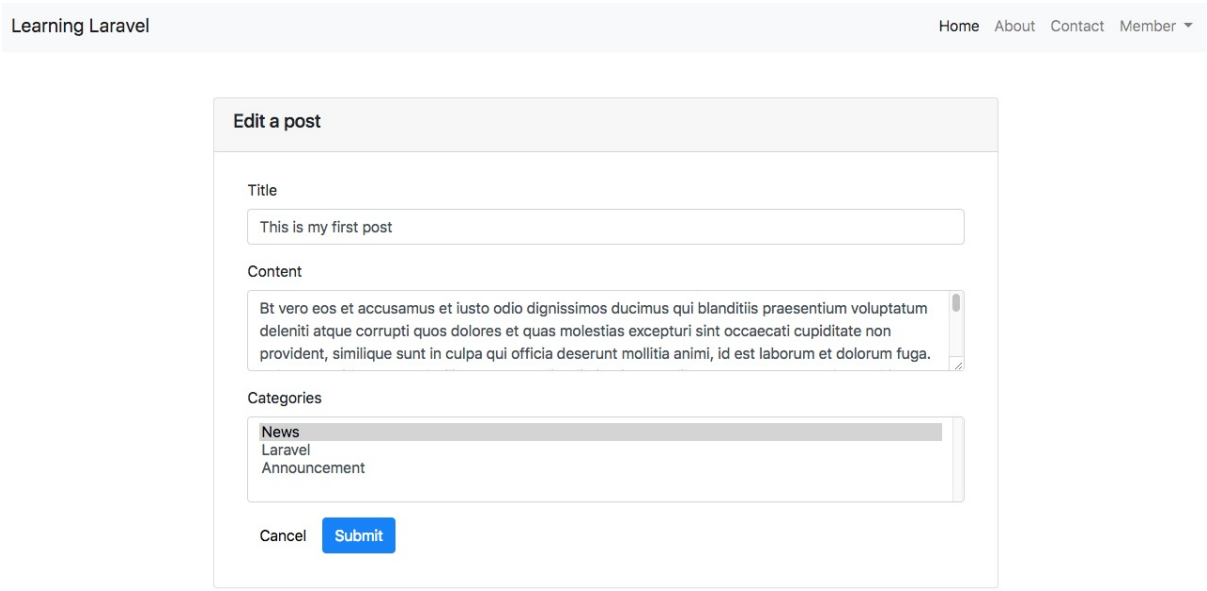
Find:

```
<a href="#">{{ $post->title }} </a>
```

Edit to:

```
<a href="{{ action('Admin\PostsController@edit', $post->id) }}">{{ $post->title }} </a>
```

Go to http://homestead.test/admin/posts and click on the post title, you should be able to see a 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 **Admin/PostsController** and find:

```
class PostsController extends Controller
```

Add above:

```
use App\Http\Requests\PostEditFormRequest;
```

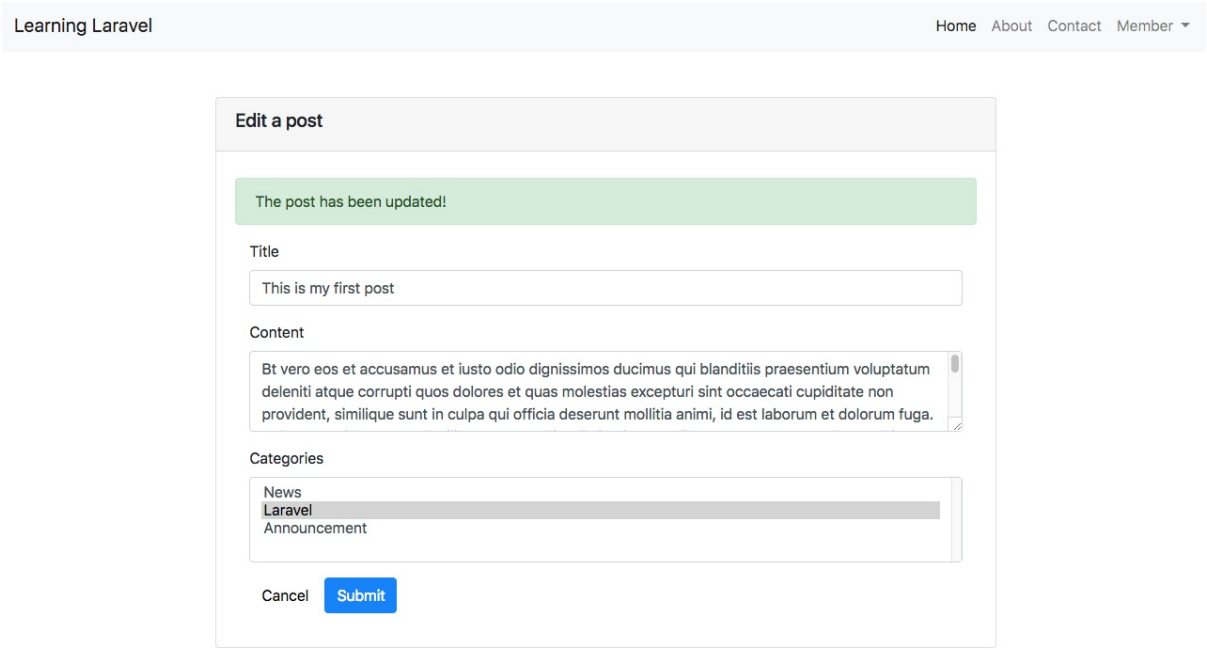
Finally, update the **update** method to save the changes to our database:

```
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('status', '
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.




Well done! Once submitted, you should be able to update the 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 **web.php**:

```
Route::get('/blog', 'BlogController@index');
```

We would need to create the **BlogController**:

```
php artisan make:controller BlogController --resource
```

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:

**views/blog/index.blade.php**

```
@extends('master')
@section('title', 'Blog')
@section('content')

    <div class="container col-md-10 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="card mt-4">
                    <div class="card-header">{{ $post->title }}</div>
                    <div class="card-body">
                        {{ mb_substr($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.

> **Tip:** 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: http://homestead.test/blog


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 **web.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 (**App\Ticket** or **App\Post**).

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();
        });
    }
```

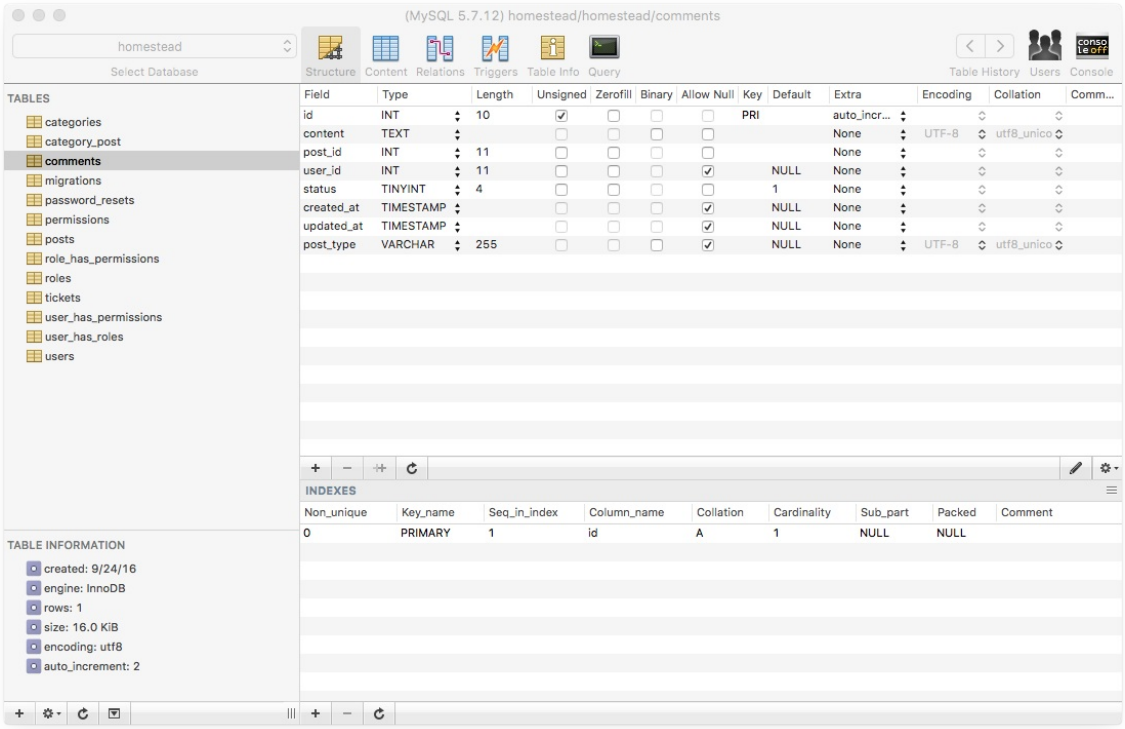
```
}

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.




It's time to build the Polymorphic relationship. Open the **Comment** model, update it as follows:

```
<?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

namespace App;

use Illuminate\Database\Eloquent\Model;

class Ticket extends Model
{
    protected $guarded = ['id'];

    public function comments()
    {
        return $this->morphMany('App\Comment', 'post');
    }
}
```

Open the **Post** model, update it as follows:

```
<?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 be used to get all post's comments.

Done! You've defined Polymorphic relations.

Now, open **BlogController** and update the **show** method:

```
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:

**views/blog/show.blade.php**

```
@extends('master')
@section('title', 'View a post')
@section('content')

    <div class="container col-md-10 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-body">
                <h2 class="header">{{ $post->title }}</h2>
                <p> {{ $post->content }} </p>
            </div>
            <div class="clearfix"></div>
        </div>

        @foreach($comments as $comment)
            <div class="card mt-4">
                <div class="card-body">
                    {{ $comment->content }}
                </div>
            </div>
        @endforeach

        <div class="card mt-4">
            <div class="card-body">
                <form 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">
                    <div class="form-group">
                        <legend>Comment</legend>
                    </div>
                    <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>
                </form>
            </div>
        </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: **"App\Post"** (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 **"App\Ticket"** value, to let Laravel know that the comments of this view belong to the **Ticket** model.

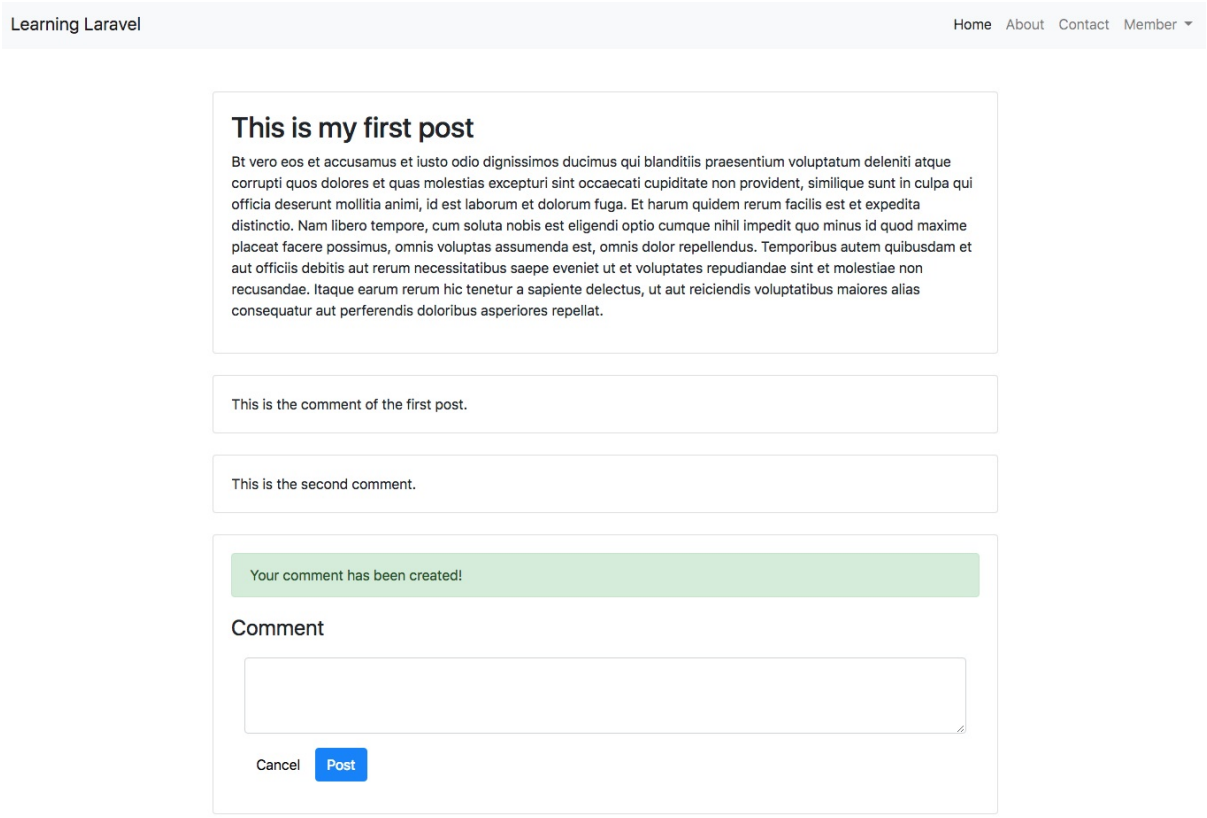
One more thing to do, open the **blog/index** view and find:

```
<div class="card-header">{{ $post->title }}</div>
```

Modify to:

```
<div class="card-header"><a href="{{ action('BlogController@show', $post->slug) }}">{{
$post->title }}</a></div>
```

Now, go to your browser and view a post. Try to post a comment as well.




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 comment | 2 | NULL | 1 | 2018-10-02 14:41:11 | 2018-10-02 14:41:11 | NULL |
| 2 | This is my second comment | 2 | NULL | 1 | 2018-10-02 14:45:54 | 2018-10-02 14:45:54 | NULL |
| 3 | This is the comment of the first post. | 1 | NULL | 1 | 2018-10-03 19:29:40 | 2018-10-03 19:29:40 | App\Post |
| 4 | This is the second comment. | 1 | NULL | 1 | 2018-10-03 19:30:47 | 2018-10-03 19:30:47 | App\Post |

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.

One last step (this is optional), if you want to access the blog page easier, you can add a link to our navbar:

Open **shared/navbar.blade.php**, and find:

```
<li class="nav-item">
    <a class="nav-link" href="/about">About</a>
</li>
```

Add above:

```
<li class="nav-item">
    <a class="nav-link" href="/blog">Blog</a>
</li>
```

# 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.

Update the **run()** method as follows:

```
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:

https://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

> **Note:** If you're using Laravel 5.1 or newer, the Faker library has been already installed by default.

After writing the seeder class, open **database/seeds/DatabaseSeeder.php**.

```
public function run()
{
    // $this->call(UsersTableSeeder::class);
}
```

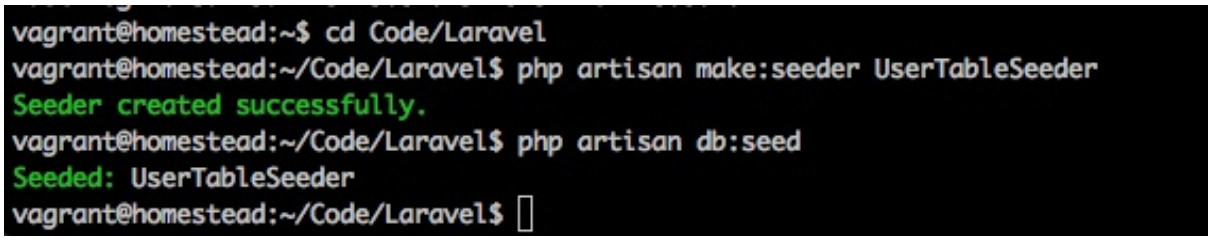
Normally, we would create many seeder classes to seed different dummy data. For example, **UserTableSeeder** is used to create fake users, **PostTableSeeder** is used to create dummy posts, etc. We use the **DatabaseSeeder** file to control the order of seeder classes. In this case, we only have one UserTableSeeder class, so let's write like this:

```
public function run()
{
     $this->call(UserTableSeeder::class);
}
```

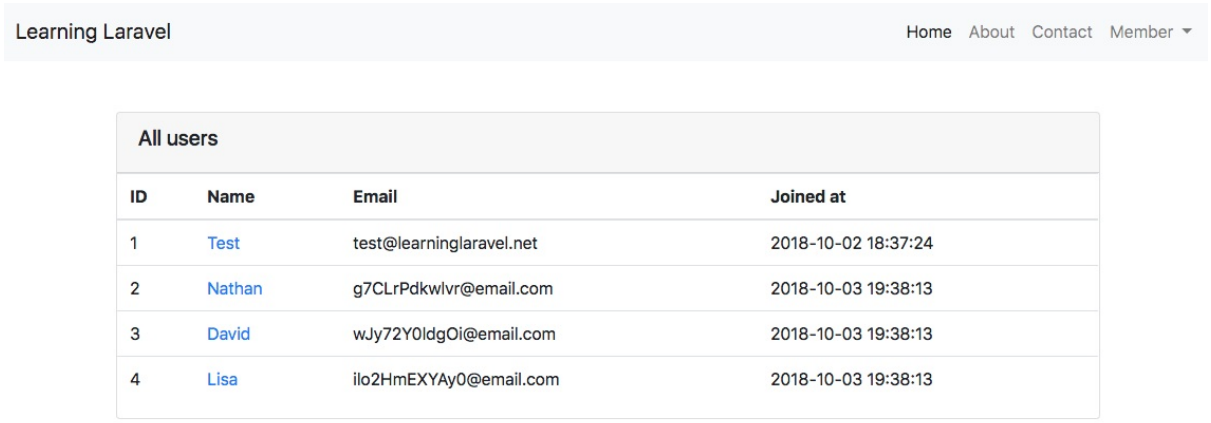
To seed data, run this Artisan command:

```
php artisan db:seed
```

This will execute the **run** method of the **UserTableSeed** class and insert data into our database.




Now check your database or visit http://homestead.test/admin/users, there are some 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 Reset Language Lines
    |-----------------------------------------------------------------------
    |
    | The following language lines are the default lines which match reasons
    | that are given by the password broker for a password update attempt
    | has failed, such as for an invalid token or invalid new password.
    |
    */

    'password' => 'Passwords must be at least six characters and match the confirmatio
n.',
    'reset' => 'Your password has been reset!',
    'sent' => 'We have e-mailed your password reset link!',
    'token' => 'This password reset token is invalid.',
    'user' => "We can't find a user with that e-mail address.",

];
```

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:

**resources/lang/en/main.php**

```
<?php

return [
```

```
    'subtitle' => 'Fastest way to learn Laravel',

];
```

Next, open the **home** view and find:

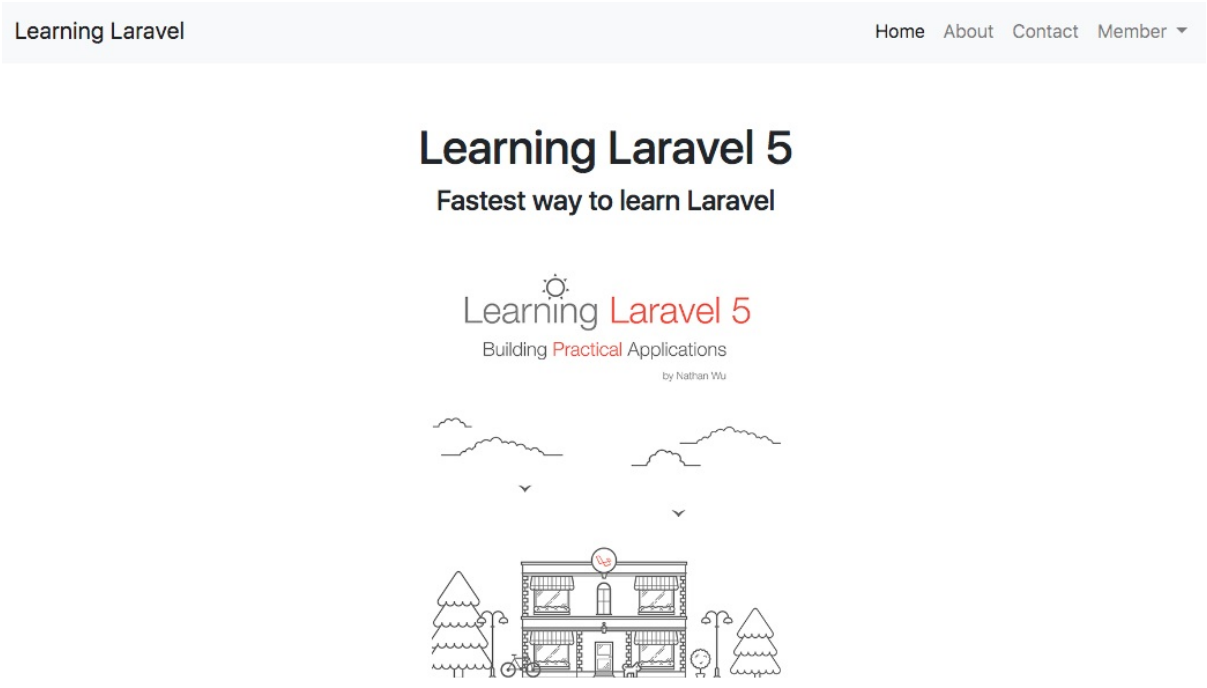
**views/home.blade.php**

```
<h3 class="text-center margin-top-100 editContent">Building Practical Applications</h3>
```

Change to:

```
<h4 class="text-center margin-top-100 editContent">{{ trans('main.subtitle') }}</h4>
```

Head over to your browser:




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:

https://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.
- You now understand more about routes and route group.
- Using Middleware, you can handle requests/responses effectively.
- Many Laravel applications are using **laravel-permission**, 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 are 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, Angular Book book and Vue.js Book out!

> **Note:** If you would like to give feedback, report bugs, translate the book, or ask any questions, please email us at support@learninglaravel.net. Thank you.

# Chapter 4 Source Code

You can view and download the source code at:

> Learning Laravel 5 Book: Chapter 4 Source Code

# 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, which 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]
```

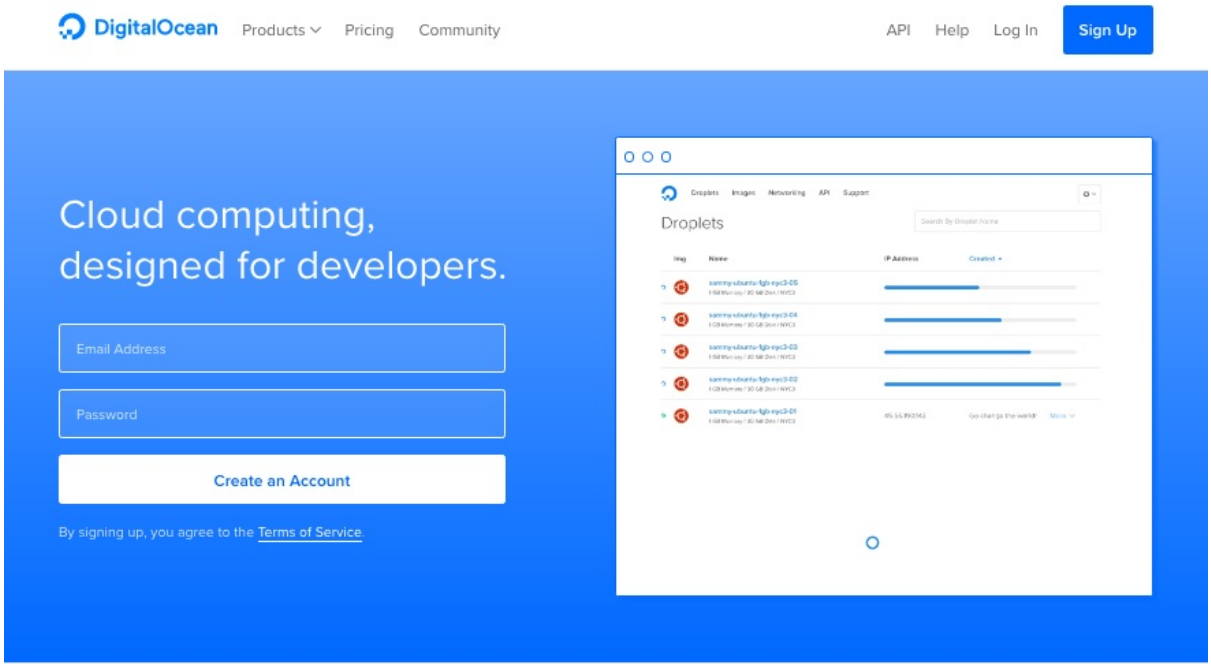
Well done, now it's time to visit your **website URL**! (www.yourdomain.com). You should see the Laravel welcome screen.

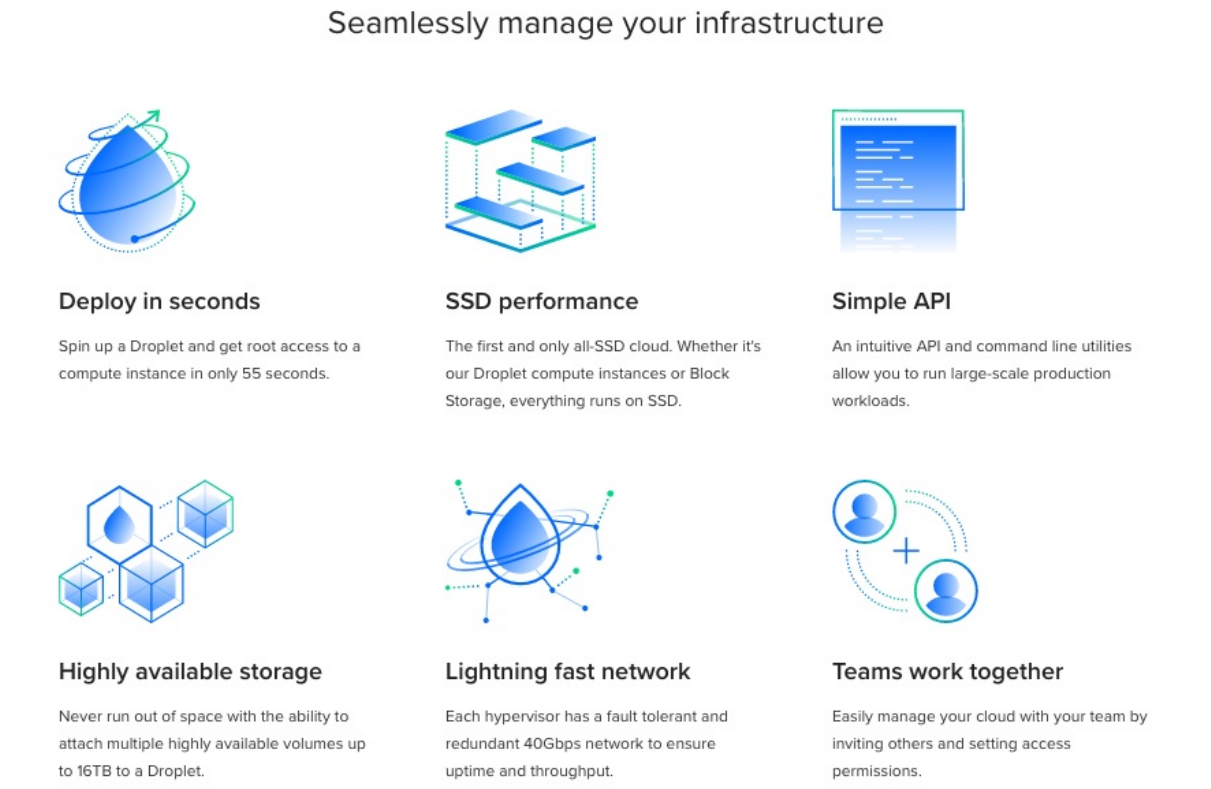

You've just deployed your Laravel app!

> **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 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 16.04.1 LTS VPS.** (Ubuntu 16.04.1 Long Term Support Virtual Private Server)

Why do I choose **Ubuntu 16.04.1**? There are some newer versions of Ubuntu, but the 16.04.1 is an LTS version, which 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 or more.

## Deploy a new Ubuntu server

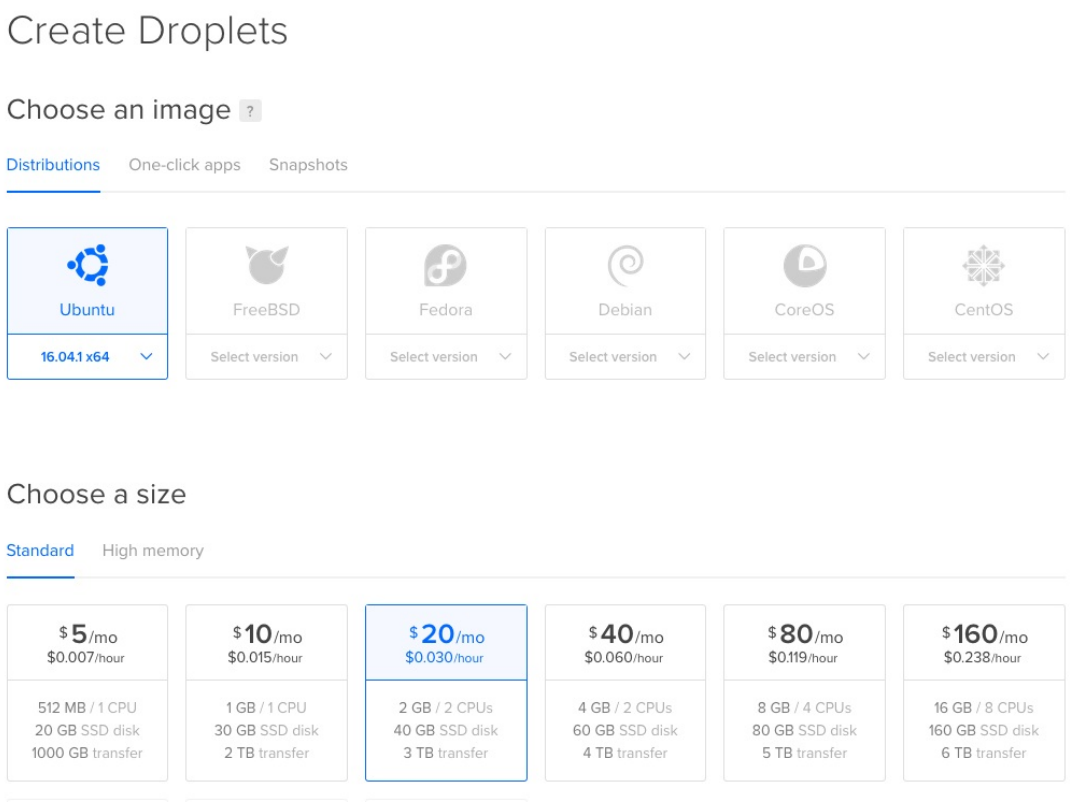
First, you will need to register a new account at DigitalOcean. You can use this link to get **$10** for free, that means you can use their **$5 cloud server** for **two months**.

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




> **Note:** The layout could be different.

Follow these steps:

- At the **Choose an Image** section, be sure to choose **Ubuntu 16.04.1 x64**.
- Select your **droplet size** and region that you like. **$5/mo** or **$10/mo** is fine.
- You may skip other settings.
- Give your Droplet **a name** (For example, **learninglaravel**).
- 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.

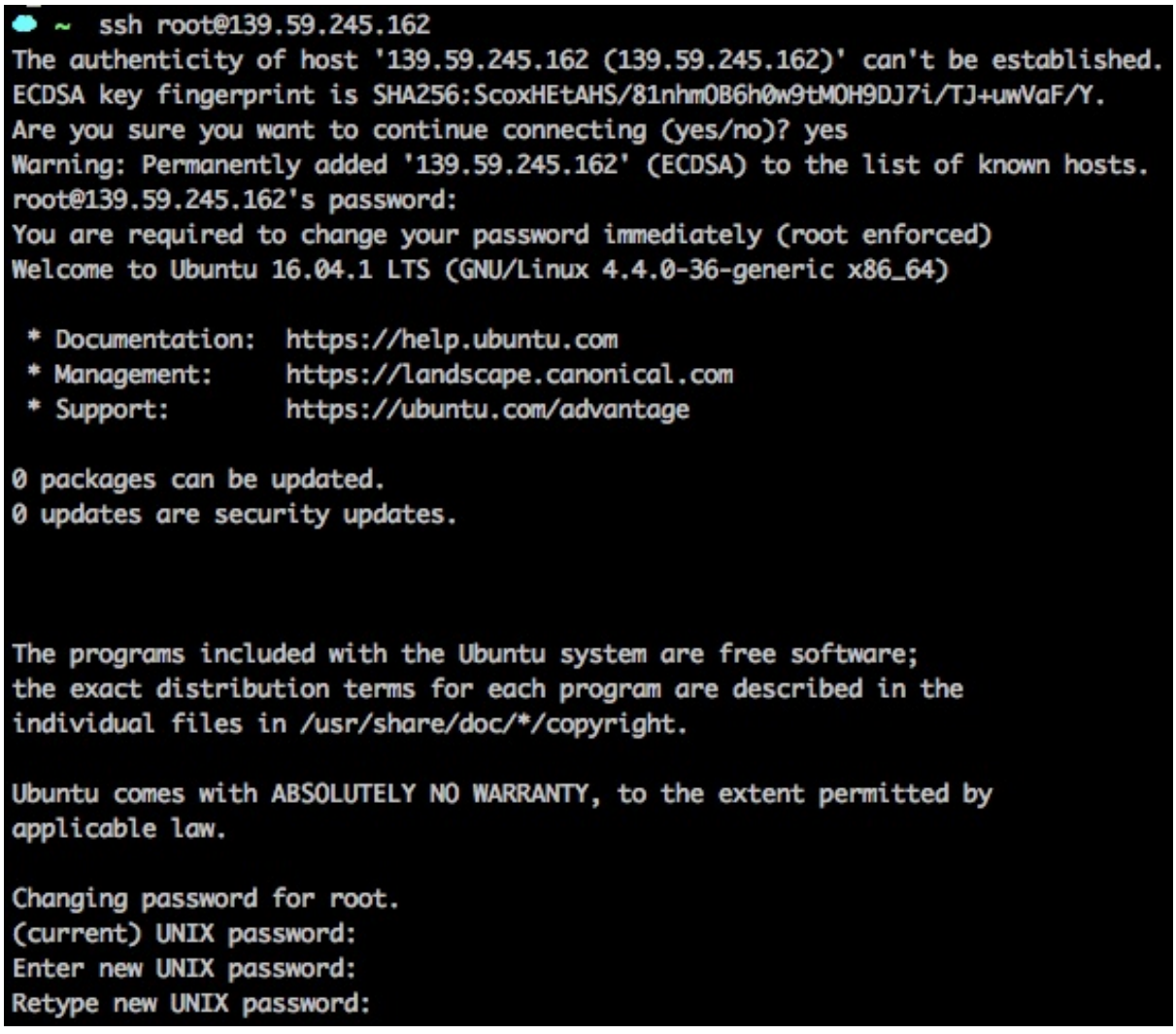
```
Droplet Name: learninglaravel
IP Address: 139.59.245.162
Username: root
Password: yourPassword
```

Great! Now you can access the new server via **Terminal** or **Git Bash** by using this command:

```
ssh root@139.59.245.162
```

> **Note:** Your IP Address should be different

Type **yes** if it asks if you want to continue connecting.




The first time you log in, 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 PHP 7, Nginx and other packages!

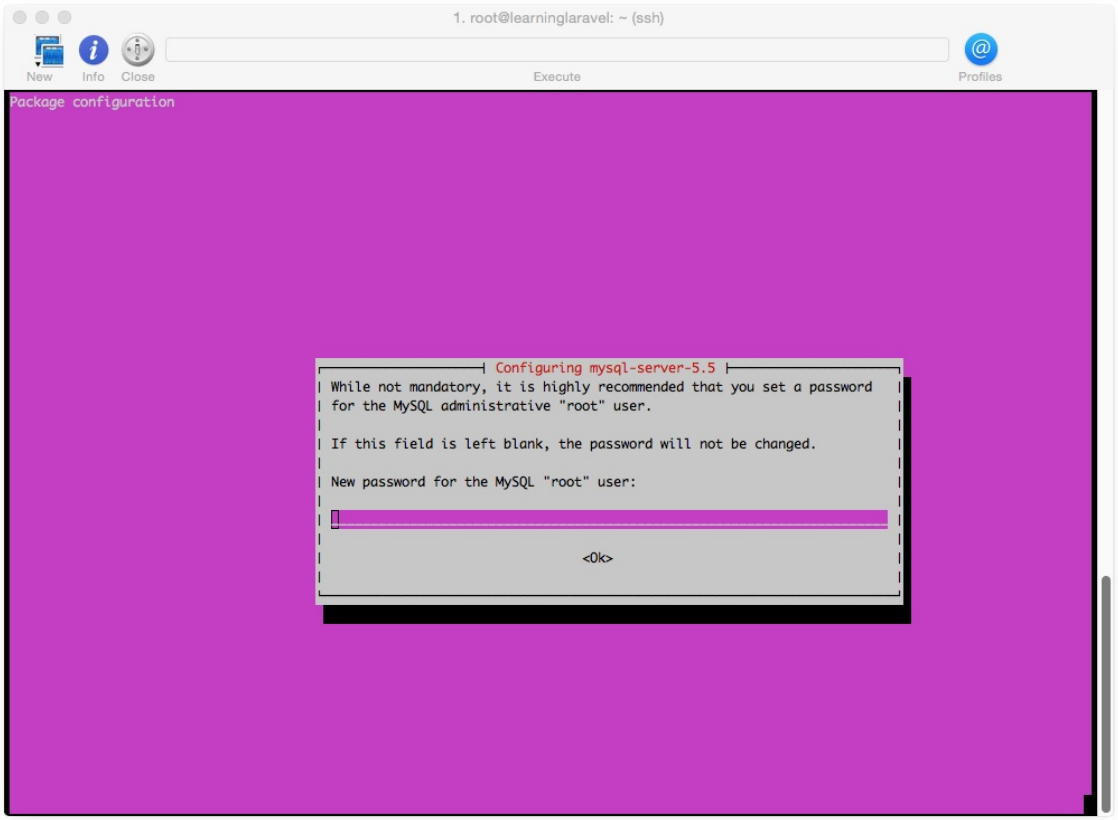
## Install MySQL Server

> **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.




Good job! You've installed MySQL. We will install PHP in the next section.

## Install Nginx, PHP and other packages

First of all, we need to add **Ondrej's PPA** to the **system's Apt sources** by running this command:

```
sudo add-apt-repository ppa:ondrej/php
```

> **Note:** A PPA (Personal Package Archive) is an Apt repository hosted on Launchpad. Third-party developers can distribute their custom PPA packages for Ubuntu outside of the official channels. We have to add the Ondrej's PPA because it **supports PHP 7.0 for Ubuntu**.

Press **Enter (or Return)** to continue if it asks you anything.

Run this command again to update our local packages:

```
sudo apt-get update
```

Next, run this command to install **Nginx, PHP 7, PHP7.0-FPM, PHP-MySQL, PHP7.0-Zip, Curl, phpredis, xdebug** and other useful packages.

```
apt-get -y install nginx php7.0 php7.0-fpm php7.0-mysql php7.0-curl php7.0-xml git php
7.0-zip php-redis php-xdebug php7.0-mcrypt
php-mbstring php7.0-mbstring php-gettext php7.0-gd
```

Alternatively, you may use this command to install more packages:

```
apt-get -y install nginx php7.0-fpm php7.0-cli php7.0-common php7.0-json php7.0-opcach
e php7.0-mysql php7.0-phpdbg
php7.0-gd php7.0-imap php7.0-ldap php7.0-pgsql php7.0-pspell php7.0-recode php7.0-tidy
 php7.0-dev php7.0-intl php7.0-gd
php7.0-curl php7.0-zip php7.0-xml git php-redis php-xdebug php7.0-mcrypt php-mbstring
php7.0-mbstring php-gettext
```

You may use this command to see all the PHP 7 packages:

```
sudo apt-cache search php7-*
```

Available packages:

```
php-radius - radius client library for PHP
php-http - PECL HTTP module for PHP Extended HTTP Support
```

```
php-uploadprogress - file upload progress tracking extension for PHP
php-mongodb - MongoDB driver for PHP
php7.0-common - documentation, examples and common module for PHP
libapache2-mod-php7.0 - server-side, HTML-embedded scripting language (Apache 2 module
)
php7.0-cgi - server-side, HTML-embedded scripting language (CGI binary)
php7.0-cli - command-line interpreter for the PHP scripting language
php7.0-phpdbg - server-side, HTML-embedded scripting language (PHPDBG binary)
php7.0-fpm - server-side, HTML-embedded scripting language (FPM-CGI binary)
libphp7.0-embed - HTML-embedded scripting language (Embedded SAPI library)
php7.0-dev - Files for PHP7.0 module development
php7.0-curl - CURL module for PHP
php7.0-enchant - Enchant module for PHP
php7.0-gd - GD module for PHP
php7.0-gmp - GMP module for PHP
php7.0-imap - IMAP module for PHP
php7.0-interbase - Interbase module for PHP
php7.0-intl - Internationalisation module for PHP
php7.0-ldap - LDAP module for PHP
php7.0-mcrypt - libmcrypt module for PHP
php7.0-readline - readline module for PHP
php7.0-odbc - ODBC module for PHP
php7.0-pgsql - PostgreSQL module for PHP
php7.0-pspell - pspell module for PHP
php7.0-recode - recode module for PHP
php7.0-snmp - SNMP module for PHP
php7.0-tidy - tidy module for PHP
php7.0-xmlrpc - XMLRPC-EPI module for PHP
php7.0-xsl - XSL module for PHP (dummy)
php7.0 - server-side, HTML-embedded scripting language (metapackage)
php7.0-json - JSON module for PHP
php-all-dev - package depending on all supported PHP development packages
php7.0-sybase - Sybase module for PHP
php7.0-sqlite3 - SQLite3 module for PHP
php7.0-mysql - MySQL module for PHP
php7.0-opcache - Zend OpCache module for PHP
php-apcu - APC User Cache for PHP
php-xdebug - Xdebug Module for PHP
php-imagick - Provides a wrapper to the ImageMagick library
php-ssh2 - Bindings for the libssh2 library
php-redis - PHP extension for interfacing with Redis
php-memcached - memcached extension module for PHP5, uses libmemcached
php-apcu-bc - APCu Backwards Compatibility Module
php-amqp - AMQP extension for PHP
php7.0-bz2 - bzip2 module for PHP
php-rrd - PHP bindings to rrd tool system
php-uuid - PHP UUID extension
php-memcache - memcache extension module for PHP5
php-gmagick - Provides a wrapper to the GraphicsMagick library
php-smbclient - PHP wrapper for libsmbclient
php-zmq - ZeroMQ messaging bindings for PHP
php-igbinary - igbinary PHP serializer
php-msgpack - PHP extension for interfacing with MessagePack
```

```
php-geoip - GeoIP module for PHP
php7.0-bcmath - Bcmath module for PHP
php7.0-mbstring - MBSTRING module for PHP
php7.0-soap - SOAP module for PHP
php7.0-xml - DOM, SimpleXML, WDDX, XML, and XSL module for PHP
php7.0-zip - Zip module for PHP
php-tideways - Tideways PHP Profiler Extension
php-yac - YAC (Yet Another Cache) for PHP
php-mailparse - Email message manipulation for PHP
php-oauth - OAuth 1.0 consumer and provider extension
php-propro - propro module for PHP
php-raphf - raphf module for PHP
php-solr - PHP extension for communicating with Apache Solr server
php-stomp - Streaming Text Oriented Messaging Protocol (STOMP) client module for PHP
php-gearman - PHP wrapper to libgearman
```

> **Note:** You may remove some packages that you don't use and install them later when you need.

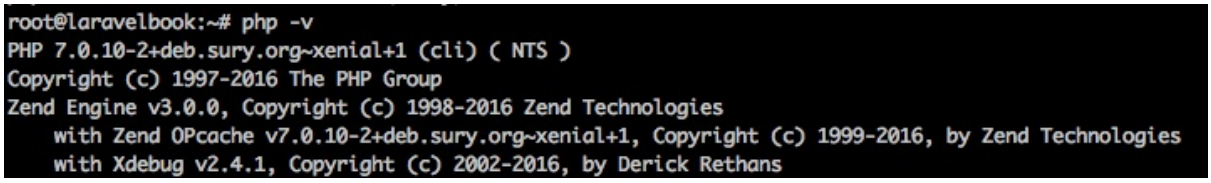
Done! You can now visit your Nginx server via the **IP address**:

http://139.59.245.162

> **Note:** Your IP address must be different.


We can check the installed version of PHP by using this command:

```
php -v
```




After that, we need to edit the **server block** (aka **virtual hosts**) file. Open it:

```
sudo nano /etc/nginx/sites-available/default
```

Find:

```
root /var/www/html;
```

This is the path to your Laravel application, we don't have a Laravel application yet, but let's change it to:

```
root /var/www/learninglaravel.net/html;
```

> **Note:** You may use a different address (change **learninglaravel.net** to your **website's address**) if you want. Be sure to replace all the addresses.

Find:

```
index index.html index.htm index.nginx-debian.html;
```

Change to:

```
index index.php index.html index.htm index.nginx-debian.html;
```

Find:

```
    location / {
            # First attempt to serve request as file, then
            # as directory, then fall back to displaying a 404.
            try_files $uri $uri/ =404;
    }
```

Change to:

```
    location / {
            # First attempt to serve request as file, then
            # as directory, then fall back to displaying a 404.
            try_files $uri/ $uri /index.php?$query_string;
    }
```

Add below:

```
location ~ \.php$ {
        try_files $uri /index.php =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass unix:/var/run/php/php7.0-fpm.sock;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include fastcgi_params;
}
```

**Save the file** and **exit**.

In **nano**, you can do this by pressing **Ctrl-X** to exit, then **press y to confirm**, and **Enter** to overwrite the file.

As you know, 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
```

Next, let's make a test file called **index.html** to test our configurations:

```
sudo nano /var/www/learninglaravel.net/html/index.html
```

Here is the content of the **index.html** file:

```
<html>
<head>
<title>Learning Laravel</title>
</head>
<body>
<h1>Learning Laravel test page. PHP 7 and Nginx</h1>
</body>
</html>
```

Finally, restart PHP and Nginx by running the following:

```
service php7.0-fpm restart
service nginx restart
```

Now when you visit your website via its IP address, you should see:


Well done! You now have a working PHP 7 installation.

## Install Laravel

Now that we have everything in order, we will be going to install **Composer** and use it to **install Laravel**!

If you're **installing Laravel on a 512MB droplet**, you must add a swapfile to Ubuntu to prevent it from running out of memory. You can add a swapfile easily by running these commands:

```
dd if=/dev/zero of=/swapfile bs=1024 count=512k
mkswap /swapfile
swapon /swapfile
```

> **Note:** If your server is restarted, you have to add the swapfile again.

Run this simple command to **install Composer**:

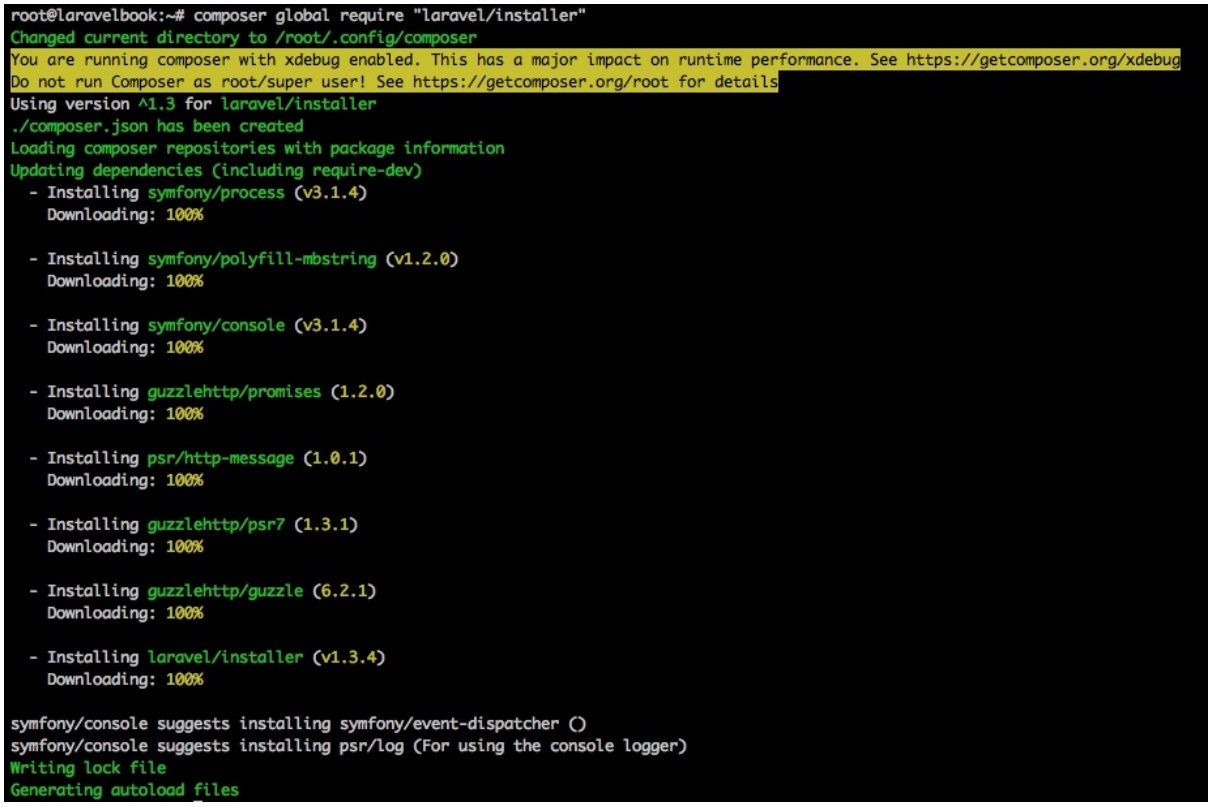
```
curl -sS https://getcomposer.org/installer | php
```

Once installed, run this command to **move composer.phar to a directory that is in your path**, so that you can **access it globally**:

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

Next, we will use **Composer** to download the Laravel Installer:

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



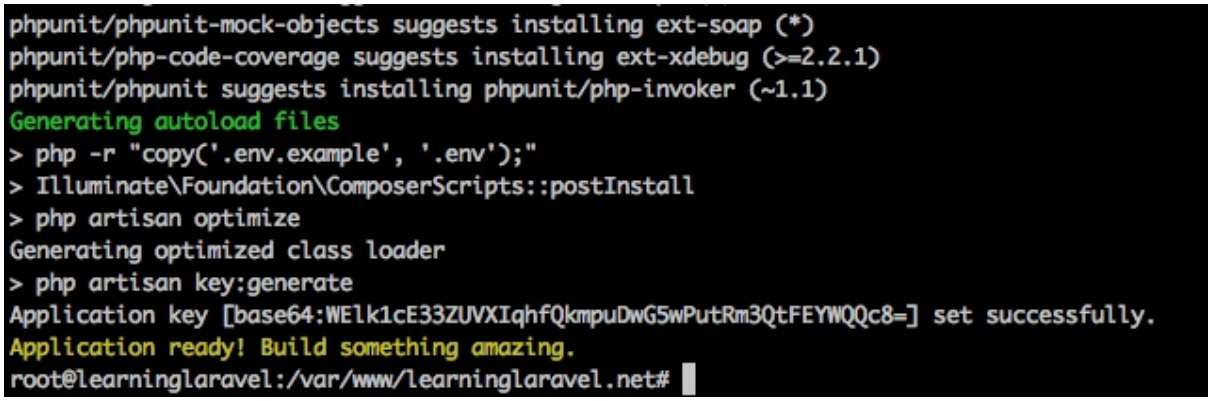

Once finished, we're finally at the part that we've been waiting for: **Installing Laravel**!

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. Run this command:

```
laravel new laravel
```



If you see this error when using the **laravel new** command:

```
laravel: command not found
```

We have to edit the **.bashrc** file, type:

```
nano ~/.bashrc
```

Add this line at end of the file:

```
alias laravel='~/.config/composer/vendor/bin/laravel'
```

Press **Ctrl + X**, then **Y**, then **Enter** to **exit and save** the file.

Lastly, run this command:

```
source ~/.bashrc
```

Now you should be able to create a new Laravel app using:

```
laravel new laravel
```

> **Note:** This is a pretty standard process. I hope you understand what we've done. If you don't, please read **chapter 1** again.

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;
```

**Save** the file and **exit**.

Finally, restart Nginx:

```
service nginx restart
```

Go ahead and visit your Laravel app in browser:


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 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
```

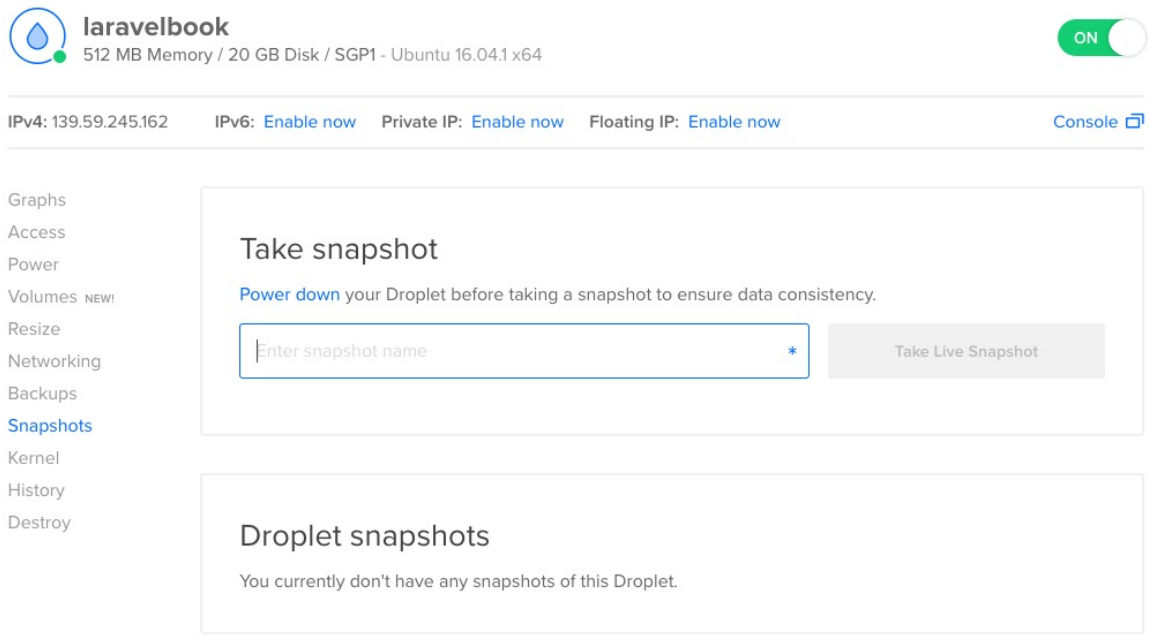
## Take a snapshot of your application

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.




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, Laravel Angular book and Vue.js Book out!

Thank you again for your support! Have a great time!