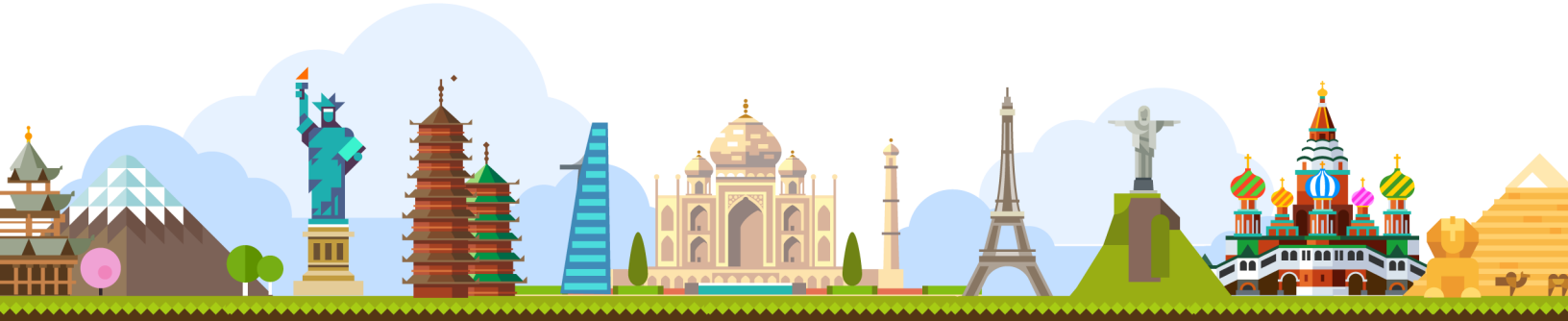
LARAVEL 5 COOKBOOK

ENHANCE YOUR AMAZING APPLICATIONS

by Nathan Wu



Laravel 5 Cookbook

Enhance Your Amazing Applications

Nathan Wu

Contents

Book Information

Book Description

If you're looking for a book that can help you to build amazing web applications, this is the book for you! Aimed at people who have some experience with Laravel, this cookbook has your back!

There are many proven code rich recipes for working with Laravel. Each recipe includes practical advice, tips and tricks for working with jQuery, AJAX, JSON, API, data persistence, complex application structure, modular PHP, testing, deployment and more.

Think about this book as a collection of all premium Laravel tutorials or the successor to the popular Learning Laravel 5 book.

Laravel 5 Cookbook also includes tested code that you can download and reuse in your own applications. You'll save time, learn more about Laravel and other related technologies in the process.

We also have a forum for discussion and debate. You can freely ask any questions, provide your valuable feedback and help others.

It's time to discover Laravel more!

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 and Laravel are not covered, you will need to:

- Read Learning Laravel 5 book. (optional)
- Have a basic knowledge of PHP, HTML, CSS and Laravel.
- Love Laravel.

What You Will Get

- Lifetime access to the online book. (Premium Only)
- 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 20000+ students learning together.
- Amazing bundles and freebies to help you become a successful developer.
- iPhone, iPad and Android Accessibility.

Book Structure

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

Chapter 1 - Back End Recipes

Building APIs and large applications using modern technologies can be a daunting task. In this chapter, you'll learn best practices and modern techniques for back-end development, starting with an introduction to the command line and Git.

These complete, easy-to-use recipes show you how to use cookies, sessions, web storage and some popular Laravel packages. You'll also learn about writing APIs and debugging techniques.

In addition to mastering the technologies, you'll understand when they're needed and how to use them.

Chapter 2 - Front End Recipes

Whether you are a beginner or intermediate web developer, if you wish to make good interactive web applications, then this chapter is for you.

In this chapter, you'll be getting some recipes about front-end web technologies and popular front-end tools. These recipes cover best practices and modern techniques for front-end development such as: integrating Twitter Bootstrap, AJAX loading, notifications, cropping images, file uploads and many more.

By the end, you should have a better understanding of how to work with AJAX, Jquery, front end frameworks and responsive design. You can apply these techniques to build beautiful applications and add that interactivity to any site you work on.

Chapter 3 - Deployment Recipes

After learning some tricky topics to successfully build a full stack application, it's time to deploy your app. This chapter contains some helpful recipes about working with Heroku, Digital Ocean, etc.

Deploy your applications blazingly fast using GIT and secret techniques are also discussed in the book!

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[1]

Book Status, Changelog and Contributors

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

Book Status[2]

Changelog[3]

Contributors[4]

[1] http://learninglaravel.net/books/laravelcookbook/cookbook-translators
[2] http://learninglaravel.net/books/laravelcookbook/cookbook-status
[3] http://learninglaravel.net/books/laravelcookbook/cookbook-changelog
[4] http://learninglaravel.net/books/laravelcookbook/cookbook-contributors

Changelog

Current Version

Latest version the book:

- Version: 0.20
- Status: Complete (Beta Version)
- Updated: May 15th, 2016

Laravel 5 Cookbook

Chapter 1: Back End Recipes

Introduction

Building APIs and large applications using modern technologies can be a daunting task. In this chapter, you'll learn best practices and modern techniques for back-end development, starting with an introduction to the command line and Git.

These complete, easy-to-use recipes show you how to use cookies, sessions, web storage and some popular Laravel packages. You'll also learn about writing APIs and debugging techniques.

In addition to mastering the technologies, you'll understand when they're needed and how to use them.

Project Files

All project files of this book can be downloaded at:

https://github.com/LearningLaravel/cookbook/releases[5]

At the end of each recipe, you can find the recipe's project files (**Tag**). Feel free to use each of them at any stage of your development process.

List Of Recipes

> **Note:** As this is a cookbook, you may skip any recipe that you know already. The book is still under active development, that means some chapters and its recipes may change. The book also may have some errors and bugs. For any feedback, please send us an email.

Backend recipes

- Recipe 1 - Introducing CLI (Command Line Interface)
- Recipe 2 - All About Git
- Recipe 3 - Build A Laravel Starter App
- Recipe 4 - Create A User Authentication System with Facebook and Socialite

[5] https://github.com/LearningLaravel/cookbook/releases

- Recipe 5 - Create A User Authentication System Using Laravel Auth Scaffold
- Recipe 6 - Image Upload In Laravel
- Recipe 7 - Seeding Your App Using Faker
- Recipe 8 - Pagination
- Recipe 9 - Testing Your App
- Recipe 10 - Writing APIs with Laravel
- (More recipes will be added later)

Recipe 1 - Introducing CLI (Command Line Interface)

Laravel 5 Cookbook contains many recipes to create interactive web applications. These recipes are premium tutorials for web developers of all skill levels. For most of the recipes in this book, you will need to use Git to install sample code and Homestead to execute your code. If you don't have Homestead installed yet, you can follow these instructions to install it:

http://learninglaravel.net/laravel5/installing-laravel

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

What will we learn?

This recipe shows you how to use the command line on PC and Mac.

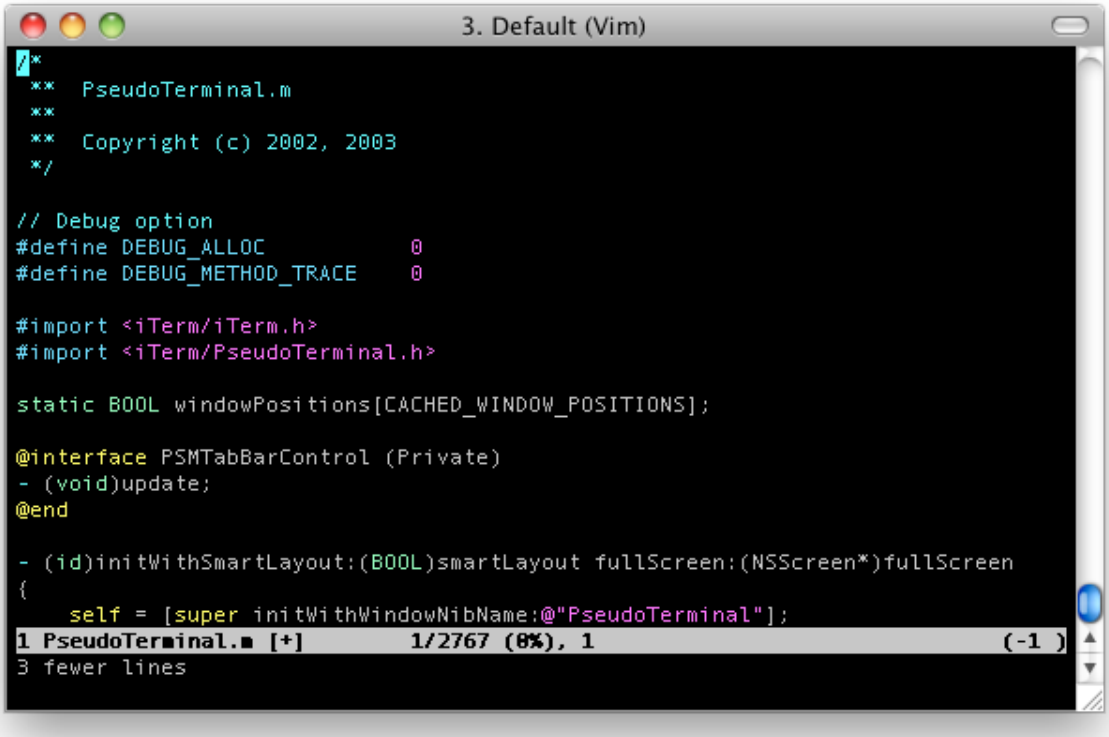
CLI for MAC OSX

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[6].

[6] https://www.iterm2.com



Iterm inteface

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[7]

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

CLI for Linux

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

[7] http://msysgit.github.io

Recipe 2 - All About Git

There's a chance you may already know about Git! Today, most programmers prefer Git over other distributed version control systems.

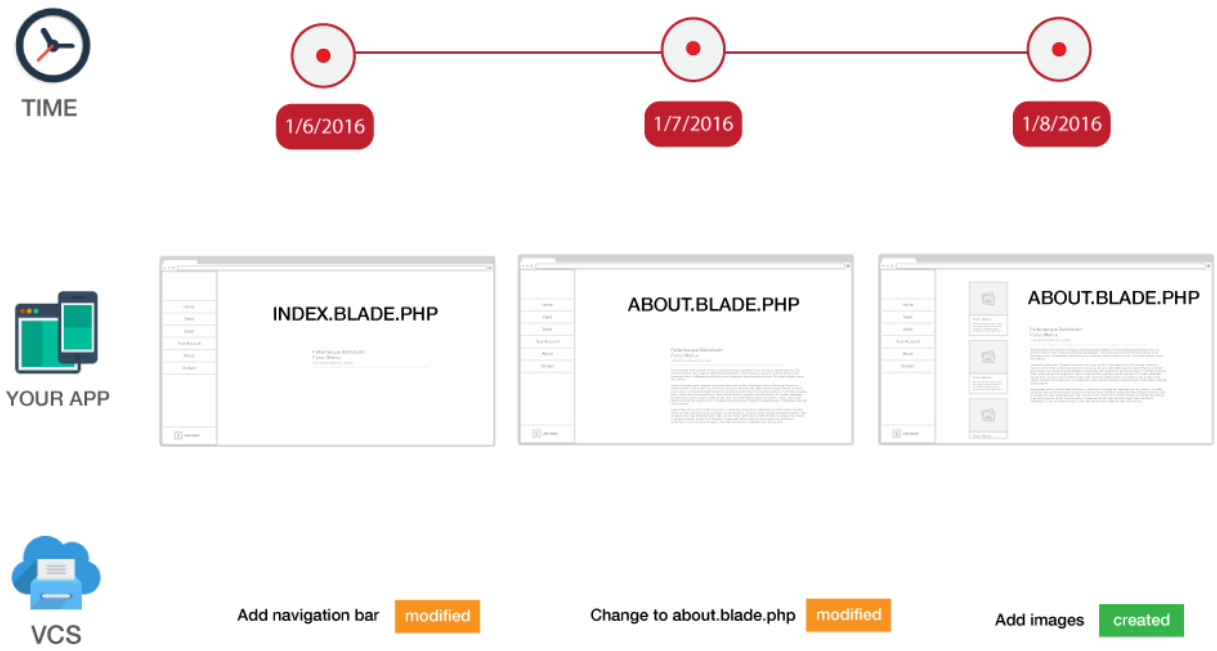
What will we learn?

This recipe introduces Git and provides a list of some important Git commands to get you going with Git.

What is Version Control?

Version Control System (VCS) let you store different versions of your projects and all its files. You can roll back to an earlier version or take a look at an older snapshot to see which files have been changed.

Here is a nice infographic about Version Controler System:



Iterm inteface

Why do you need to use Git?

Git is becoming an 'industry standard'. If you want to become a better developer, you may need to use Git to develop software and collaborate with other developers. Git lets you manage code development in a virtually endless variety of way. Here are benefits of using Version Control System/Git:

- Git allows you to create as many branches of your project as you want. You can use each branch to test, create a new feature, fix bugs, etc.
- You can see what was changed in your project's files. This helps you understand what happened and improve your code.
- You can easily store all the versions of your site and restore previous versions whenever you want.
- Store your files on cloud-based Git repository host services like Github and Bitbucket.
- You can easily share your files with others.
- A VCS or Git helps your team work more efficiently. Everyone knows what is going on and can merge the changes into a common version.

How to install Git?

> **Note**: if you don't know how to run a command, please read the **Recipe 1 - Introducing CLI (Command Line Interface)**.

Install Git on Mac

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

```
1 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[8]

Install Git on Windows

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

https://windows.github.com[9]

Install Git on Linux/Unix

You can install Git by running this command:

[8] http://git-scm.com/download/mac
[9] https://windows.github.com

```
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[10]

Configuring Git

When you first install Git, you should set your name, email address and enable coloring to pretty up command outputs. Open your CLI and run these commands:

```
git config --global user.name "Your Name"
git config --global user.email "Your Email Address"
git config --global color.ui auto
```

> Note: Remember to replace **Your Name** and **Your Email Address**.

Start versioning your project using Git

Git is very simple to use. First, you need to go to your working directory:

```
cd Code/Laravel
```

> Note: If you're using Homestead, the **Code** directory is where we will put our Laravel apps. **Code/Laravel** is your working directory. You can use Git on Homestead or on your local machine, it's up to you.

Now we can use the **git init** command to initialize Git:

```
git init
```

This command creates **an empty Git repository**. If you're using Homestead, the path of the Git directory is:

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

```
/home/vagrant/Code/Laravel/.git/
```

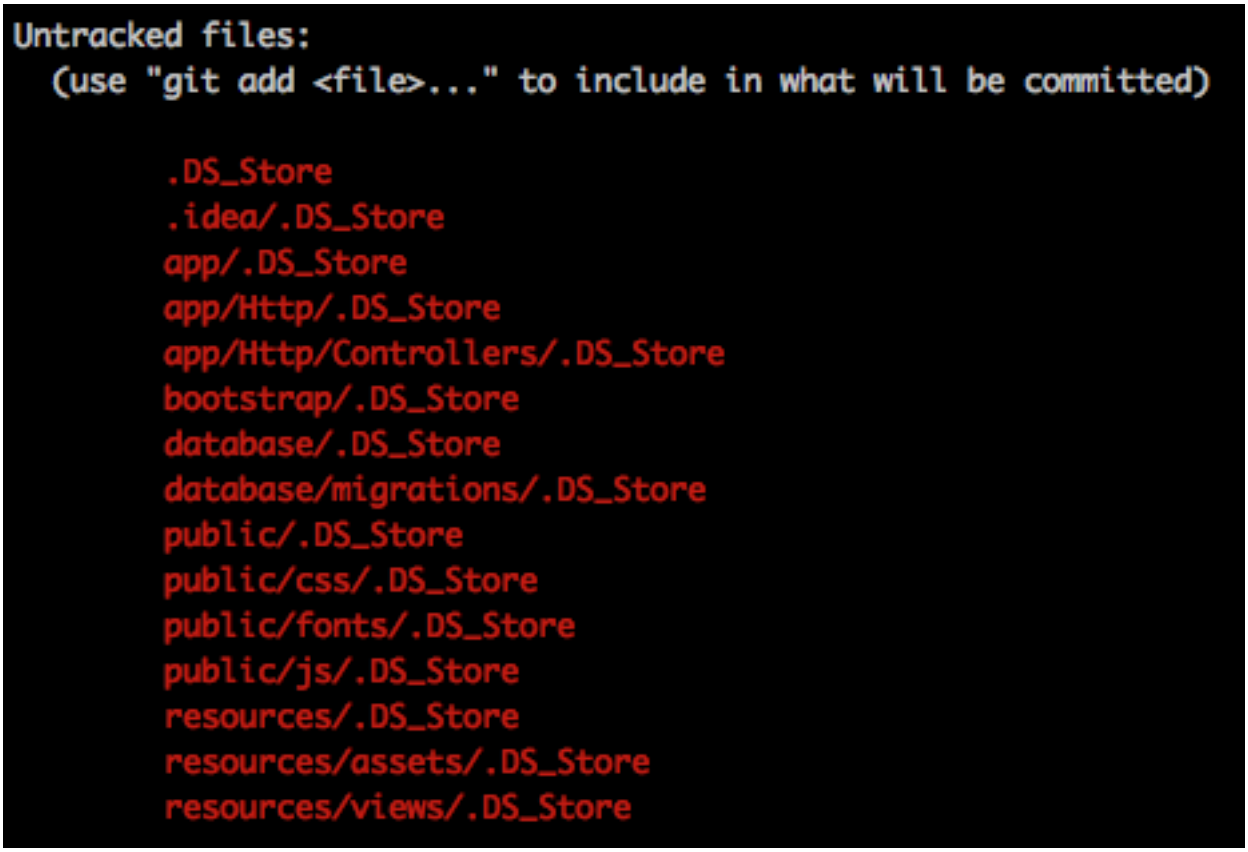
"**.git**" is a hidden folder and it doesn't contain your project's files yet.

Add and commit your files

Now we can use **git status** command to check the status of our working directory:

```
git status
```

You will see a list of **untracked files**, that means Git doesn't monitor those files yet.



Untracked files

To tell Git that you want to include all these files, use the **git add -A** command:

```
git add -A
```

> **Note**: Alternatively, you can use **git add –a** or **git add –all** or **git add .** command.

When we run the **git status** command again, you'll see:

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .DS_Store
        new file:   .idea/.DS_Store
        modified:   .idea/blade.xml
        modified:   .idea/encodings.xml
        deleted:    .idea/misc.xml
        modified:   .idea/vcs.xml
        modified:   .idea/workspace.xml
        new file:   app/.DS_Store
        modified:   app/Category.php
        new file:   app/Http/.DS_Store
        new file:   app/Http/Controllers/.DS_Store
        modified:   app/Http/Middleware/Manager.php
        new file:   bootstrap/.DS_Store
        modified:   composer.lock
        new file:   database/.DS_Store
        new file:   database/migrations/.DS_Store
        new file:   public/.DS_Store
        new file:   public/css/.DS_Store
        new file:   public/fonts/.DS_Store
        new file:   public/js/.DS_Store
        new file:   resources/.DS_Store
        new file:   resources/assets/.DS_Store
        new file:   resources/views/.DS_Store
        modified:   resources/views/auth/register.blade.php
        modified:   resources/views/backend/categories/create.blade.php
        modified:   resources/views/backend/home.blade.php
```

Add files

The **git add** command tells git to add changes in your project to the staging area. However, those changes aren't saved yet until you run **git commit**:

```
1  git commit -m "First commit"
```

You can use the **-m** flag (stands for **message**) to give a comment on the command line. My message is "**First commit**", but you can use whatever you like.

Well done! You've made your first commit!

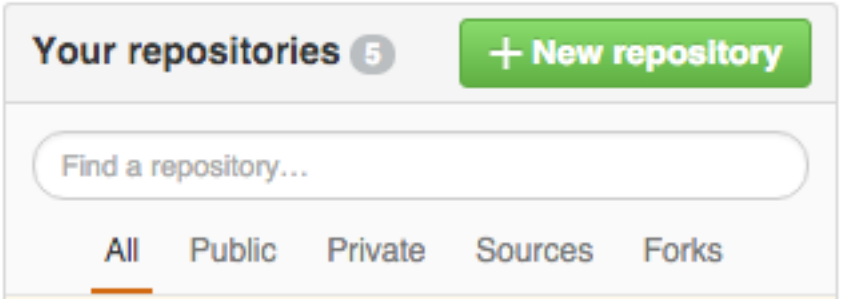
Store your files on Git repository host services

You can store all your files on cloud-based Git repository host services and access them anywhere, anytime.

The two most popular services are Github[11] and Bitbucket[12].

We'll use Github in this book, but feel free to use what you like.

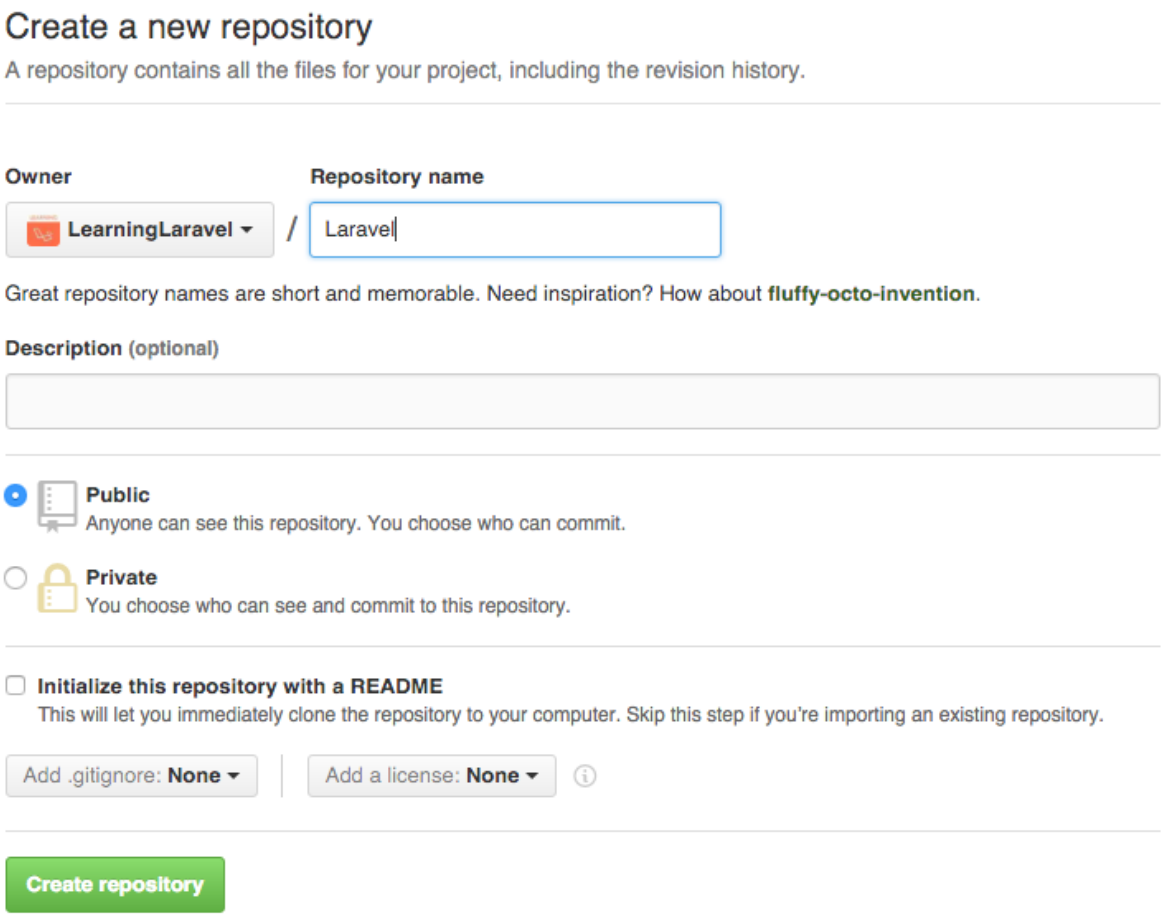
Let's register an account on Github if you don't have one yet. After that, click the **New repository** button to create a new repository.



New repository

This repository contains all your project's files.

[11] http://github.com
[12] http://bitbucket.org



Creata a new repository

When creating a new repository, you can choose any name that you like. Choose **Private** if you don't want anyone access your files.

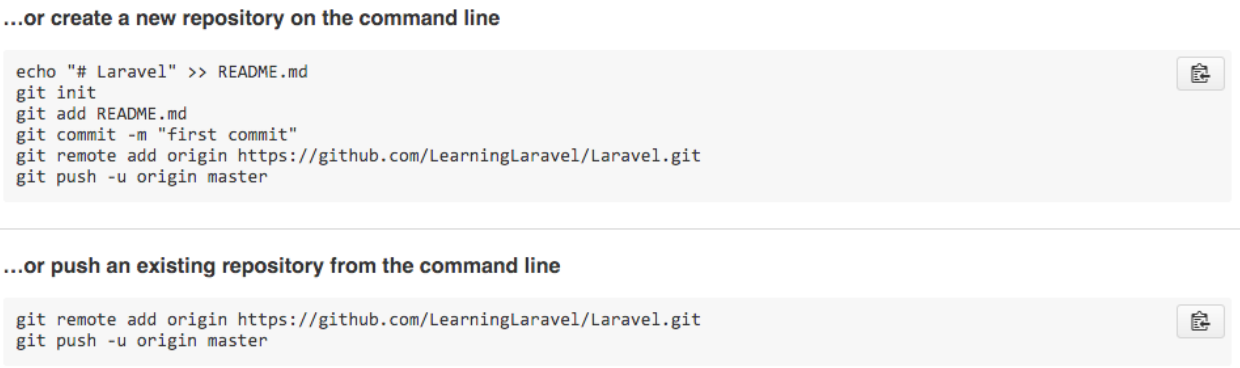
> **Note:** Don't worry too much about the settings, you can change those settings later.

Click **Create repository** to confirm.

Great! You now have a new Github repository!

Push your project to Github

You should see Github's quick setup guide:



Creata a new repository

Your new repository (repo) is empty. You need to upload your files to that Github repo.

Every repository has a unique remote URL, your remote URL should look like this:

https://github.com/YourGithubUsername/YourRepoName.git

Take note of this link.

Good! We will try to upload our **Laravel app** (/Code/Laravel) to Github.

Navigate to the working directory (on your Homestead or local machine):

```
cd /Code/Laravel
```

Add a new remote by using **git remote add** command:

```
git remote add origin https://github.com/LearningLaravel/Laravel.git
```

origin is the **remote name**, **https://github.com/LearningLaravel/Laravel.git** is the **remote URL**.

> **Note**: Your remote URL should be different. Be sure to use your remote URL.

After adding a new remote, we can push our files to Github using the **git push** command:

```
git push -u origin master
```

If it asks for a password, enter your Github password.

origin is the remote that we've just added. **master** is your working directory.

When we use the **-u** (stands for **upstream**) flag, we add the **upstream reference**. If we successfully push our files to the repo, Git will remember it, so we don't have to type **origin master** next time. That means, if we want to upload our files again, we can just type:

```
git push
```

You now have your files on the cloud!

Cloning a repository

To download any repo, you can use the **git clone** command.

First, navigate to the location where you want to place the cloned directory:

```
cd Code
```

Type **git clone** and the **unique remote URL** to clone the repo:

```
git clone https://github.com/YourGithubUsername/YourRepoName.git
```

This command creates a local clone of the repository on your computer.

> Note: you can clone any **public repository**. If you don't want anyone to download your repo, set it **private**.

Recipe 2 Wrap-up

In this recipe, you learned some major Git commands. Throughout this book, we'll use Git to download the source code, front end components and deploy our Laravel applications. We won't talk about it anymore because this is a Laravel book, not a Git book. If you wish to learn more about Git, check these sites out:

Atlassian Git Tutorials[13]

Git-Tower[14]

Codeschool - Try Git[15]

Super Useful Need To Know Git Commands[16]

It's time to start learning about Laravel!

[13] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
[14] https://www.git-tower.com/learn/git/ebook
[15] https://www.codeschool.com/courses/try-git
[16] http://zackperdue.com/tutorials/super-useful-need-to-know-git-commands

Recipe 3 - Build A Laravel Starter App

What will we learn?

We will build a simple app and use it as a template for our next recipes.

Throughout this process, we will learn how to run multiple Laravel sites on Homestead and how to integrate Twitter Bootstrap into our apps.

Installing Laravel

Let's start by installing Laravel!

> **Note**: Please note that I'm using Homestead. If you don't use Homestead, the process could be different.

First, **SSH** into our **Homestead**:

```
1  vagrant ssh
```

Then navigate to our **Code** directory.

```
1  cd Code
```

Be sure to have the latest version of **Laravel Installer** by running this command:

```
1  Composer global require "laravel/installer"
```

```
vagrant@homestead:~/Code$ composer global require "laravel/installer"
Changed current directory to /home/vagrant/.composer
Using version ^1.3 for laravel/installer
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Nothing to install or update
Generating autoload files
```

New repository

Now let's create a new **cookbook** app:

```
laravel new cookbook
```

Great! You should have a new Laravel app! Feel free to change the name of the app to your liking.

> **Note:** If you're not familiar with this process, please read the Learning Laravel 5 book.

Create multiple Laravel apps on Homestead

You can't access your new app because Homestead doesn't know about it yet. Therefore, let's follow these steps to activate your site:

> **Note:** Be sure to backup your current projects' files and databases.

First, we have to go to the **Homestead directory**:

```
cd ~/.homestead
```

And edit the **Homestead.yaml** file:

```
vim Homestead.yaml
```

We use VIM to edit the file. If you don't know how to use VI or VIM, you can open it with your favorite editor by using this command:

```
open Homestead.yaml
```

Find:

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

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

Our new app is called **cookbook**, and I would like to access it via this address: "**http://cookbook.app**". So, let's add the following code:

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

Save the file.

> **Tip**: if you're using **VIM**, press **ESC** and then write **:wq** (write quit) to save and exit

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 vim /private/etc/hosts
```

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

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

```
192.168.10.10  cookbook.app
```

> **Note**: All sites will be accessible by HTTP via port 8000 and HTTPS via port 44300 by default.

To let the system know that we've edited the hosts file, run this command:

```
source /private/etc/hosts
```

Finally, SSH into our Homestead (by using **vagrant ssh** or **homestead ssh**), and use the **serve** command to activate our new site:

```
serve cookbook.app /home/vagrant/Code/cookbook/public/
```

Good job! If everything is working correctly, we should see our app's home page:

Laravel 5

New home page

Creating Our Home Page

> **Note:** If you don't understand any step in this section, be sure to check out the Learning Laravel 5 book. If you don't want to follow along, you may skip these steps and download the sample app at the end of this recipe.

We will create a new home page for our app.

First, let's create a **home** view (**views/home.blade.php**) for our homepage:

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

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

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

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

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

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

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

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

After that, generate a new **PagesController**:

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

Open **PagesController**, which can be found at **app/Http/Controllers**, and create a new **home action**:

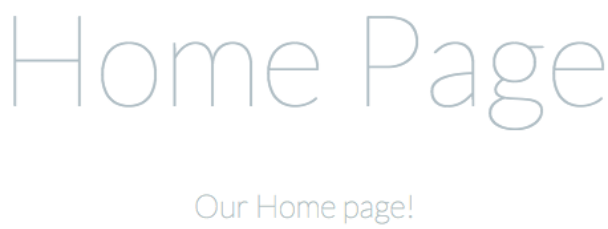
```
?php namespace App\Http\Controllers;

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

use Illuminate\Http\Request;

class PagesController extends Controller {

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

}
```

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

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

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

Great! We should now have a new home page!



New home page

Integrating Twitter Bootstrap

Nowadays, the most popular front-end framework is Twitter Bootstrap. It's free, open source and has a large active community.

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

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

You can get Bootstrap and read its official documentation here:

http://getbootstrap.com[17]

There are three ways to integrate Bootstrap:

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

[17] http://getbootstrap.com

In this book, we will use the first one (using Bootstrap CDN). This is also the fastest method.

Let's open **home.blade.php**, remove the **Lato font** and these **css styles**:

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

Place these links inside the head tag

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css\
/bootstrap.min.css">
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css\
/bootstrap-theme.min.css">

<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js\
"></script>
```

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

Create a master layout, app navigation bar and other pages

It's time to create a **master layout** (**master.blade.php**) for our app:

```
<html>
<head>
    <title> @yield('title') </title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6\
/css/bootstrap.min.css">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6\
/css/bootstrap-theme.min.css">

    <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.mi\
n.js"></script>
</head>
<body>

@include('shared.navbar')

@yield('content')

</body>
</html>
```

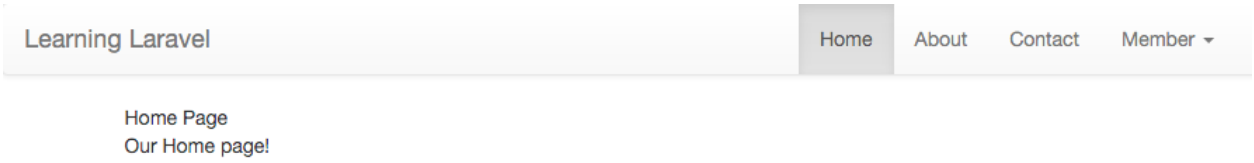
Next, create a new **navbar** view and place it at **shared/navbar.blade.php**:

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

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

We can now change our home view to extend the master layout.

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

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

Refresh your browser, we should have a new home page with a nice navigation bar:



New home page

We can then continue to create the about and contact page:

about view (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 view (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
```

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

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

Please note that we're using **Laravel 5.2**, so we need to add these routes into the **web middleware group**:

```
Route::group(['middleware' => ['web']], function () {
    Route::get('login/facebook', 'Auth\AuthController@redirectToFacebook');
    Route::get('login/facebook/callback', 'Auth\AuthController@getFacebookCallba\
ck');
});
```

Open **PagesController**, add:

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

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

Congratulations! You've just taken the first step in building awesome Laravel applications!

> **Note:** If you're using Git, this is a good time to initialize your repo with **git init**.

This will be our **starter template**.

Recipe 3 Wrap-up

> **Tag:** Version 0.1 - Recipe 3[18]

Good job! We've got our app running.

As you can see, our main app template is very simple and it isn't what we really want. Therefore, let's start adding more features into it!

Recipe 4 - Create A User Authentication System with Facebook and Socialite

What will we learn?

We will learn how to use Socialite - a new Laravel 5 feature - to let users log in using their Facebook account.

After learning this recipe, you may apply the technique to authenticate users with other social networks (Twitter, Github, Gmail, etc.) as well.

[18] https://github.com/LearningLaravel/cookbook/releases/tag/v0.1

Installing Socialite

In most web frameworks, authenticating users using 3rd party providers is never as easy as it could be. Luckily, Laravel 5 provides a simple way to authenticate with OAuth providers using Socialite.

Currently, Socialite officially supports authentication with **Facebook**, **GitHub**, **Google**, **Twitter and Bitbucket**. If you want to use Socialite with other providers (**Youtube**, **Wordpress**, **etc**.), check out SocialiteProviders[19], which is a collection of **OAuth 1 and 2 packages** that extends **Socialite**.

Actually, Socialite is an official package, and it's not included in Laravel by default. To use Socialite, we have to install it by running this command:

```
composer require laravel/socialite
```

Alternatively, you may edit the **composer.json** file, add below code into the **require** section and run **composer update**:

```
"laravel/socialite": "~2.0"
```

Next, open **config/app.php**.

Add the following line into the **providers** array:

```
Laravel\Socialite\SocialiteServiceProvider::class,
```

Add the **Socialite facade** into the **aliases** array:

```
'Socialite' => Laravel\Socialite\Facades\Socialite::class,
```

Done! Socialte is now ready to use!

Create a Facebook app

In order to use Facebook as our authentication provider, we must create a Facebook app.

Don't worry, it's very simple.

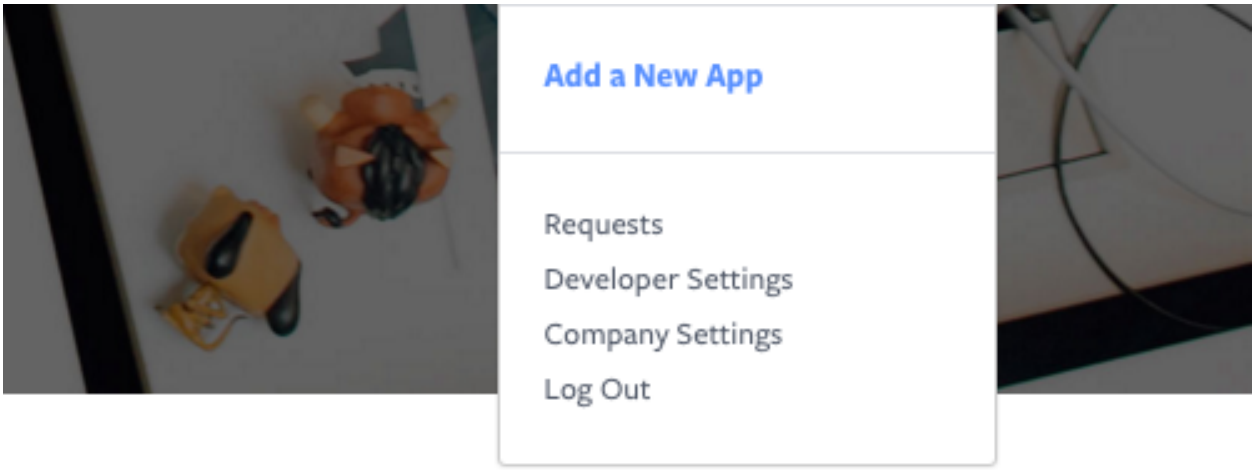
First, let's go to:

https://developers.facebook.com

Register a Facebook account or **login** if you have one already.

There is an **Account Menu** at the top right corner of the page.

[19] https://socialiteproviders.github.io



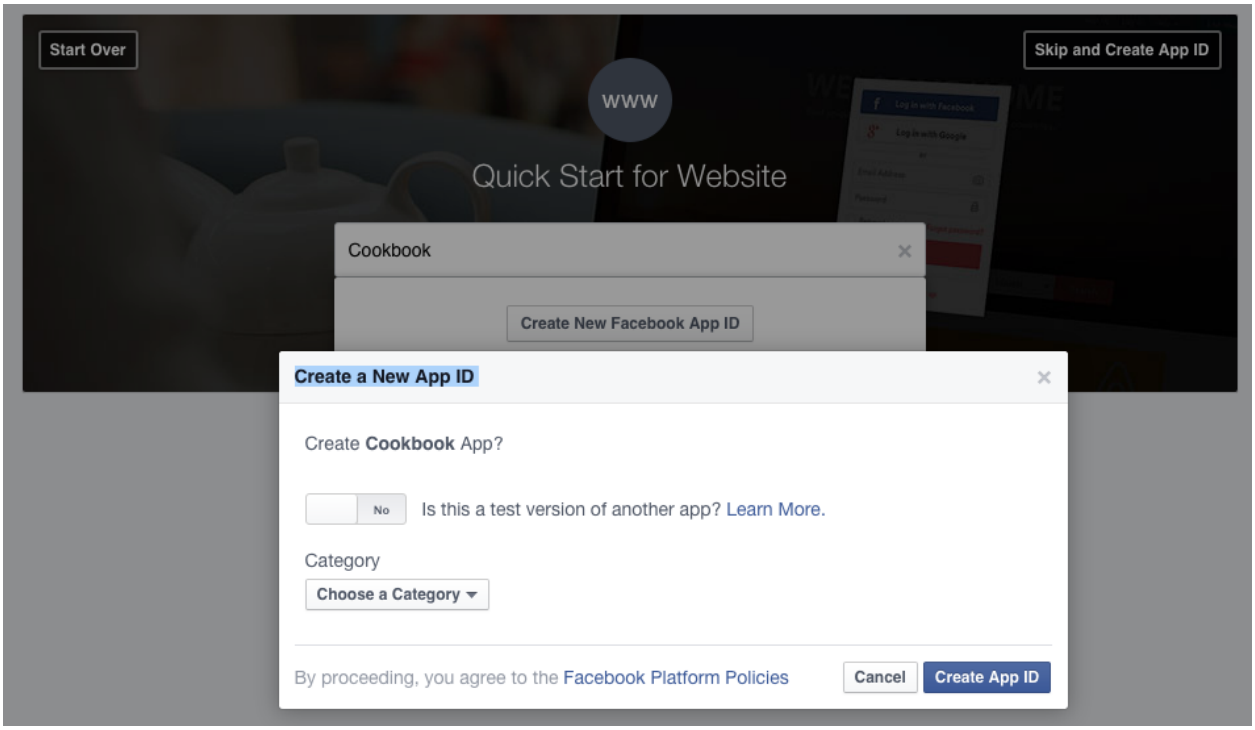
New home page

Click **Add a New App** and choose **Website** to create a Facebook app.

Alternatively, you can access this link:

https://developers.facebook.com/quickstarts/?platform=web

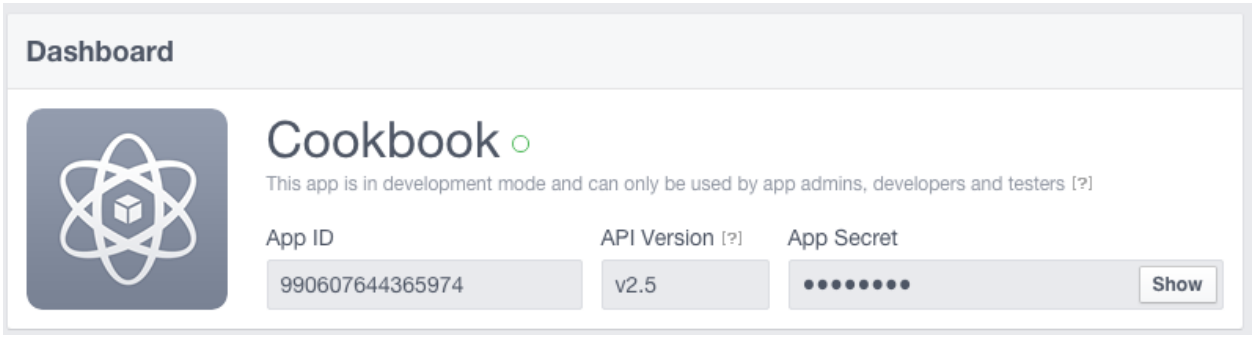
Enter your app's name and click **Create New Facebook App ID**:



New home page

Choose a Category for your app and then click **Create App ID**

Click **Skip Quick Start** to access your **App Dashboard**



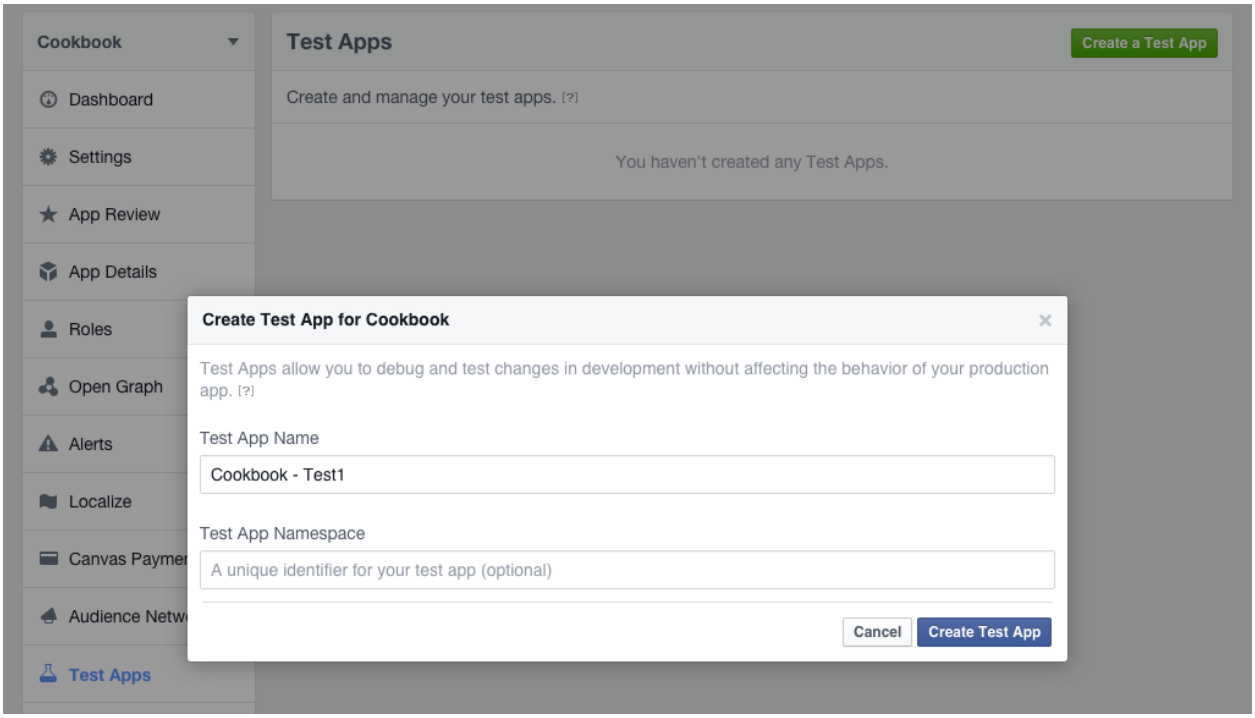
New home page

That's it! You can get your **App ID** and **App Secret** here.

Create a Facebook Test App

If you're working on **Homestead**, you will have to create a **Test App** to test the authentication locally.

Find the **Test Apps** button, which is on the left menu:

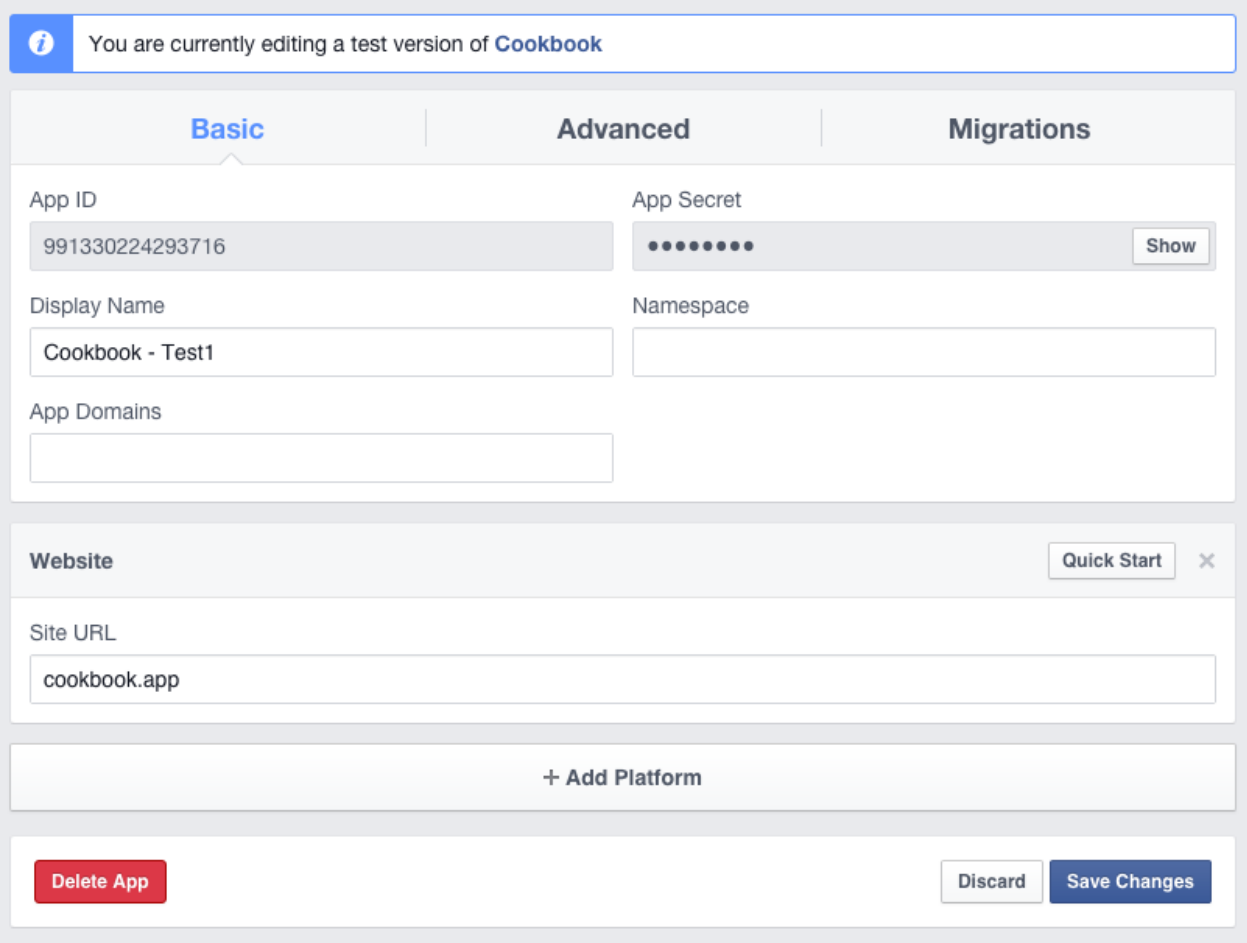


Create a test app

Follow the instructions to create a test app. You can name the app whatever you want.

After that, click **Settings** -> **Add Platforms** -> **Website**

Enter your **app URL** into the **Site URL** field. (For example, cookbook.app)



Add platform and app domain

Enter your **app URL** into the **App Domains** field, too.

Click **Save Changes** to update your Test App.

Well done! Don't forget to grab your **Test App ID** and **Test App Secret**.

Tell Laravel about your Facebook app

After creating your Facebook app, you can connect it to your Laravel app by simply editing the **config/services.php** file. Add this:

```
'facebook' => [
    'client_id' => 'yourFacebookAppID',
    'client_secret' => 'yourFacebookAppSecret',
    'redirect' => 'http://yourLaravelAppURL/login/facebook/callback',
],
```

If you're using the **.env** config file, you may use the following instead:

```
'facebook' => [
    'client_id' => env('FACEBOOK_ID'),
    'client_secret' => env('FACEBOOK_SECRET'),
    'redirect' => env('FACEBOOK_URL'),
],
```

Edit your **.env** files:

```
FACEBOOK_ID=yourFacebookAppID
FACEBOOK_SECRET=yourFacebookAppSecret
FACEBOOK_URL=http://yourLaravelAppURL/login/facebook/callback
```

Done! Laravel automatically detects your Facebook app information and prepares everything for you!

> **Note**: If you're using Homestead, use your **Test App ID and Secret**.

Update Users Migration

We have to update our database to store **users' Facebook Unique ID** and other related information.

This is a new app, so we just need to update the **create_users_table** migration.

> **Note**: Laravel includes the create_users_table by default (2014_10_12_000000_create_users_table.php), we don't need to create it.

Open **database/migrations/timestamps_create_users_table.php** file and update the **up** method:

```
public function up()
{
    Schema::create('users', function (Blueprint $table) {
        $table->increments('id');
        $table->string('facebook_id')->unique();
        $table->string('name');
        $table->string('email')->unique();
        $table->string('password', 60);
        $table->rememberToken();
        $table->timestamps();
    });
}
```

This migration will create a new **users** table with the **facebook_id** column.

If you have an **existing app**, you have to create a new migration to update the **users** table:

```
php artisan make:migration update_users_table
```

Open the **update_users_table** file and update the code as follows:

```
<?php

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

class UpdateUsersTable extends Migration
{
    public function up()
    {
        if(Schema::hasColumn('users', 'facebook_id')) {

        } else {
            Schema::table('users', function ($table) {
                $table->string('facebook_id')->unique();
            });
        }
    }

    public function down()
    {
        Schema::table('users', function ($table) {
            $table->dropColumn('facebook_id');
        });
    }
}
```

Next, don't forget to run **php artisan migrate** to update your database.

> **Note**: If you don't have a database yet, create a new one. The name of our database is **cookbook**. If you wish to learn more about working with databases, read Learning Laravel 5 book's Chapter 3[20].

One last step, open **app/User.php** file and update our **User Model**:

[20] http://learninglaravel.net/laravel5/building-a-support-ticket-system

```
protected $fillable = [
    'name', 'email', 'password', 'facebook_id',
];
```

Update our Routes and AuthController

We will need two routes:

1. A route that redirects users to our OAuth provider, which is Facebook.
2. Another route that receives a response (callback) from Facebook.

Let's open our **routes.php** file and add **these routes**:

```
Route::get('login/facebook', 'Auth\AuthController@redirectToFacebook');
Route::get('login/facebook/callback', 'Auth\AuthController@getFacebookCallback');
```

When users visit http://cookbook.app/login/facebook, Laravel redirects users to Facebook and receive the callback at this route http://cookbook.app/login/facebook/callback.

Easy?

Now we need to create two controller methods, open **Auth/AuthController**, add:

The **redirectToFacebook** method:

```
public function redirectToFacebook()
{
    return Socialite::with('facebook')->redirect();
}
```

The **getFacebookCallback** method:

```
public function getFacebookCallback()
{

    $data = Socialite::with('facebook')->user();
    $user = User::where('email', $data->email)->first();

    if(!is_null($user)) {
        Auth::login($user);
        $user->name = $data->user['name'];
        $user->facebook_id = $data->id;
```

```
11          $user->save();
12      } else {
13          $user = User::where('facebook_id', $data->id)->first();
14          if(is_null($user)){
15              // Create a new user
16              $user = new User();
17              $user->name = $data->user['name'];
18              $user->email = $data->email;
19              $user->facebook_id = $data->id;
20              $user->save();
21          }
22
23          Auth::login($user);
24      }
25      return redirect('/')->with('success', 'Successfully logged in!');
26  }
```

> **Note:** Facebook now returns a **full name** instead of **first name and last name**. We have to use **$data->user['name']** to get the name of the user.

Because we're using the **Socialite facade** and the **Auth facade**, be sure to tell Laravel about them. Find:

```
1  class AuthController extends Controller
```

Add above:

```
1  use Socialite;
2  use Auth;
```

Done! We can now be able to log in or register a new account using Facebook.

Login or register using Facebook

Now, let's visit this link:

http://cookbook.app/login/facebook

We'll be redirected to **Facebook** for **authentication**. If everything is ok, Facebook will redirect us back to our application.

Check our **users** table using Sequel Pro or your favorite database management app, we should see **a new user has been created**.

Just for testing purposes, let's modify our home page:

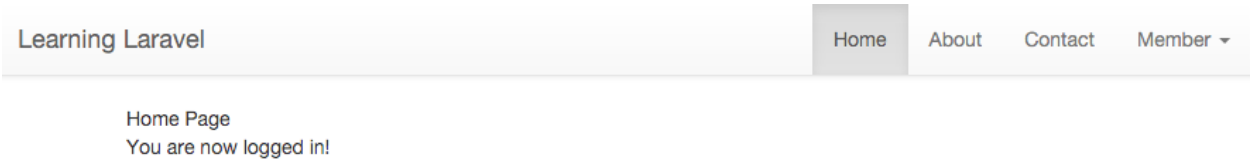
```
@section('content')
    <div class="container">
        <div class="content">
            <div class="title">Home Page</div>
            @if(!Auth::check())
                <div class="quote">Our Home page!</div>
            @else
                <div class="quote">You are now logged in!</div>
            @endif
        </div>
    </div>
@endsection
```

We use **Auth::check()** to check if the user is already logged into our application. If the user is authenticated, we display the **You are now logged in** message.

Once again, because we're using **Laravel 5.2**, we have to put our routes into the **web middleware group** to use **Session**. The **routes.php** should look like this:

```
Route::group(['middleware' => ['web']], function () {
    Route::get('login/facebook', 'Auth\AuthController@redirectToFacebook');
    Route::get('login/facebook/callback', 'Auth\AuthController@getFacebookCallba\
ck');

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

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

    Route::get('users/register', 'Auth\AuthController@getRegister');
    Route::post('users/register', 'Auth\AuthController@postRegister');

});
```

Save the changes and reload our home page, we should see:



New home page

> **Note**: The **users/register** routes are optional. You may use them to build a registration page to create test users (Recipe 203 or Learning Laravel 5 book's Chapter 4) and test the Facebook login feature. If you don't need them, you may just remove them.

Recipe 4 Wrap-up

> **Tag**: Version 0.2 - Recipe 4[21]

That's it! You can now log in or register a new member using Facebook.

Using this technique, you can use **Socialite** to authenticate users with other providers!

By default, your **Facebook app** is in **development mode** (aka **sandbox mode**). Don't forget to **make your app live** and use the real **App ID** and **App Secret**.

If you want to add a **Facebook login button** to your app, simply add the following to wherever you want:

```
1 <a href="/login/facebook"> <div class="btn btn-md btn-primary"> <i class="fa fa-\
2 facebook"></i> Login with Facebook </div></a>
```

> **Note**: We use **Font Awesome** here. If you want to learn how to integrate Font Awesome, please read the next recipe. **Recipe 5** also shows how to **add the Facebook button** into your **login page**.

Recipe 5 - Create A User Authentication System Using Laravel Auth Scaffold

What will we learn?

This recipe shows you how to use Laravel 5.2's Auth Scaffold to build a user authentication system that has: user signup, login/logout, password reset and user dashboard.

[21] https://github.com/LearningLaravel/cookbook/releases/tag/v0.2

Generate the user authentication system

Laravel 5.2 comes with a new auth scaffold that we can use to generate a complete user authentication system with just one line of code.

Let's start by running this Artisan command:

```
1  php artisan make:auth
```

```
vagrant@homestead:~/Code/cookbook$ php artisan make:auth
Created View: /home/vagrant/Code/cookbook/resources/views/auth/login.blade.php
Created View: /home/vagrant/Code/cookbook/resources/views/auth/register.blade.php
Created View: /home/vagrant/Code/cookbook/resources/views/auth/passwords/email.blade.php
Created View: /home/vagrant/Code/cookbook/resources/views/auth/passwords/reset.blade.php
Created View: /home/vagrant/Code/cookbook/resources/views/auth/emails/password.blade.php
Created View: /home/vagrant/Code/cookbook/resources/views/layouts/app.blade.php
Created View: /home/vagrant/Code/cookbook/resources/views/home.blade.php
Created View: /home/vagrant/Code/cookbook/resources/views/welcome.blade.php
Installed HomeController.
Updated Routes File.
Authentication scaffolding generated successfully!
```

Make:auth command

As you see, Laravel has generated **some views**, created a new **HomeController** and updated our **routes.php** file.

Go ahead and reload our home page:

Make:auth command

With all this done, we now have a complete user authentication system!

Understanding the auth scaffold

When you run the **Artisan command**, these views are generated:

- **auth/login.blade.php** - the login page
- **auth/register.blade.php** - the signup page
- **auth/passwords/email.blade.php** - the password reset confirmation page

- **auth/passwords/reset.blade.php** - the password reset page
- **auth/emails/password.blade.php** - the password reset email
- **home.blade.php** - the user dashboard page
- **welcome.blade.php** - the new welcome page

The **routes.php** has been changed:

```
Route::group(['middleware' => 'web'], function () {
    Route::auth();

    Route::get('/home', 'HomeController@index');
});
```

The **home route** (user dashboard) and **HomeController** are also created.

The **Routes::auth()** method is used to define **the login route**, **the register route** and **the password reset routes**.

You may realize that our **home page** and other pages now have **a different master layout**. The new layout is **views/layouts/app.blade.php**:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Laravel</title>

    <!-- Fonts -->
    <link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.4.0/css/fo\
nt-awesome.min.css" rel='stylesheet' type='text/css'>
    <link href="https://fonts.googleapis.com/css?family=Lato:100,300,400,700" re\
l='stylesheet' type='text/css'>

    <!-- Styles -->
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.mi\
n.css" rel="stylesheet">
    {{-- <link href="{{ elixir('css/app.css') }}" rel="stylesheet"> --}}

    <style>
```

```
        body {
            font-family: 'Lato';
        }

        .fa-btn {
            margin-right: 6px;
        }
    </style>
</head>
<body id="app-layout">
    <nav class="navbar navbar-default">
        <div class="container">
            <div class="navbar-header">

                <!-- Collapsed Hamburger -->
                <button type="button" class="navbar-toggle collapsed" data-toggl\
e="collapse" data-target="#spark-navbar-collapse">
                    <span class="sr-only">Toggle Navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>

                <!-- Branding Image -->
                <a class="navbar-brand" href="{{ url('/') }}">
                    Laravel
                </a>
            </div>

            <div class="collapse navbar-collapse" id="spark-navbar-collapse">
                <!-- Left Side Of Navbar -->
                <ul class="nav navbar-nav">
                    <li><a href="{{ url('/home') }}">Home</a></li>
                </ul>

                <!-- Right Side Of Navbar -->
                <ul class="nav navbar-nav navbar-right">
                    <!-- Authentication Links -->
                    @if (Auth::guest())
                        <li><a href="{{ url('/login') }}">Login</a></li>
                        <li><a href="{{ url('/register') }}">Register</a></li>
                    @else
```

```
                        <li class="dropdown">
                            <a href="#" class="dropdown-toggle" data-toggle="dro\
pdown" role="button" aria-expanded="false">
                                {{ Auth::user()->name }} <span class="caret"></s\
pan>
                            </a>

                            <ul class="dropdown-menu" role="menu">
                                <li><a href="{{ url('/logout') }}"><i class="fa \
fa-btn fa-sign-out"></i>Logout</a></li>
                            </ul>
                        </li>
                    @endif
                </ul>
            </div>
        </div>
    </nav>

    @yield('content')

    <!-- JavaScripts -->
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/jquery.min.\
js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.mi\
n.js"></script>
    {{-- <script src="{{ elixir('js/app.js') }}"></script> --}}
</body>
</html>
```

This master layout also has Bootstrap, jQuery, Font Awesome, Lato font and a navigation bar.

Updating our app's layout

Now we're going to update the current layout. We'll use the old **master layout** that we've created because it's much cleaner.

First, clear the contents of **views/layouts/app.blade.php**.

Next, copy the contents of **views/master.blade.php** into **views/layouts/app.blade.php**. To save time, you can copy the code below:

```
<html>
<head>
    <title> @yield('title') </title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6\
/css/bootstrap.min.css">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6\
/css/bootstrap-theme.min.css">

    <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.mi\
n.js"></script>
</head>
<body>

@include('shared.navbar')

@yield('content')

</body>
</html>
```

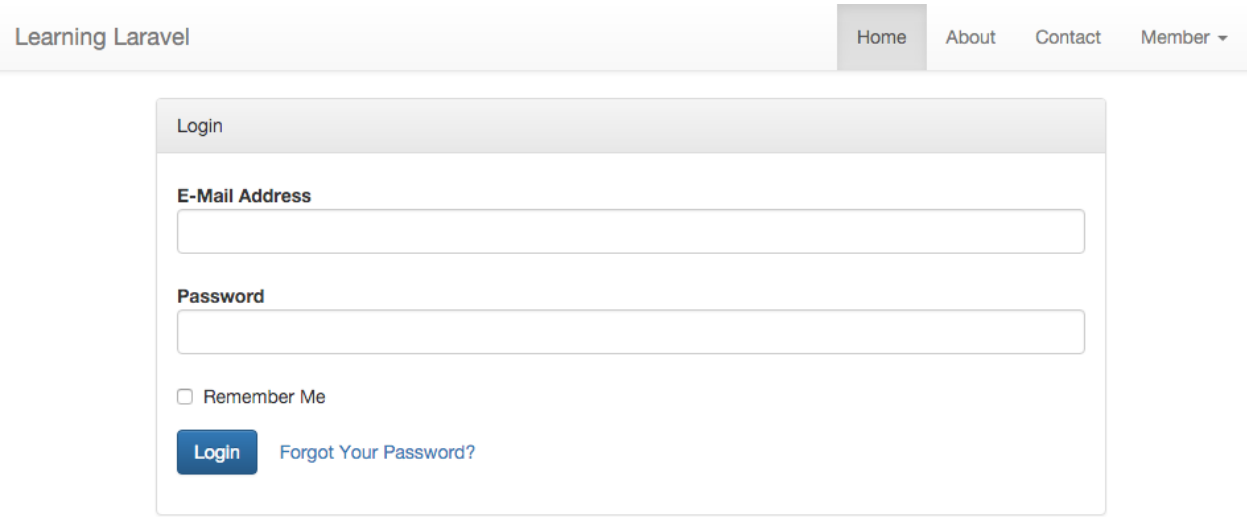
Last step, open **views/shared/navbar.blade.php** and find:

```
<ul class="dropdown-menu" role="menu">
    <li><a href="/register">Register</a></li>
    <li><a href="/login">Login</a></li>
</ul>
```

Replace with:

```
<ul class="dropdown-menu" role="menu">
    <!-- Authentication Links -->
    @if (Auth::guest())
        <li><a href="{{ url('/login') }}">Login</a></li>
        <li><a href="{{ url('/register') }}">Register</a></li>
    @else
        <li class="dropdown">
                <li><a href="{{ url('/logout') }}"><i class="fa fa-btn fa-sign-o\
ut"></i>Logout</a></li>
        </li>
    @endif
</ul>
```

We now have a new layout. All the pages are still working fine.



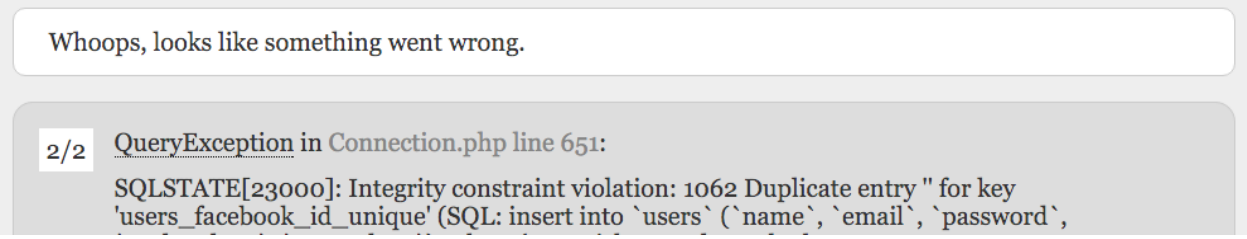
Login page

To test out the new pages, you can try to register a new member, login and logout.

Sadly, our app still has one bug. Let's see what it is and how to fix it in the next section.

Fixing the "new member" bug

If you use **Socialite**, when you try to **register a new member** multiple times, you may encounter this error:



New member error

As you may have guessed, the **facebook_id** column should be **unique**.

To fix this bug, open our **AuthController** file. Update the **create method** as follows:

```
protected function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => bcrypt($data['password']),
        'facebook_id' =>  $data['email'],
    ]);
}
```

It's fairly simple to fix. When users register a new account, their **facebook id** is set as their **provided email**, which should be unique.

Now that we've taken care of the bug!

Adding a Facebook login button

So far, we've worked our way through building an awesome Facebook user authentication. Let's add a Facebook login button to our login and register page, when users click on that button, they can be able to log into our app.

First, open **views/auth/login.blade.php**, and find:

```
<button type="submit" class="btn btn-primary">
    <i class="fa fa-btn fa-sign-in"></i> Login
</button>
```

Add the **Facebook login button** below:

```
<a href="/login/facebook"> <div class="btn btn-md btn-primary"> <i class="fa fa-\
facebook"></i> Login with Facebook </div></a>
```

Next, open **views/auth/register.blade.php**, and find:

```
<button type="submit" class="btn btn-primary">
    <i class="fa fa-btn fa-user"></i>Register
</button>
```

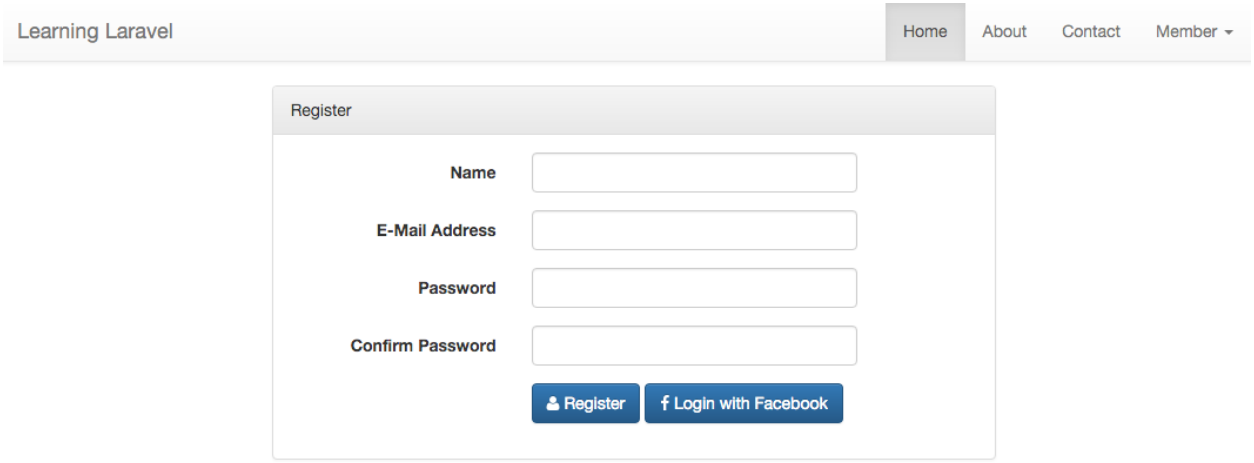
Add the **Facebook login button** below:

```
<a href="/login/facebook"> <div class="btn btn-md btn-primary"> <i class="fa fa-\
facebook"></i> Login with Facebook </div></a>
```

We're using Font Awesome here, so let's add the following inside of our **master layout's head tag** to integrate Font Awesome:

```
<link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.4.0/css/font-a\
wesome.min.css" rel='stylesheet' type='text/css'>
```

Now we can see a nice Facebook login button:



Login Facebook button

Recipe 5 Wrap-up

Tag: Version 0.3 - Recipe 5[22]

Our user authentication system is working perfectly!

Please note that there are many ways to implement authentication, this is just a quick look at implementing a good basis of authentication for our application. If you have time, try to create a user authentication system manually to understand more about it.

Recipe 6 - Image Upload In Laravel

What will we learn?

This recipe shows you how to upload and handle images in Laravel 5.

[22] https://github.com/LearningLaravel/cookbook/releases/tag/v0.3

Installing Intervention Image

Intervention Image is one of the most popular open source **PHP image handling and manipulation** libraries. Using the package, you can easily **create**, **edit and compose** images. The best thing is, Intervention Image doesn't require Laravel or any other framework, it only needs the following components to work correctly:

- **PHP** >= 5.4
- **Fileinfo** Extension
- **GD Library** (>=2.0) or **Imagick PHP** extension (>=6.5.7)

To install Intervention Image, run the following command:

```
composer require intervention/image
```

Next, open your **config/app.php** file.

Add the following line into your **$providers** array:

```
Intervention\Image\ImageServiceProvider::class,
```

Add this Intervention Image's facade into your ** $aliases ** array:

```
'Image' => Intervention\Image\Facades\Image::class,
```

Intervention Image supports both PHP's GD library and Imagick extension. You can choose which one that you want to use in **Intervention Image's configuration file**.

Let's generate **the configuration file** by running this command:

```
php artisan vendor:publish --provider="Intervention\Image\ImageServiceProviderLa\
ravel5"
```

The configuration file is copied to **config/image.php**.

We did it! You can now use **Intervention Image** to manipulate all images!

There's lots more to learn:

- Uploading images
- Display images
- Manipulating images (resize, crop, etc.)

In the next section, we're going to implement a cover image for our home page.

Creating an upload form

Let's start by building an **Image Upload** form.

To keep things simple, let's put the form at our home page. Here's the very beginnings of our **home view (home.blade.php)**:

```
@extends('layouts.app')

@section('content')

<div class="container spark-screen">
    <div class="row">
        <div class="col-md-10 col-md-offset-1">
            <div class="panel panel-default">

                <div class="panel-heading">Dashboard</div>

                <div class="panel-body">

                    <form method="POST" action="/upload" enctype="multipart/form\
-data">

                        @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="image">Choose an image</label>
                            <input type="file" id="image" name="image">
                        </div>

                        <button type="submit" class="btn btn-default">Upload</bu\
tton>

```

```
                    </form>

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

As you see, we've just inserted a new form:

```
<form method="POST" action="/upload" enctype="multipart/form-data">

    @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="image">Choose an image</label>
        <input type="file" id="image" name="image">
    </div>

    <button type="submit" class="btn btn-default">Upload</button>

</form>
```

Please keep in mind that when you upload a file, you need to include this line in your form:

```
enctype="multipart/form-data"
```

Just a quick reminder, we display the **errors** when the form is not valid. If the validator fails, Laravel will **store all errors in the session**. We can easily access the errors via **$errors** object:

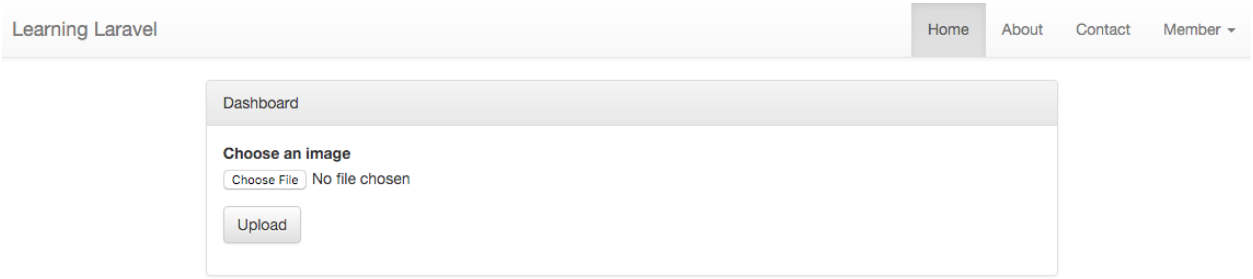
```
@foreach ($errors->all() as $error)
    <p class="alert alert-danger">{{ $error }}</p>
@endforeach
```

We also display **a status message** if the image is uploaded successfully:

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

> **Note**: If you want to learn more about how to work with forms, read the Learning Laravel 5 book's Chapter 3[23]

Now we can see the upload form in our browser:



Upload image form

Storing images

To validate the form, we will create a new **ImageFormRequest**:

```
php artisan make:request ImageFormRequest
```

Go ahead and define the **form's rules** here:

[23] http://learninglaravel.net/laravel5/building-a-support-ticket-system

```php
<?php

namespace App\Http\Requests;

use App\Http\Requests\Request;

class ImageFormRequest extends Request
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'image' => 'required',
        ];
    }
}
```

The next step is **to wire** the form to **a controller** and **process it**.

Let's generate a new **ImagesController**:

```
php artisan make:controller ImagesController
```

Next, create a **store action** to handle images:

```
    public function store(ImageFormRequest $request)
    {

        if ($request->hasFile('image')) {

            $file = $request->file('image');

            $name = $file->getClientOriginalName();

            $file->move(public_path() . '/images/', $name);

            return redirect('/')->with('status', 'Your image has been uploaded s\
uccessfully!');
        }

    }
```

That seems overwhelming at first, but it's actually very easy to understand. First, we check if the form has a file (image) or not:

```
 if ($request->hasFile('image')) {
```

Once the validation is done, we may retrieve all of the input data using:

```
$file = $request->file('image');
```

Now we can get the **image's name**:

```
$name = $file->getClientOriginalName();
```

Sweet! If everything looks good, we can store the file at our **public/images** folder:

```
$file->move(public_path() . '/images/', $name);
```

Finally, **redirect users back** to our home page and **display a status message**:

```
return redirect('/')->with('status', 'Your image has been uploaded successfully!\
');
```

Here is the entire **ImagesController** file:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Requests\ImageFormRequest;

class ImagesController extends Controller
{
    public function store(ImageFormRequest $request)
    {

        if ($request->hasFile('image')) {

            $file = $request->file('image');

            $name = $file->getClientOriginalName();

            $file->move(public_path() . '/images/', $name);

            return redirect('/')->with('status', 'Your image has been uploaded s\
uccessfully!');
        }

    }
}
```

The last thing we need to do is create **a route**:

```php
Route::post('upload', 'ImagesController@store');
```

Well done! Now when users make a **POST request** to this route, Laravel will execute the **ImagesController's store** action.

Let's test what we've just made to make sure that it works properly:

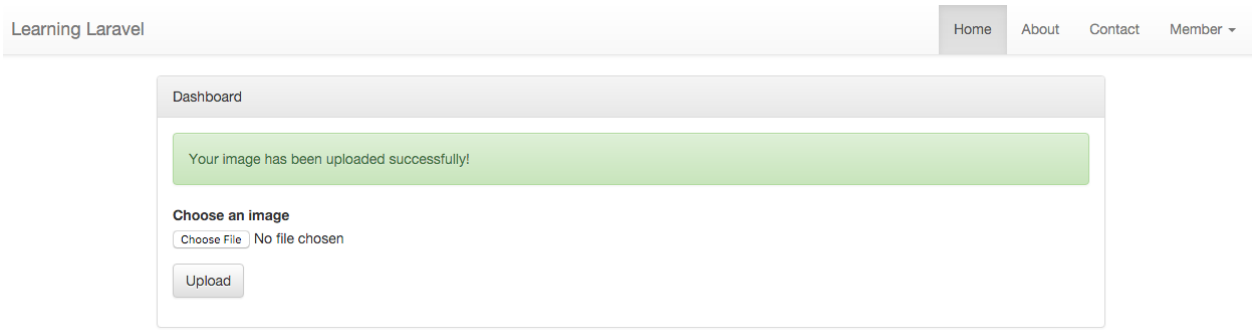
Sample image

I will try to upload this image, you may download it at:

Laravel Cookbook Image[24]

> **Note**: Feel free to use your own image.

Let's view the site in your browser and try to upload the image:



Upload image successfully

Check your **public/images** directory, you will see the image there.

Displaying images

This is the simple part. Once we have that image file, we can easily display it on our home page. Open **home.blade.php**, find:

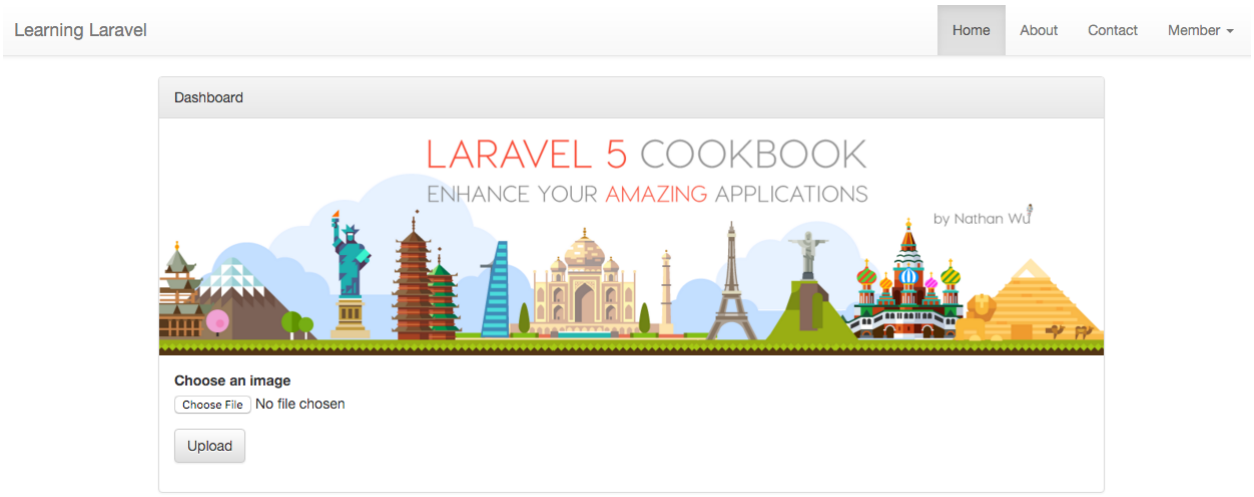
```
1  <div class="panel-heading">Dashboard</div>
```

Add below:

[24] images/chap1-pic24.png

```
<div><img src="/images/chap1-pic24.png" width="100%"/></div>
```

Now we should be able to see our image:



Cover image

Note: If you're using your own image, be sure to change the file name.

Manipulating images

What's the point in installing **Intervention Image** if we don't use it? It's time to use "the power" of **Intervention Image** to manipulate our images!

Note: We talked about **Intervention Image** earlier and this will be your first use of it.

Let's start by telling Intervention Image **where our image is**. First, open the **ImagesController** and find:

```
if ($request->hasFile('image')) {

    $file = $request->file('image');

    $name = $file->getClientOriginalName();

    $file->move(public_path().'/images/', $name);
```

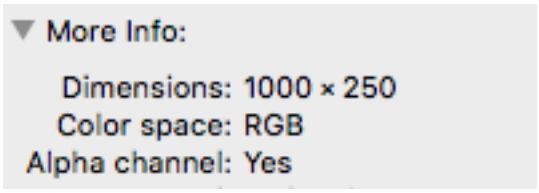
Add below:

```
$imagePath = public_path().'/images/'.$name;
```

When we have **the path of our image**, this is how we **resize** the image:

```
$image = Image::make($imagePath)->resize(1000, 250)->save();
```

Now try to upload the image again and check its size. The image should be resized to **1000x250 px**:



New image size

Super simple stuff!

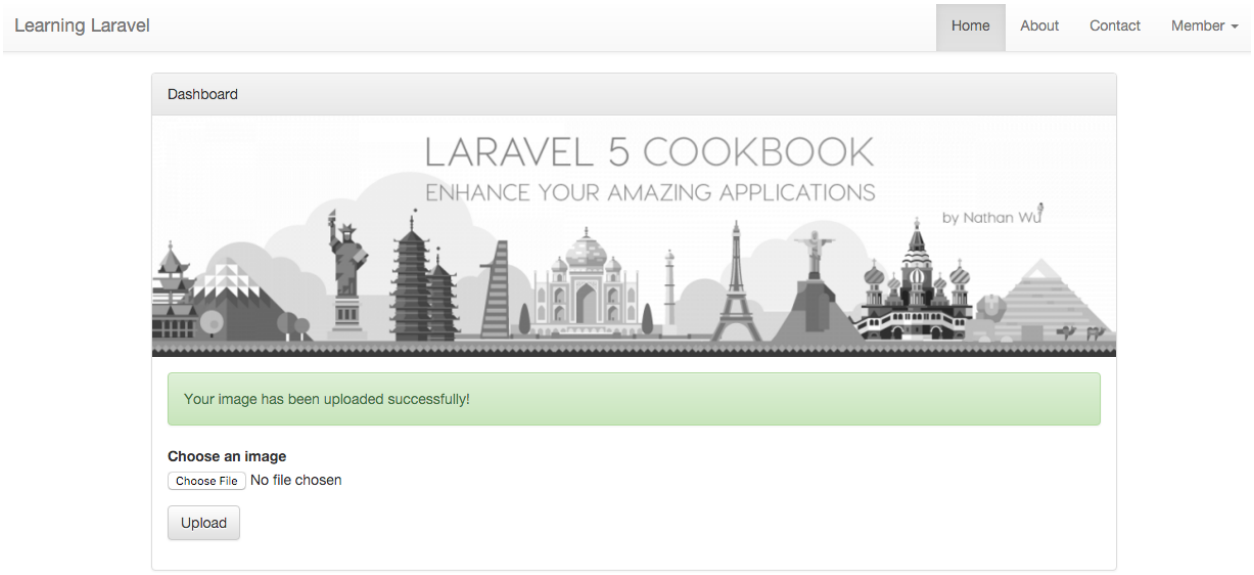
The beauty of Intervention Image is that you can do many things more: **crop**, **blur**, **flip**, **sharpen**, **etc**.

Don't forget to take a look at the **Intervention Image's API**:

http://image.intervention.io[25]

Now let's try to use one more API:

```
$image = Image::make($imagePath)->resize(1000, 250)->greyscale()->save();
```



Greyscale API

[25] http://image.intervention.io/

As you see, the image has been converted to grayscale!

This is what's so cool about **Intervention Image**!

Recipe 6 Wrap-up

> **Tag**: Version 0.4 - Recipe 6[26]

Image upload is a hard concept to grasp if you are new to Laravel development. However, by using **Intervention Image** and **Laravel Request**, we have just created our first image upload function! As you see, the syntax is fairly straightforward.

Recipe 7 - Seeding Your App Using Faker

What will we learn?

This recipe shows you how to use **Faker** - a popular PHP library - to generate fake data for testing purposes.

What is Faker?

Faker is a PHP library that we use to generate dummy data. It can be used to generate all sorts of data for testing purposes or bootstrapping our applications.

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

If you're an old version of Laravel, you can install Faker by running this Composer command:

```
1 composer require fzaninotto/faker
```

Faker can be used to generate:

- Random Digit
- Word
- Paragraph
- Name
- City
- Year
- Domain Name
- Credit Card Number

[26] https://github.com/LearningLaravel/cookbook/releases/tag/v0.4

... and many more.

For more information, please visit **Faker's official documentation**:

https://github.com/fzaninotto/Faker[27]

Creating blog posts using Faker

Here is a quick overview of how to use Faker.

The first step is creating our **Post model** and its **migration**:

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

> **Note**: You can generate the **Post model** and **its 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** and the **down method** as follows:

```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->increments('id');
        $table->string('title', 255);
        $table->text('content');
        $table->string('slug')->nullable();
        $table->tinyInteger('status')->default(1);
        $table->integer('user_id');
        $table->timestamps();
    });
}

public function down()
{
    Schema::drop('posts');
}
```

Don't forget to run:

[27] https://github.com/fzaninotto/Faker

```
php artisan migrate
```

Your database should now have a new **posts** table.

Just like we create our Post model, run this command to create a new **PostsTableSeeder** file:

```
php artisan make:seeder PostsTableSeeder
```

Open the **app/database/seeds/PostsTableSeeder.php** file, update the **run method** as follows:

```
public function run()
{
    $faker = Faker::create();

    foreach(range(1,20) as $index)
    {
        $title = $faker->text(80);

        Post::create([
            'title' => $title,
            'content' =>$faker->paragraph(30),
            'slug' => Str::slug($title, '-'),
            'status' => 1,
            'user_id' => $faker->numberBetween($min = 1, $max = 5),
        ]);
    }

}
```

It's very easy to understand, right?

Find:

```
class PostsTableSeeder extends Seeder
```

Add above:

```
use App\Post;
use Faker\Factory as Faker;
use Illuminate\Support\Str;
```

In our **app/database/seeds/DatabaseSeeder.php**, update the **run method** as follows:

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

The last part needed to **seed data** is to run this Artisan command:

```
php artisan db:seed
```

Check your database, you should see **20 new posts**:

id	title	content	slug	status	user_id
1	Modi non architecto distinctio dolorem provident.	In inventore quo commodi cumque placeat eos. Ab iste fugiat incidunt tenetur id. Soluta exce…	modi-non-architecto-distinctio-dolorem-provident	1	1
2	Adipisci provident et aut laborum et.	Expedita similique est ea dignissimos quisquam error nulla odit. Qui omnis libero sit a sed ist…	adipisci-provident-et-aut-laborum-et	1	2
3	Sint voluptas ab veritatis iste quaerat.	Dolores recusandae vitae totam laborum aliquid. Ipsa facere error neque et. Velit qui harum fu…	sint-voluptas-ab-veritatis-iste-quaerat	1	4
4	Magni enim voluptas ut sunt et. In dolorem animi fuga quo…	Laborum ut quasi perspiciatis voluptas sit. Voluptates ipsam praesentium eligendi quia laboru…	magni-enim-voluptas-ut-sunt-et-in-dolorem-animi-fuga-quo-unde-totam	1	2
5	Optio qui doloribus iste natus.	Voluptatem quasi qui facere. Earum eum in omnis esse et. Maxime aut minima numquam qui…	optio-qui-doloribus-iste-natus	1	2
6	Harum voluptas esse vel. Facilis maxime consectetur iure in…	Iusto aut doloremque asperiores voluptatem. Consequatur harum commodi sed distinctio. Exp…	harum-voluptas-esse-vel-facilis-maxime-consectetur-iure-in-non-atque	1	2
7	Sunt possimus veniam deserunt provident.	Non modi non qui autem consequatur dignissimos. Nam quibusdam similique placeat labore…	sunt-possimus-veniam-deserunt-provident	1	3
8	Eos minima non dolorum. Cum est qui voluptatem laborum quia.	Dolorem possimus dolore dolorem dolorem qui. Omnis qui dolores voluptatibus labore. Non e…	eos-minima-non-dolorum-cum-est-qui-voluptatem-laborum-quia	1	3
9	Impedit expedita ullam et consequatur officia ad.	Qui deserunt harum reprehenderit quis ullam provident omnis molestiae. Ea dolor ut quos rati…	impedit-expedita-ullam-et-consequatur-officia-ad	1	3
10	Quos architecto accusantium non. Eos praesentium repellen…	Quisquam veniam id rerum. Reprehenderit eius omnis eos excepturi odio. Molestias quod dele…	quos-architecto-accusantium-non-eos-praesentium-repellendus-quia-qui	1	5
11	Ut voluptas cupiditate placeat non voluptates doloremque v…	Aperiam quaerat consequuntur aut molestias. Quos et provident quo itaque aut nesciunt aliqu…	ut-voluptas-cupiditate-placeat-non-voluptates-doloremque-vero-repellat	1	2
12	Debitis minima ut iure deserunt fugiat impedit quia.	Totam vitae voluptatem iusto. Qui consequatur eaque est quos. Repudiandae minus nisi alias…	debitis-minima-ut-iure-deserunt-fugiat-impedit-quia	1	1
13	Et nihil similique iusto quis et optio. Tenetur a voluptatibus…	Aut aut voluptatem numquam. Recusandae ad dolores accusamus voluptatem illo ad eos. Lau…	et-nihil-similique-iusto-quis-et-optio-tenetur-a-voluptatibus-quaerat-sit	1	1
14	Accusamus nostrum ducimus aut aut adipisci maiores et rei…	Quo molestias quo quasi. Deleniti sit corporis voluptatem quia temporibus qui et. Nisi eaque s…	accusamus-nostrum-ducimus-aut-aut-adipisci-maiores-et-reiciendis	1	2
15	Cumque qui deserunt consequuntur voluptates eaque volup…	Quis eveniet odit et id occaecati. Et incidunt molestiae iure eos error fugit sit debitis. Quam au…	cumque-qui-deserunt-consequuntur-voluptates-eaque-voluptatem	1	4
16	Consequuntur temporibus aut corporis nam nihil vel dicta.	Maiores ab laboriosam quis in placeat. Ad sit ut reprehenderit. Nihil nihil ex vel aut. Illum aper…	consequuntur-temporibus-aut-corporis-nam-nihil-vel-dicta	1	3
17	Ut quia ad magnam accusantium.	Accusantium est officia corrupti deleniti quo sed fuga. Est ea totam et et officiis. Voluptates cu…	ut-quia-ad-magnam-accusantium	1	4
18	Illo aut eligendi ea. Inventore temporibus et quia voluptas a…	Non quaerat ipsa tenetur ad doloremque unde. Laboriosam tenetur ex recusandae ut. Rerum…	illo-aut-eligendi-ea-inventore-temporibus-et-quia-voluptas-ad-sequi	1	4
19	Dolores quis cumque ut commodi accusamus similique con…	Vero explicabo non aliquid dolores. Et provident et mollitia. Aliquid nihil tempore qui maxime…	dolores-quis-cumque-ut-commodi-accusamus-similique-consequatur	1	2
20	Aperiam ut necessitatibus voluptatibus praesentium est ut id.	Doloremque recusandae quasi modi vel. Est ea ab expedita labore nobis ea. Optio nam corpori…	aperiam-ut-necessitatibus-voluptatibus-praesentium-est-ut-id	1	1

20 new posts

Display all blog posts

Once we have the dummy data, we can create **a blog page** to view **all blog posts**.

Let's register a blog route by adding the following code to our **routes.php** file:

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

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

```
php artisan make:controller BlogController
```

Insert the following code into it:

Find:

```
class BlogController extends Controller
```

Tell Laravel that you want to use the **Post model** in this controller. Add above:

```
use App\Post;
```

Next, add the **index** action:

```
public function index()
{
    $posts = Post::all();
    return view('blog.index', compact('posts'));
}
```

Create a new **index** view at **views/blog** directory. The following are the contents of the **views/blog/index.blade.php** file:

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

    <div class="container col-md-8 col-md-offset-2">

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

        @if ($posts->isEmpty())
            <p> There is no post.</p>
        @else
            @foreach ($posts as $post)
                <div class="panel panel-default">
                    <div class="panel-heading">{!! $post->title !!}</div>
                    <div class="panel-body">
                        {!! mb_substr($post->content,0,500) !!}
                    </div>
                </div>
            @endforeach
        @endif
    </div>

@endsection
```

We use **mb_substr** (Multibyte String) function to display only **500 characters** of the post.

If you want to learn more about the function, visit:

http://php.net/manual/en/function.mb-substr.php[28]

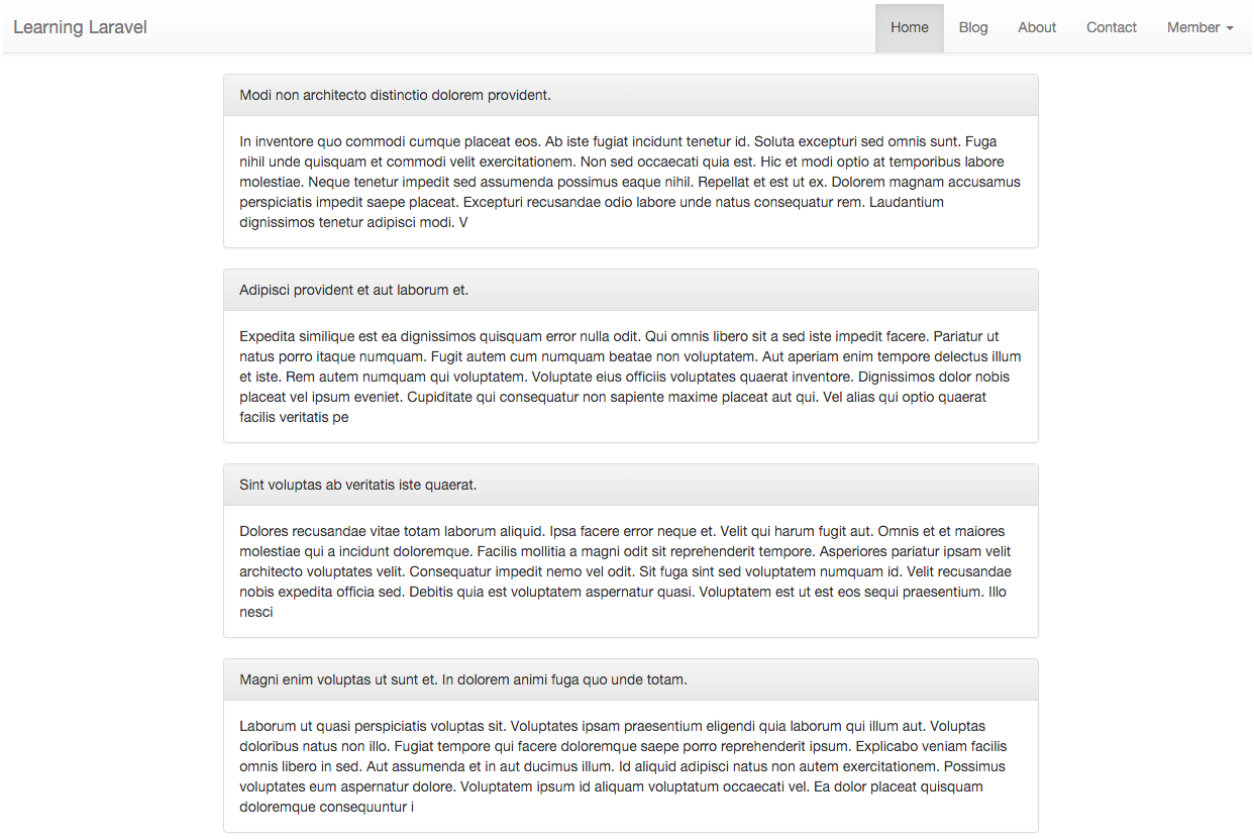
We should add a **blog** link to our navigation bar to access the blog page faster. Open **shared/-navbar.blade.php** and find:

```
1 <li><a href="/about">About</a></li>
```

Add above:

```
1 <li><a href="/blog">Blog</a></li>
```

Head over to your browser and visit the **blog page**.



The blog page

Recipe 7 Wrap-up

Tag: Version 0.5 - Recipe 7[29]

[28] http://php.net/manual/en/function.mb-substr.php

[29] https://github.com/LearningLaravel/cookbook/releases/tag/v0.5

You now know how to use Faker to generate good-looking dummy data!

This technique can be applied to more than just posts. You can use it to create larger applications and test your apps.

Recipe 8 - Pagination

What will we learn?

We will learn how to implement pagination in your Laravel application.

Simple pagination

Recently, we showed all blog posts in one page. Eventually, if thousands of posts are posted, this will become problematic. **Pagination** is a good solution to this overload issue.

As you may know, creating pagination from scratch is not an easy task. Luckily, Laravel has a **paginate method** that we can use to create the pagination without having to write any extra code.

Let's open our **BlogController**, find:

```
1  $posts = Post::all();
```

Replace with:

```
1  $posts = Post::paginate(10);
```

As you see, we use the **paginate method** to create the pagination. This will return an instance of **IlluminatePaginationLengthAwarePaginator**.

Alternatively, you can use the new **simplePaginate** method. This will return an instance of **IlluminatePaginationPaginatorThis Paginator** class.

```
1  $posts = Post::simplePaginate(10);
```

These objects provide several useful methods that we can use to customize and display our pagination.

I want to display **10 posts** on a page, so I put **10** as **the parameter**.

If you would like to use **Query Builder**, you may write:

```
$posts = DB::table('posts')->paginate(10);
```

For more information, be sure to read the official documentation:

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

Next, open the **blog/index.blade.php** view and find:

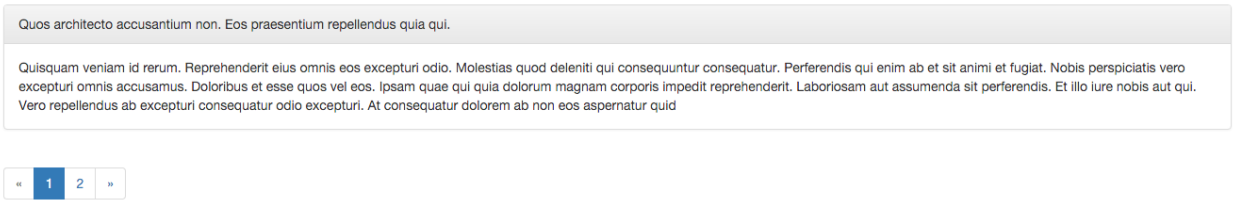
```
    @if ($posts->isEmpty())
        <p> There is no post.</p>
    @else
        @foreach ($posts as $post)
            <div class="panel panel-default">
                <div class="panel-heading">{!! $post->title !!}</div>
                <div class="panel-body">
                    {!! mb_substr($post->content,0,500) !!}
                </div>
            </div>
        @endforeach
    @endif
```

Add below:

```
{!! $posts->render() !!}
```

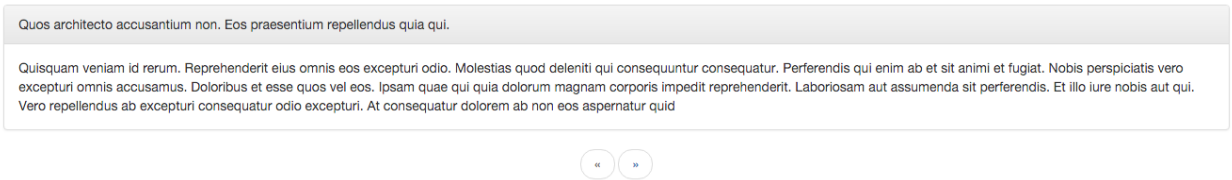
Good job!

Let's give our brand new pagination system a try:



Pagination

If you use the **simplePaginate** method, you should now see something like this:



Simple Pagination

Additional helper methods

The **Paginator instances** also have many useful methods that you can access:

- $posts->count()
- $posts->currentPage()
- $posts->hasMorePages()
- $posts->lastPage() (Not available when using simplePaginate)
- $posts->nextPageUrl()
- $posts->perPage()
- $posts->previousPageUrl()
- $posts->total() (Not available when using simplePaginate)
- $posts->url($page)

As Laravel is constantly updated, be sure to check the documentation to know all latest helper methods:

https://laravel.com/docs/master/pagination[30]

Ajax pagination

When creating Ajax pagination, we will need to return the pagination as JSON. **The Paginator classes** implement the **IlluminateContractsSupportJsonableInterface** contract and have **toJson** method. That means you can easily convert the result instance to JSON by simply **returning** it from **a route** or **controller action**.

Let's try to return the pagination as JSON from our **BlogController's index action**:

```
1  use Response;
2  class BlogController extends Controller
3  {
4      public function index()
5      {
6          $posts = Post::paginate(10);
7          $response = Response::json($posts,200);
8          return $response;
9      }
10 }
```

[30] https://laravel.com/docs/master/pagination

Here is the new blog:


```
{
   total: 20,
   per_page: 10,
   current_page: 1,
   last_page: 2,
   next_page_url: "http://cookbook.app/blog?page=2",
   prev_page_url: null,
   from: 1,
   to: 10,
 - data: [
    - {
         id: 1,
         title: "Modi non architecto distinctio dolorem provident.",
         content: "In inventore quo commodi cumque placeat eos. Ab iste fugiat incidunt tenetur id. Soluta excepturi sed omnis sunt. Fuga nihil unde quisquam et commodi velit
         exercitationem. Non sed occaecati quia est. Hic et modi optio at temporibus labore molestiae. Neque tenetur impedit sed assumenda possimus eaque nihil. Repellat et est
         ut ex. Dolorem magnam accusamus perspiciatis impedit saepe placeat. Excepturi recusandae odio labore unde natus consequatur rem. Laudantium dignissimos tenetur adipisci
         modi. Voluptatem sit adipisci ullam autem sint. Expedita numquam eaque adipisci vero et rerum. Magni ducimus fugit optio perferendis. Asperiores doloribus rerum illo
         aut. Qui ad ipsum qui ipsam ut quibusdam minus qui. Eligendi doloremque tenetur quia eius tempora architecto sit modi. Nihil voluptas laborum et rerum deserunt est vel.
         Eius qui vel nemo qui itaque sit. Occaecati explicabo culpa quisquam molestiae. Sapiente sunt expedita est debitis aliquam officiis. Enim veritatis iste aut minus
         tempore tempora debitis. Repellat ipsum dolor voluptatem aperiam. Unde itaque distinctio tempora rerum dolores perspiciatis non. Quam dolorum voluptatibus provident
         nulla. Voluptatibus rerum architecto sunt sequi suscipit nisi.",
         slug: "modi-non-architecto-distinctio-dolorem-provident",
         status: 1,
         user_id: 1,
         created_at: "2016-02-27 19:55:47",
         updated_at: "2016-02-27 19:55:47"
      },
    - {
         id: 2,
         title: "Adipisci provident et aut laborum et.",
         content: "Expedita similique est ea dignissimos quisquam error nulla odit. Qui omnis libero sit a sed iste impedit facere. Pariatur ut natus porro itaque numquam. Fugit
         autem cum numquam beatae non voluptatem. Aut aperiam enim tempore delectus illum et iste. Rem autem numquam qui voluptatem. Voluptate eius officiis voluptates quaerat
         inventore. Dignissimos dolor nobis placeat vel ipsum eveniet. Cupiditate qui consequatur non sapiente maxime placeat aut qui. Vel alias qui optio quaerat facilis
         veritatis perferendis. Nesciunt incidunt itaque vitae itaque sit consequuntur quisquam. Labore incidunt dolor praesentium dignissimos ut. Iusto corrupti beatae et
         laboriosam et ad et. Ut quo minima quia odio. Quasi nihil tenetur ad perferendis aut minus veritatis. Laudantium qui soluta odio autem id dolorem. Vitae nisi id est
         similique. Facere aut hic et quia quis excepturi. Corrupti ea doloremque sit consequatur iure sit commodi.",
         slug: "adipisci-provident-et-aut-laborum-et",
         status: 1,
         user_id: 2,
         created_at: "2016-02-27 19:55:47",
         updated_at: "2016-02-27 19:55:47"
      },
```


Retuning the pagination as JSON

Now let's try to return the instance from **a route**. Open our **routes.php** file, add a new route:

```
1  Route::get('json', function () {
2      return App\Post::paginate();
3  });
```

Visit **cookbook.app/json**, you should see:

```
{
   total: 20,
   per_page: 15,
   current_page: 1,
   last_page: 2,
   next_page_url: "http://cookbook.app/json?page=2",
   prev_page_url: null,
   from: 1,
   to: 15,
 - data: [
    - {
         id: 1,
         title: "Modi non architecto distinctio dolorem provident.",
         content: "In inventore quo commodi cumque placeat eos. Ab iste fugiat incidunt tenetur id. Soluta excepturi sed omnis sunt. Fuga nihil unde quisquam et commodi velit
         exercitationem. Non sed occaecati quia est. Hic et modi optio at temporibus labore molestiae. Neque tenetur impedit sed assumenda possimus eaque nihil. Repellat et est
         ut ex. Dolorem magnam accusamus perspiciatis impedit saepe placeat. Excepturi recusandae odio labore unde natus consequatur rem. Laudantium dignissimos tenetur adipisci
         modi. Voluptatem sit adipisci ullam autem sint. Expedita numquam eaque adipisci vero et rerum. Magni ducimus fugit optio perferendis. Asperiores doloribus rerum illo
         aut. Qui ad ipsum qui ipsam ut quibusdam minus qui. Eligendi doloremque tenetur quia eius tempora architecto sit modi. Nihil voluptas laborum et rerum deserunt est vel.
         Eius qui vel nemo qui itaque sit. Occaecati explicabo culpa quisquam molestiae. Sapiente sunt expedita est debitis aliquam officiis. Enim veritatis iste aut minus
         tempore tempora debitis. Repellat ipsum dolor voluptatem aperiam. Unde itaque distinctio tempora rerum dolores perspiciatis non. Quam dolorum voluptatibus provident
         nulla. Voluptatibus rerum architecto sunt sequi suscipit nisi.",
         slug: "modi-non-architecto-distinctio-dolorem-provident",
         status: 1,
         user_id: 1,
         created_at: "2016-02-27 19:55:47",
         updated_at: "2016-02-27 19:55:47"
      },
    - {
         id: 2,
         title: "Adipisci provident et aut laborum et.",
         content: "Expedita similique est ea dignissimos quisquam error nulla odit. Qui omnis libero sit a sed iste impedit facere. Pariatur ut natus porro itaque numquam. Fugit
         autem cum numquam beatae non voluptatem. Aut aperiam enim tempore delectus illum et iste. Rem autem numquam qui voluptatem. Voluptate eius officiis voluptates quaerat
         inventore. Dignissimos dolor nobis placeat vel ipsum eveniet. Cupiditate qui consequatur non sapiente maxime placeat aut qui. Vel alias qui optio quaerat facilis
         veritatis perferendis. Nesciunt incidunt itaque vitae itaque sit consequuntur quisquam. Labore incidunt dolor praesentium dignissimos ut. Iusto corrupti beatae et
         laboriosam et ad et. Ut quo minima quia odio. Quasi nihil tenetur ad perferendis aut minus veritatis. Laudantium qui soluta odio autem id dolorem. Vitae nisi id est
         similique. Facere aut hic et quia quis excepturi. Corrupti ea doloremque sit consequatur iure sit commodi.",
         slug: "adipisci-provident-et-aut-laborum-et",
         status: 1,
         user_id: 2,
         created_at: "2016-02-27 19:55:47",
         updated_at: "2016-02-27 19:55:47"
      },
```


Retuning the pagination as JSON from a route

Recipe 8 Wrap-up

Tag: Version 0.6 - Recipe 8[31]

Great! Having the knowledge of the above will let you create a simple pagination or ajax pagination in no time.

Now this won't do much for our app yet, but these techniques can be used to build many applications in all sorts of different styles.

Recipe 9 - Testing Your App

What will we learn?

No code is safe! This recipe shows you how to do testing to make sure that everything is working like it's supposed to.

Why should we do testing?

Most of the time, you can successfully deploy and run your apps without any problems. Unfortunately, sometimes things might go terribly horribly wrong. That's when you need testing skills to pinpoint at the right issues.

[31] https://github.com/LearningLaravel/cookbook/releases/tag/v0.6

Testing is really boring and there's a lot of code which is hard to unit test properly. That is true. However, not having a comprehensive test suite means that our applications may not meet the stated requirements and we're taking more risks.

> "Even good programmers make mistakes. The difference between a good programmer and a bad one is that a good one detects it sooner by using automated tests" - Sebastian Bergmann

That may sound a bit exaggerated, but we should know how to write and test our code effectively to improve the quality of our applications.

To beginners of Laravel, testing can truly seem like a very difficult job. But don't worry.

Remember that, **if you can write PHP**, **you can write tests**.

Now let's get started and talk more about these concepts as we go along!

Manual testing and automated testing

There are many ways to test your app. When talking about testing, people usually think about **automated test**, but we can always go for a **manual testing** approach as well.

- **Manual Testing**: is the process of running series of tasks manually to find the defects in our applications.
- **Automated Testing**: is the process of using automated tools to run tests based on algorithms to check your applications.

There are different types of **automated tests**. Here are the popular ones:

- Unit tests
- Integration tests
- Acceptance tests (aka Functional tests)

In this recipe, we will talk about some useful tools and techniques that we can use to manually test our app. We also learn about **PHPUnit** and write some automated tests to check our application.

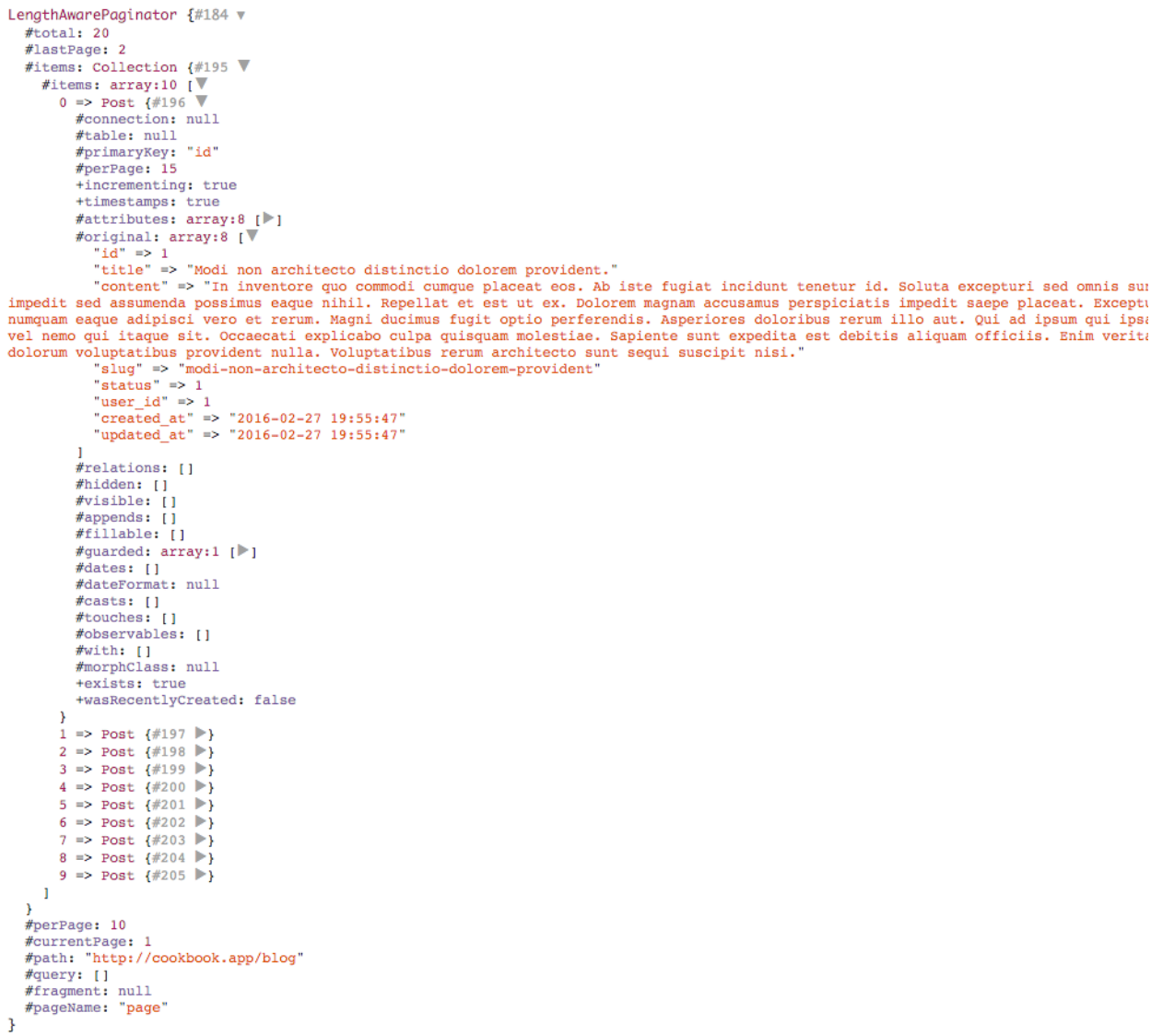
dd(), var_dump(), print_r() and Kint

Laravel has a popular helper function that we can use to **display structured information** of **the given variable** and **stop the script's execution**: **dd()**.

Now let's open our **BlogController**, and add this function to our **index action**:

```php
public function index()
{
    $posts = Post::paginate(10);
    dd($posts);
```

Visit our blog to see the changes:



Use dd() function

Using the **paginate method**, we will receive an instance of **IlluminatePaginationLengthAwarePaginator**. The **dd() function** helps us to see the contents of the instance. If you don't get the instance or the information is not correct, then your app is probably having a bug somewhere.

As you see, we can read the title and content of our posts (and many things more) without using

views to display it.

We can also use the **dd() function** to dump the **response object** as well:

```
    $response = Response::json($posts,200);
    dd($response);
```

Or we can just display a simple text:

```
dd("This is a test");
```

Amazing, right?

You'll be using this function a lot since it's very useful for showing off data and debugging our app.

Alternatively, we can use **var_dump()** and **print_r()** PHP function to display the information about a variable. These functions are very useful when working with arrays.

W3resouce has a really good tutorial about them, you may check it out at:

http://www.w3resource.com/php/function-reference/var_dump.php[32]

Recently, **Kint** - a powerful and modern PHP debugging tool - is also becoming very popular.

It's a good replacement for **var_dump()** and **print_r()**. Using **Kint** in conjunction with the **dd() function** is a powerful combination.

You can install **Kint** by simply running this Composer command:

```
composer require raveren/kint
```

Or add it into your **composer.json** file.

```
"require": {
   "raveren/kint": "^1.0"
}
```

For more information, check out **Kint**'s official home page:

https://github.com/raveren/kint[33]

[32] http://www.w3resource.com/php/function-reference/var_dump.php
[33] https://github.com/raveren/kint

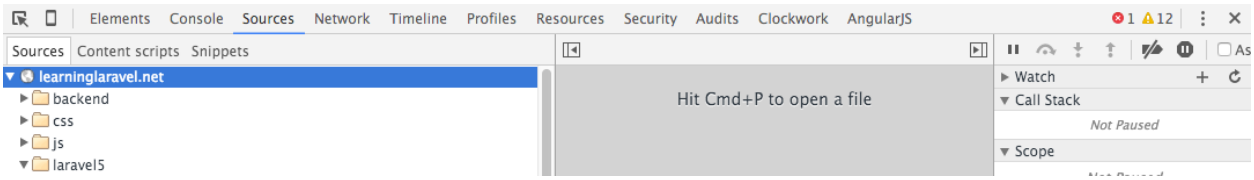
Useful tools, extensions and packages

As PHP is the most popular open source server-side scripting language, it has ready-to-use tools, well-supported extensions, and free packages that can help us properly test, debug and optimize our application.

Here is a list of interesting tools that I use when developing different types of applications:

Developer Tools:[34] The Developer Tools are part of the open source **Webkit** project. They are bundled and available in **Chrome**, **Safari**, **Opera and any Webkit browser**. If you're using **Chrome**, you should use **Chrome Developer Tools** (aka **Chrome DevTools**). The tools let you do many things: inspect elements, view raw html/css, manipulate DOM, debug local browser storage, etc.

You can open **the DevTools** by **right click** and choose **Inspect** or use **this shortcut**: **Ctrl + Shift + I** (Windows), **F12** (Windows), or **Cmd + Opt + I** (Mac)



Chrome DevTools

Want to learn how to use **Devtools**? **CodeSchool** has a nice video course about it (free):

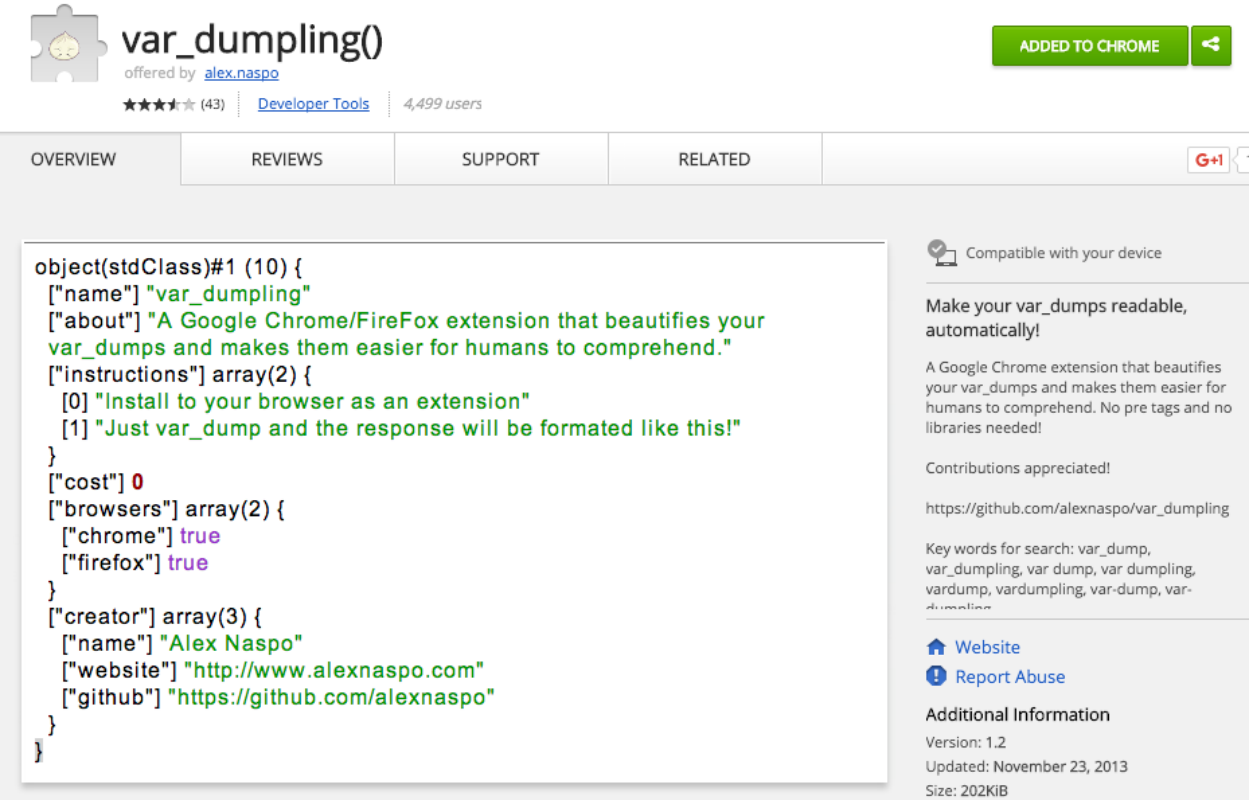
http://discover-devtools.codeschool.com[35]

You may also check the **Chrome Devtools' documentation**:

https://developers.google.com/web/tools/chrome-devtools/iterate/inspect-styles/shortcuts [36]

vardumpling() extension:[37] This Google Chrome extension beautifies your **var_dumps** and makes them easier to read.

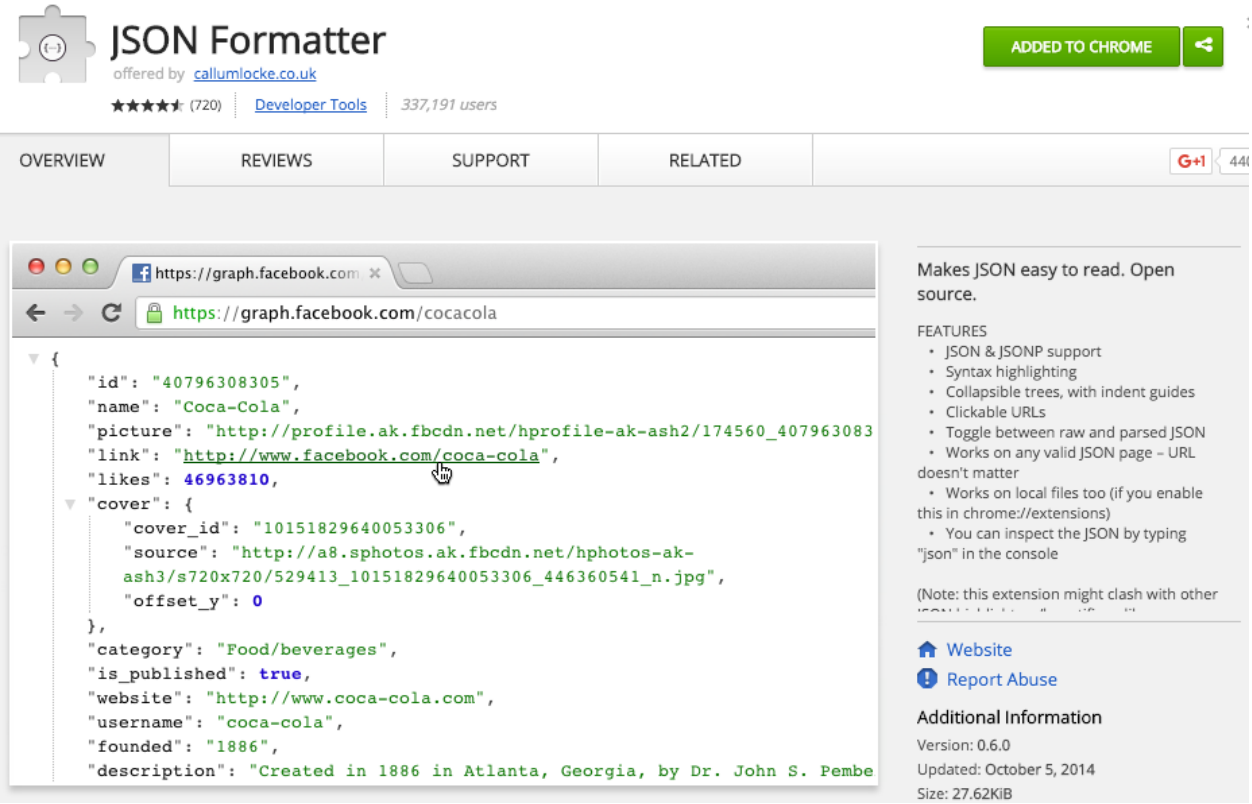
[34] https://developers.google.com/web/tools/chrome-devtools/iterate/inspect-styles/shortcuts
[35] http://discover-devtools.codeschool.com
[36] https://developers.google.com/web/tools/chrome-devtools/iterate/inspect-styles/shortcuts
[37] https://chrome.google.com/webstore/detail/vardumpling/aikblkmigebodlhkdepmfmgdgmbokkdn



vardumpling

JSON Formatter extension:[38] Similar to the **vardumpling** extension, if you want to view JSON files directly on your browser, you'll love this Chrome's plugin.

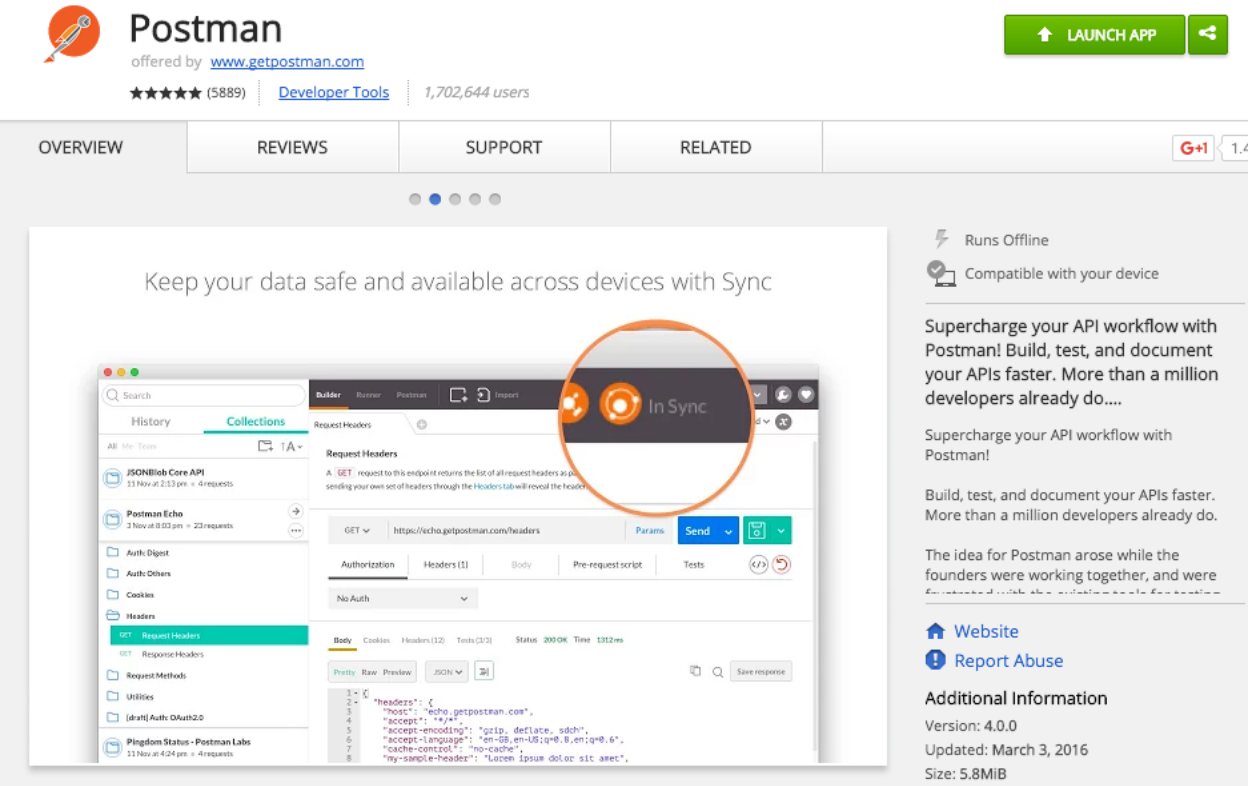
[38] https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa



JSON Formatter

Postman:[39] This is the most popular extension for working with APIs. Postman helps us build, test, and document APIs faster.

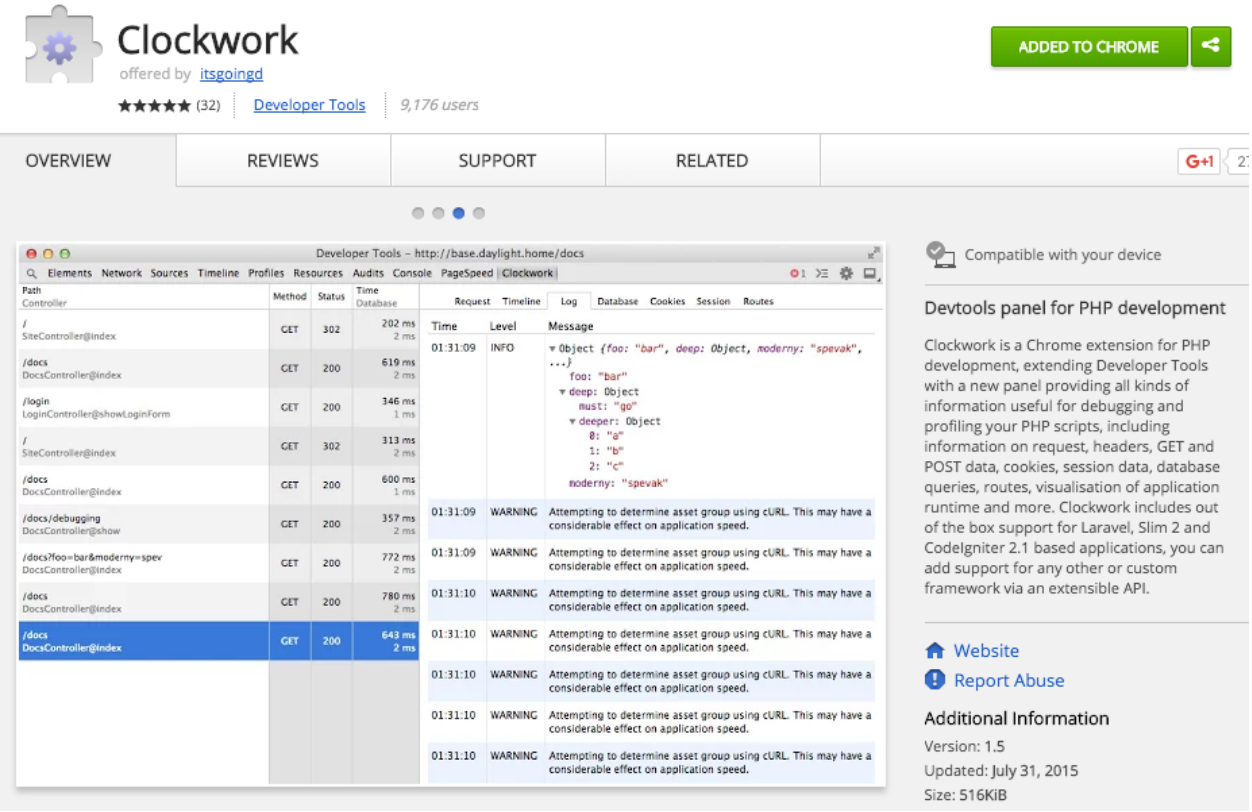
[39] https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en



Postman

Clockwork:[40] Chrome DevTools doesn't support PHP by default, but we can extend the DevTools with a new panel that supports PHP and Laravel. Once installed, we can debug, view cookies/sessions data, run database queries, etc.

[40] https://chrome.google.com/webstore/detail/clockwork/dmggabnehkmmfmdffgajcflpdjlnoemp



Clockwork

Laravel Debug Bar:[41] Laravel Debugbar is one of the best Laravel packages. It adds a "**debug bar**" to our application. Using the bar, we can view all important information of our site, such as: queries, routes, views, collections, etc.

It's very easy to install the package. First, run this Composer command:

```
composer require barryvdh/laravel-debugbar
```

After that, open your **config/app.php** file and add **Debugbar's ServiceProvider** to the **providers** array:

```
Barryvdh\Debugbar\ServiceProvider::class,
```

Next, add the **Debugbar facade** to the **aliases** array:

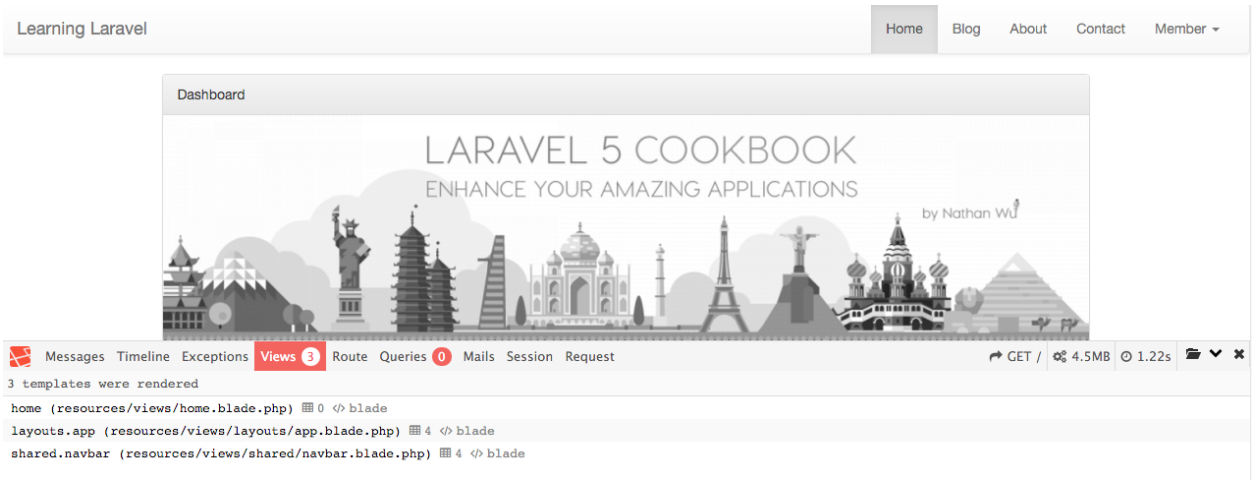
```
'Debugbar' => Barryvdh\Debugbar\Facade::class,
```

Finally, run this command to generate Debugbar's config file:

[41] https://github.com/barryvdh/laravel-debugbar

```
1 php artisan vendor:publish --provider="Barryvdh\Debugbar\ServiceProvider"
```

Go ahead and view our app in the browser, you'll see a nice debug bar:



Debugbar

You may read the **official Debugbar documentation** at:

https://github.com/barryvdh/laravel-debugbar[42]

There are many other tools, extensions, and packages. If you have a good one, feel free to send us an email, I'll add it to the list.

It's time to create some tests!

Running tests with PHPUnit

When talking about a PHP testing framework, people usually think about **PHPUnit**.

PHPUnit is one of the most widely-used PHP testing frameworks. The great news is, Laravel 5 ships with **PHPUnit** out of the box. It's now a part of the Laravel core. That means we can write some unit tests right away without worrying about setting everything up. There are some convenient **helper methods** and **example files** for us to use as well.

In the root of your application, there is a file called **phpunit.xml**. This is the **PHPUnit's configuration** file.

[42] https://github.com/barryvdh/laravel-debugbar

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit backupGlobals="false"
         backupStaticAttributes="false"
         bootstrap="bootstrap/autoload.php"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false">
    <testsuites>
        <testsuite name="Application Test Suite">
            <directory>./tests/</directory>
        </testsuite>
    </testsuites>
    <filter>
        <whitelist>
            <directory suffix=".php">app/</directory>
        </whitelist>
    </filter>
    <php>
        <env name="APP_ENV" value="testing"/>
        <env name="CACHE_DRIVER" value="array"/>
        <env name="SESSION_DRIVER" value="array"/>
        <env name="QUEUE_DRIVER" value="sync"/>
    </php>
</phpunit>
```

By looking at the following:

```
    <testsuite name="Application Test Suite">
        <directory>./tests/</directory>
    </testsuite>
```

you may know that **all our test files** are placed in the **tests** directory.

Let's go to the **tests** directory, we should see two files:

TestCase.php:

```php
<?php

class TestCase extends Illuminate\Foundation\Testing\TestCase
{
    /**
     * The base URL to use while testing the application.
     *
     * @var string
     */
    protected $baseUrl = 'http://localhost';

    /**
     * Creates the application.
     *
     * @return \Illuminate\Foundation\Application
     */
    public function createApplication()
    {
        $app = require __DIR__.'/../bootstrap/app.php';

        $app->make(Illuminate\Contracts\Console\Kernel::class)->bootstrap();

        return $app;
    }
}
```

Basically, we don't need to worry about this file, it's just a **base class**. If we want to **write new tests**, simply extend this **TestCase** class.

If you want to use a different URL while testing your application, you may change the URL here.

ExampleTest.php:

```php
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
```

```
11      *
12      * @return void
13      */
14     public function testBasicExample()
15     {
16         $this->visit('/')
17              ->see('Laravel 5');
18     }
19 }
```

This is **a testing class**. In PHPUnit, we call it "**a test case**".

A **test case** is **a term** for **a class** that **contains different tests**. All the tests usually have **the same functionality**.

As mentioned above, we need to have our test class **extend** the **TestCase** class.

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

This is **a test**. As you see, it's just a method. If you write a **new method (a new test)**, your method **must be public** and you must start them with **test**. It's a naming convention.

By reading the **testBasicExample** test, can you guess what it does?

Very simple! It says: "Visit our home page (/) and see the words **Laravel 5**".

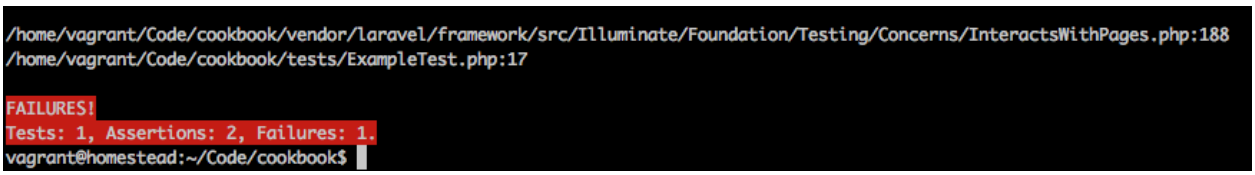
Now, let's try to run our first test!

Vagrant ssh into your homestead, navigate to your app, and run this command:

```
1  vendor/bin/phpunit
```

Because we don't have the words **Laravel 5** on our home page, **the test should fail**.

We should see **a red message**.



First test

Next, let's open our **home view** and find:

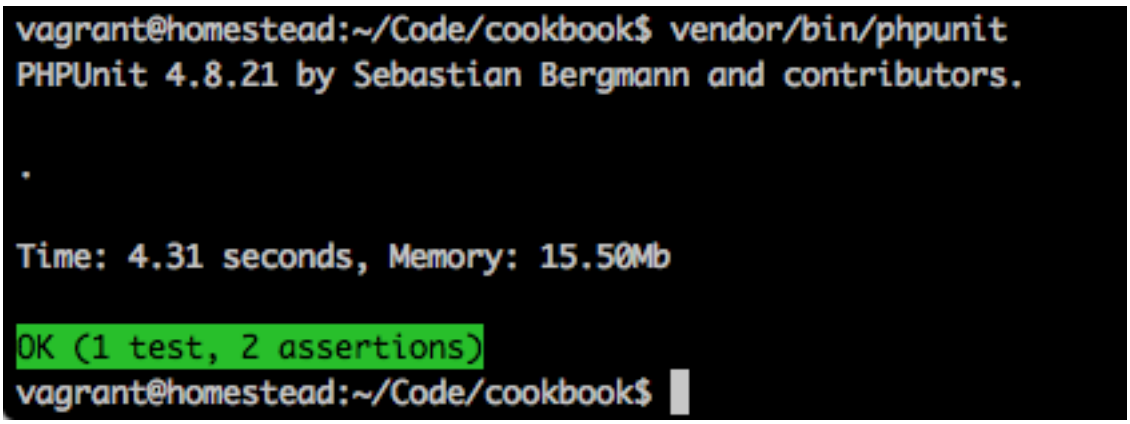
```
<div class="panel-heading">Dashboard</div>
```

Replace with:

```
div class="panel-heading">Laravel 5</div>
```

Run the test again:

```
vendor/bin/phpunit
```



Second test

Now our example test shows a passing test.

We should see **a green message**.

PHPUnit documentation and Laravel's PHPUnit methods

If you want to learn more about PHPUnit methods and how to use them. I encourage you to spend some time reading PHPUnit's documentation[43].

The PHPUnit's documentation is wonderful. It's just like a book and it's totally free.

Laravel also has some additional assertion methods for PHPUnit tests that we can use:

- **->assertResponseOk()**: Assert that the client response has an OK status code.
- **->assertResponseStatus($code)**: Assert that the client response has a given code.
- **->assertViewHas($key, $value = null)**: Assert that the response view has a given piece of bound data.
- **->assertViewHasAll(array $bindings)**; Assert that the view has a given list of bound data.

[43] https://phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.test-dependencies

- **->assertViewMissing($key)**: Assert that the response view is missing a piece of bound data.
- **->assertRedirectedTo($uri, $with = [])**: Assert whether the client was redirected to a given URI.
- **->assertRedirectedToRoute($name, $parameters = [], $with = [])**: Assert whether the client was redirected to a given route.
- **->assertRedirectedToAction($name, $parameters = [], $with = [])**: Assert whether the client was redirected to a given action.
- **->assertSessionHas($key, $value = null)**: Assert that the session has a given value.
- **->assertSessionHasAll(array $bindings)**: Assert that the session has a given list of values.
- **->assertSessionHasErrors($bindings = [], $format = null)**: Assert that the session has errors bound.
- **->assertHasOldInput()**: Assert that the session has old input.

I get these methods from the testing section of Laravel's documentation[44]. Be sure to check it out!

Writing our first PHPUnit test

To better understand PHPUnit, let's try to create **a new test case**.

To create a new test case, use the following command:

```
php artisan make:test BlogTest
```

Check the **tests** directory, we should see a new **BlogTest.php** file.

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class BlogTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testExample()
    {
```

[44] https://laravel.com/docs/master/testing

```
16          $this->assertTrue(true);
17      }
18  }
```

We don't need the **testExample** method, so just **remove it** and write our new test:

```
1  public function testBlogResponseIsValid()
2  {
3      $this->visit('/blog')
4          ->assertResponseOk();
5  }
```

As you see, the name of our test is **testBlogResponseIsValid**.

Our blog's response should have **an OK status code (200)**. Of course, **the test would fail** if **the response is not valid** (return other status code).

This is a pretty standard process to us by now. Run PHPUnit again:

```
1  vendor/bin/phpunit
```

Open our **BlogController**, if the response has **the 200 status code**...

```
1      $response = Response::json($posts,200);
2      return $response;
```

...the test case is **green** and everything **passes**.

If we **modify the response**, we should see:

```
There was 1 failure:

1) ExampleTest::testBlogResponseIsValid
A request to [http://localhost/blog] failed. Received status code [404].

/home/vagrant/Code/cookbook/vendor/laravel/framework/src/Illuminate/Foundation/Testing/Concerns/InteractsWithPages.php:166
/home/vagrant/Code/cookbook/vendor/laravel/framework/src/Illuminate/Foundation/Testing/Concerns/InteractsWithPages.php:64
/home/vagrant/Code/cookbook/vendor/laravel/framework/src/Illuminate/Foundation/Testing/Concerns/InteractsWithPages.php:45
/home/vagrant/Code/cookbook/tests/ExampleTest.php:22

FAILURES!
Tests: 2, Assertions: 3, Failures: 1.
```

Second test

Another thing to note is that we can use **multiple methods** to create our test:

```
public function testBlogResponseIsValid()
{
    $this->visit('/')
        ->click('Blog')
        ->see('current_page')
        ->assertResponseOk();
}
```

Recipe 9 Wrap Up

Tag: Version 0.7 - Recipe 9[45]

Up to this point, we have learned some useful testing techniques and created several unit tests.

You should be able to **test** your application effectively now.

Remember that, the more tests that you create, the more your testing skills will be improved.

Practice makes perfect.

Recipe 10 - Writing APIs with Laravel

What will we learn?

This recipe shows you how to build a **REST API** on top of our Laravel application. We can then use our app as **a backend service** for **mobile applications** or **AJAX-based client applications**.

What is REST API?

API stands for **Application Program Interface**. Simply put, an API is an **interface** for coders to communicate with applications.

API acts just like a **middleware**. When we send **requests** to an API, it checks the requests. If the requests are allowed, data will be returned. **Proper responses** are also returned to let us know the **result** of our requests.

Using APIs, we can effectively create a backend service that supports many types of applications. Developers can change the look and feel of their apps frequently without worrying about breaking the apps.

REST stands for **Representational State Transfer**. It's a style of web architecture. Basically, **REST** is just a set of **agreements and constraints** on how components should work together.

[45] https://github.com/LearningLaravel/cookbook/releases/tag/v0.7

When APIs use REST architecture, they are called **REST APIs** (aka **RESTful APIs**).

A typical REST API has these following **constraints**[46]:

- **Client - server**: Servers (back end) and clients (front end) can be developed independently.
- **Stateless**: Session state should be stored on the client. Client data should not be stored on the server between requests.
- **Cacheable**: Client can cache responses to improve scalability and performance.

REST API use **HTTP requests** to communicate with the servers. Each request specifies a certain **HTTP verb** in the request header, such as:

```
1 GET /posts HTTP/1.1
```

There are many **HTTP verbs**, but the most **popular ones** for building **REST APIs** are:

- **GET**
- **POST**
- **PUT**
- **DELETE**

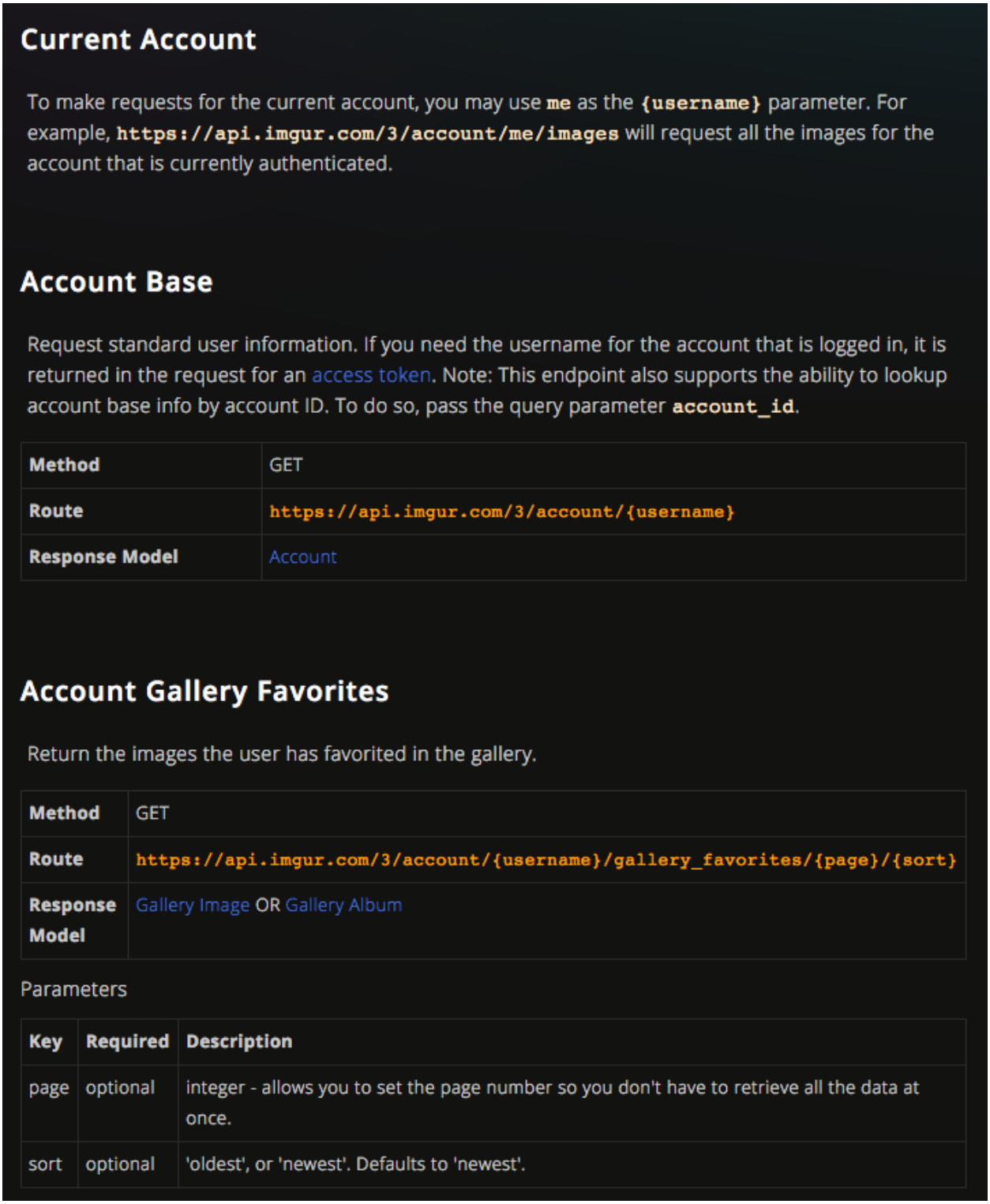
Creating an API endpoint

The **API endpoint** is **a URL** that we use to connect and send requests to our application. Every **dataset** or **individual data record** of our application has its own **endpoint**.

Example **Imgur**[47] API's endpoints:

[46] https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch5-4

[47] http://www.imgure.com



Current Account

To make requests for the current account, you may use **me** as the **{username}** parameter. For example, **https://api.imgur.com/3/account/me/images** will request all the images for the account that is currently authenticated.

Account Base

Request standard user information. If you need the username for the account that is logged in, it is returned in the request for an access token. Note: This endpoint also supports the ability to lookup account base info by account ID. To do so, pass the query parameter **account_id**.

Method	GET
Route	**https://api.imgur.com/3/account/{username}**
Response Model	Account

Account Gallery Favorites

Return the images the user has favorited in the gallery.

Method	GET
Route	**https://api.imgur.com/3/account/{username}/gallery_favorites/{page}/{sort}**
Response Model	Gallery Image OR Gallery Album

Parameters

Key	Required	Description
page	optional	integer - allows you to set the page number so you don't have to retrieve all the data at once.
sort	optional	'oldest', or 'newest'. Defaults to 'newest'.

Imgur API

In this section, let's move onto creating **a new endpoint** that **lists all blog posts**.

First, we will add a new route. Open **routes.php** and add:

```
Route::resource('posts', 'PostsController');
```

As you see, we don't use **Route::get** or **Route::post** here, we use **Route:resource**. In Laravel, this is a **resourceful route**.

This route tells Laravel to **create multiple routes** to handle a variety of **RESTful actions** on the **posts** resource.

Simply put, instead of creating multiple routes manually:

```
Route::get('posts', 'PostsController@index');
Route::post('posts', 'PostsController@store');
...
```

we may just use **a resourceful route** and Laravel will automatically generate all the related routes for us.

Once again, when having a new route, we may need a new controller.

We don't have the **PostsController** yet. Let's create one by running this **Artisan command**:

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

By adding a **–resource** flag, Laravel generates a new **resource controller** for us, instead of a plain controller.

> **Note**: If you're using older versions of Laravel, a resource controller is generated by default.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

class PostsController extends Controller
{
```

```
12      /**
13       * Display a listing of the resource.
14       *
15       * @return \Illuminate\Http\Response
16       */
17      public function index()
18      {
19          //
20      }
21
22      /**
23       * Show the form for creating a new resource.
24       *
25       * @return \Illuminate\Http\Response
26       */
27      public function create()
28      {
29          //
30      }
31
32      /**
33       * Store a newly created resource in storage.
34       *
35       * @param  \Illuminate\Http\Request  $request
36       * @return \Illuminate\Http\Response
37       */
38      public function store(Request $request)
39      {
40          //
41      }
42
43      /**
44       * Display the specified resource.
45       *
46       * @param  int  $id
47       * @return \Illuminate\Http\Response
48       */
49      public function show($id)
50      {
51          //
52      }
53
```

```
54      /**
55       * Show the form for editing the specified resource.
56       *
57       * @param  int  $id
58       * @return \Illuminate\Http\Response
59       */
60      public function edit($id)
61      {
62          //
63      }
64
65      /**
66       * Update the specified resource in storage.
67       *
68       * @param  \Illuminate\Http\Request  $request
69       * @param  int  $id
70       * @return \Illuminate\Http\Response
71       */
72      public function update(Request $request, $id)
73      {
74          //
75      }
76
77      /**
78       * Remove the specified resource from storage.
79       *
80       * @param  int  $id
81       * @return \Illuminate\Http\Response
82       */
83      public function destroy($id)
84      {
85          //
86      }
87  }
```

Believe it or not, by just running two commands, we have all **RESTful routes and actions** that we need to make **an API endpoint**.

To make sure that we have all the posts' routes, you can list all **your application's routes** by running the following command:

```
1  php artisan route:list
```

```
GET|HEAD  | posts                  | posts.index   | App\Http\Controllers\PostsController@index
POST      | posts                  | posts.store   | App\Http\Controllers\PostsController@store
GET|HEAD  | posts/create           | posts.create  | App\Http\Controllers\PostsController@create
GET|HEAD  | posts/{posts}          | posts.show    | App\Http\Controllers\PostsController@show
PUT|PATCH | posts/{posts}          | posts.update  | App\Http\Controllers\PostsController@update
DELETE    | posts/{posts}          | posts.destroy | App\Http\Controllers\PostsController@destroy
GET|HEAD  | posts/{posts}/edit     | posts.edit    | App\Http\Controllers\PostsController@edit
```

Posts' routes

Here is a **list of actions** handled by the **generated resource controller**:

Verb	Path	Action	Route Name
GET	`/posts`	index	posts.index
GET	`/posts/create`	create	posts.create
POST	`/posts`	store	posts.store
GET	`/posts/{postid}`	show	posts.show
GET	`/posts/{postid}/edit`	edit	posts.edit
PUT/PATCH	`/posts/{postid}`	update	posts.update
DELETE	`/posts/{postid}`	destroy	posts.destroy

Posts' routes

If you want to learn more about **RESTful resource controllers**, you may take a look at the official documentation[48].

Next, open **PostsController** and update the **index action** as follows:

[48] https://laravel.com/docs/master/controllers#restful-resource-controllers

```
public function index()
{
    $posts = Post::all();
    $response = Response::json($posts,200);
    return $response;
}
```

Alternatively, you may use the following:

```
public function index()
{
    $posts = Post::all();
    return $posts;
}
```

Go ahead and visit http://cookbook.app/posts[49], you should see all the blog posts in JSON format:


```
[
 - {
      id: 1,
      title: "Modi non architecto distinctio dolorem provident.",
      content: "In inventore quo commodi cumque placeat eos. Ab iste fugiat incidunt tenetur id. Soluta excepturi sed omnis sunt. Fuga nihil unde quisqua
      molestiae. Neque tenetur impedit sed assumenda possimus eaque nihil. Repellat et est ut ex. Dolorem magnam accusamus perspiciatis impedit saepe pla
      adipisci modi. Voluptatem sit adipisci ullam autem sint. Expedita numquam eaque adipisci vero et rerum. Magni ducimus fugit optio perferendis. Aspe
      tenetur quia eius tempora architecto sit modi. Nihil voluptas laborum et rerum deserunt est vel. Eius qui vel nemo qui itaque sit. Occaecati explic
      aut minus tempore tempora debitis. Repellat ipsum dolor voluptatem aperiam. Unde itaque distinctio tempora rerum dolores perspiciatis non. Quam dol
      slug: "modi-non-architecto-distinctio-dolorem-provident",
      status: 1,
      user_id: 1,
      created_at: "2016-02-27 19:55:47",
      updated_at: "2016-02-27 19:55:47"
  },
 - {
      id: 2,
      title: "Adipisci provident et aut laborum et.",
      content: "Expedita similique est ea dignissimos quisquam error nulla odit. Qui omnis libero sit a sed iste impedit facere. Pariatur ut natus porro
      et iste. Rem autem numquam qui voluptatem. Voluptate eius officiis voluptates quaerat inventore. Dignissimos dolor nobis placeat vel ipsum eveniet.
      veritatis perferendis. Nesciunt incidunt itaque vitae itaque sit consequuntur quisquam. Labore incidunt dolor praesentium dignissimos ut. Iusto cor
      minus veritatis. Laudantium qui soluta odio autem id dolorem. Vitae nisi id est similique. Facere aut hic et quia quis excepturi. Corrupti ea dolor
      slug: "adipisci-provident-et-aut-laborum-et",
      status: 1,
      user_id: 2,
      created_at: "2016-02-27 19:55:47",
      updated_at: "2016-02-27 19:55:47"
  },
 - {
      id: 3,
      title: "Sint voluptas ab veritatis iste quaerat.",
      content: "Dolores recusandae vitae totam laborum aliquid. Ipsa facere error neque et. Velit qui harum fugit aut. Omnis et et maiores molestiae qui
      ipsam velit architecto voluptates velit. Consequatur impedit nemo vel odit. Sit fuga sint sed voluptatem numquam id. Velit recusandae nobis expedit
      praesentium. Illo nesciunt nam non qui. Soluta architecto iste perspiciatis necessitatibus provident est. Nisi sunt sed est qui. Quia voluptate ut
      commodi voluptatem. Consequuntur consequuntur doloremque corrupti non dicta voluptas molestias. Quia et aperiam ad dolores repellat atque quidem su
      numquam alias fugiat id sit. Et maiores sint ducimus quisquam illum quod fugit. Id vitae quisquam accusamus sint. Deleniti nulla sunt cum molestiae
      dolores ut. Cum quia laboriosam quasi est dolores. Quam architecto voluptas impedit laboriosam. Veniam sint reiciendis ut autem quis velit sapiente
      velit et quo. Commodi fugiat beatae sed ducimus. Asperiores inventore sed maxime et et repudiandae veniam. Occaecati quia unde non et ad voluptas l
      officiis.",
      slug: "sint-voluptas-ab-veritatis-iste-quaerat",
      status: 1,
      user_id: 4,
      created_at: "2016-02-27 19:55:47",
      updated_at: "2016-02-27 19:55:47"
  },
```


posts route

Great! We have our first **API endpoint**!

However, we need to do one more thing.

[49] http://cookbook.app/posts

Our API will likely change over time. One day, we may need to change our code significantly to add more features or restructure our application. Therefore, we should **version our API** from the beginning.

As you know, Laravel 5.2 has introduced a new feature called **middleware groups** and we've used the **web middleware group** in previous recipes. Let's open the **app/Http/Kernel.php** file:

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

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

As you see, there is another **middleware group** called **api**. When building APIs with Laravel 5.2 (or newer), it's better to utilize this middleware group.

Open **routes.php**, add **our posts resourceful route** into the **api middleware group**:

```
Route::group(['prefix' => 'api/v1', 'middleware' => 'api'], function(){
    Route::resource('posts', 'PostsController');
});
```

In order to **version our API**, we also add the **prefix (api/v1)** to the group. Now we can access our first API endpoint at http://cookbook.app/api/v1/posts[50].

In the future, if we want to develop a new version of our APIs, all we have to do is creating a new middleware group!

Another thing to note, you may see the words **throttle:60,1** in the **api middleware group**:

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

Well, it's the **API rate limiting** feature of Laravel. If a user (or a bot) is hitting our **API endpoint** a **million times** a **minute**, our application would be still running fine. When they try to make too many requests in a short time, they will get this message:

[50] http://cookbook.app/api/v1/posts

```
429: Too Many Attempts
```

The **default throttle** allows users to make **60 requests** per minute. They can't access our application for **one minute** if they hit the limit.

Feel free to change the limit to whatever you want.

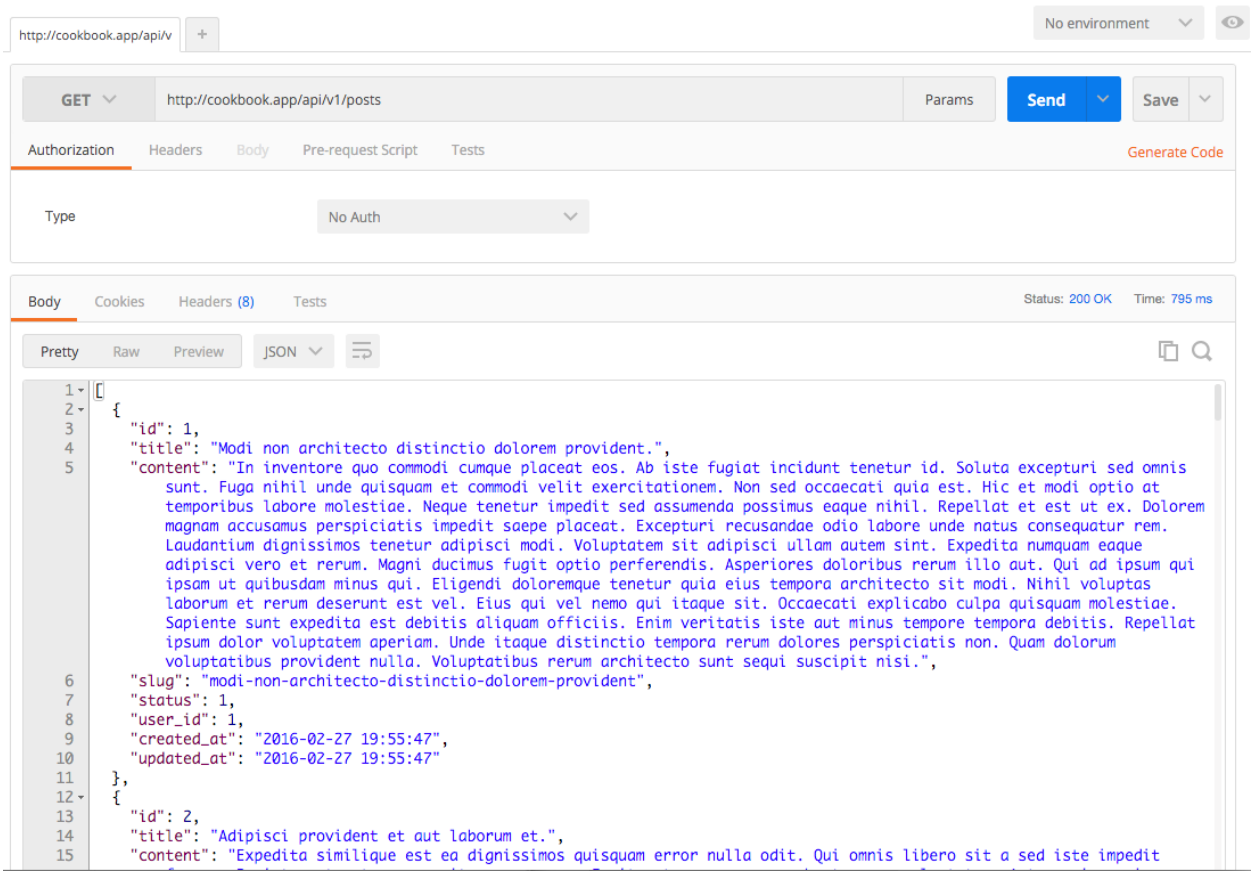
Using Postman to test our API

As mentioned before, when working with APIs, we should use **Postman**[51] - a Google Chrome extension.

Postman has many features and amazing interface that help us to test our APIs faster. Using Postman, we can send **GET, POST, PUT, PATCH, and DELETE** request to test our APIs effectively.

Once installed, open **Postman** and choose **GET** (which means GET request).

Enter the **URL** of our **API** (http://cookbook.app/api/v1/posts) into the input box. Finally, click the **blue Send button**.



Postman

[51] https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en

Pretty simple so far, right?

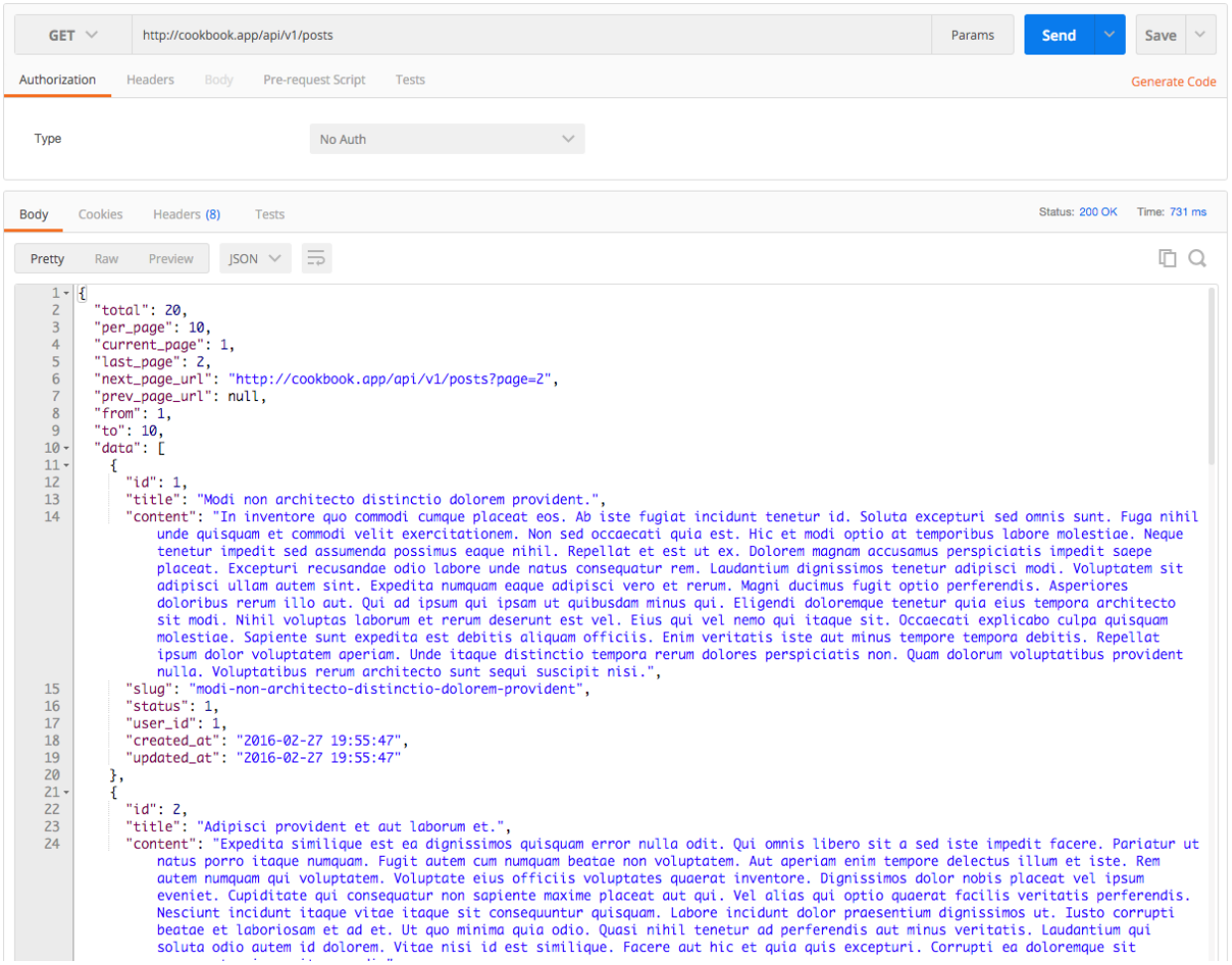
The output is very useful. We can see the **status code (200)**, the **execution time (795 ms)** and our **posts data** in pretty JSON format.

Pagination

Usually, we don't want to display all posts at once. As you may have guessed, we can easily **paginate** our posts by using the **paginate method**.

Open **PostsController** and update the **index action** as follows:

```
1  public function index()
2  {
3      $posts = Post::paginate(10);
4      $response = Response::json($posts,200);
5      return $response;
6  }
```



Pagination

Status Code

After sending requests, we usually get back a message with a **status code**. This status code is called **HTTP Response Code**. It's very important to understand those status codes because they are used to express various **success** and **failure states** of our application.

You may view a list of response codes here:

https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html[52]

These are the most common ones:

- **100**: Continue - The client should continue with its request.
- **200**: OK - The request has succeeded.

[52] https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

- **201:** Created - The request has been fulfilled and resulted in a new resource being created.
- **202:** Accepted - The request has been accepted for processing, but the processing has not been completed.
- **204:** No Content - The server has fulfilled the request but does not need to return an entity-body, and might want to return updated metainformation.
- **301:** Moved Permanently - The requested resource has been assigned a new permanent URI and any future references to this resource should use one of the returned URIs.
- **302:** Moved Temporarily - The requested resource resides temporarily under a different URI.
- **400:** Bad Request - The request could not be understood by the server due to malformed syntax.
- **401:** Unauthorized - The request requires user authentication.
- **403:** Forbidden - The server understood the request, but is refusing to fulfill it.
- **404:** Not Found - The server has not found anything matching the Request-URI.
- **500:** Internal Server Error - The server encountered an unexpected condition which prevented it from fulfilling the request.

Getting a single post

Now that you know about the response code. Let's try to send a **GET request** to grab a single post.

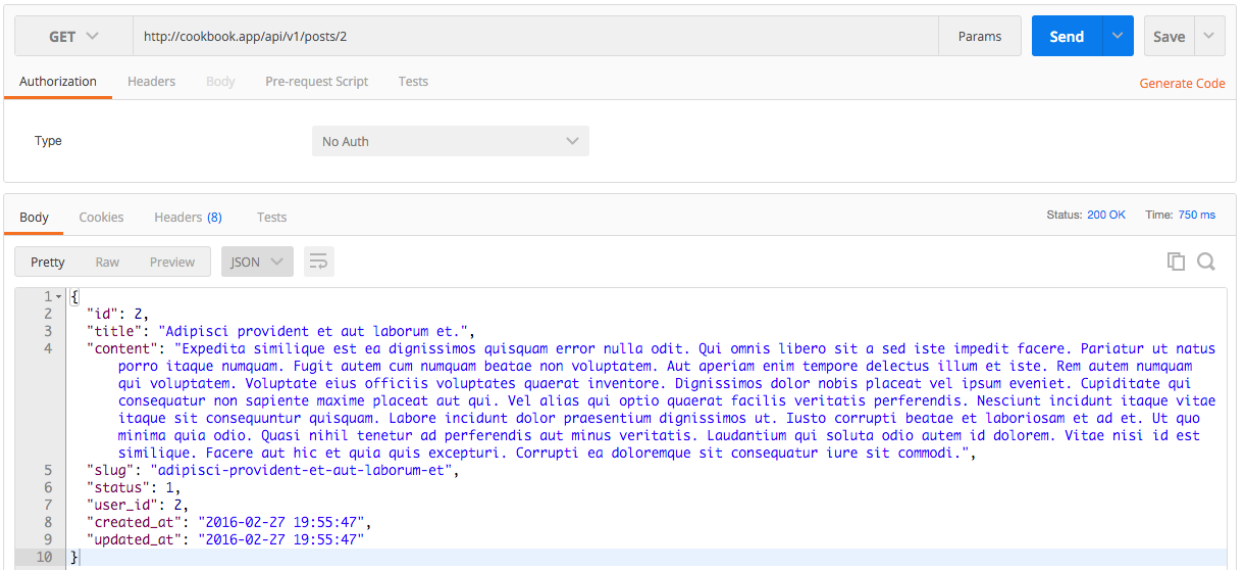
The **API endpoint** should be **/posts/{postid}** and the action that we use is **show**.

Open **PostsController** and update the **show** action as follows:

```
public function show($id)
{
    $post= Post::find($id);

    if(!$post){
        $response = Response::json([
            'error' => [
                'message' => 'This post cannot be found.'
            ]
        ], 404);
        return $response;
    }

    $response = Response::json($post
        , 200);
    return $response;
}
```

Open **Postman**. Enter this URL: http://cookbook.app/api/v1/posts/2[53], and hit **Send**.

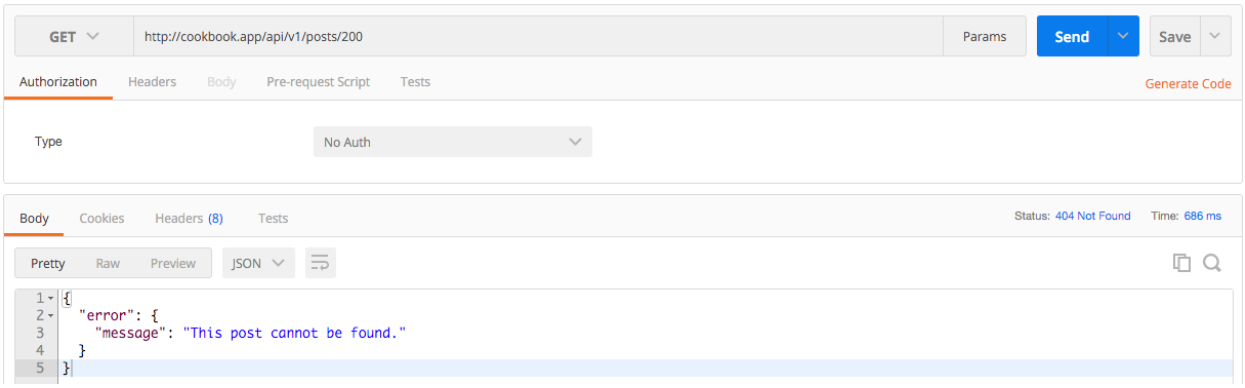
This is how we can get a post by using its id.



1 single post

Cool! Now we have **the post** and the **status code** is **200**. Everything's working fine.

If we enter a **wrong id**, we should get this message:



Wrong id

Now the **status code** is **404**. We know that the post **doesn't exist**.

Adding a new post using POST request

Just like we created the API endpoint to grab data, we will use **POST request** to insert a new post.

The API endpoint should be **/posts** and the action that we use is **store**.

[53] http://cookbook.app/api/v1/posts/2

Open **PostsController**, update the **store** action as follows:

```
public function store(Request $request)
{
    if((!$request->title) || (!$request->content)){

        $response = Response::json([
            'error' => [
                'message' => 'Please enter all required fields'
            ]
        ], 422);
        return $response;
    }

    $post = new Post(array(
        'title' => $request->title,
        'content' => $request->content,
        'slug' => Str::slug($request->title, '-'),
    ));

    $post->save();

    $response = Response::json([
        'message' => 'The post has been created succesfully',
        'data' => $post,
    ], 201);

    return $response;
}
```

Open our **Post model (app/Post.php)**:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $fillable = [
        'title', 'content', 'slug', 'status',
```

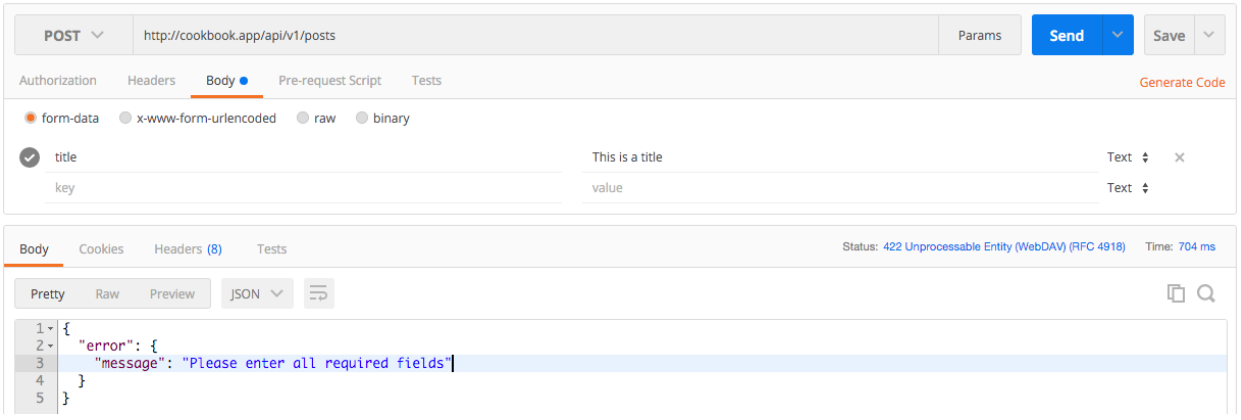
```
    ];

}
```

The **$fillable property** should have the following: **title**, **content**, **slug**, **status**.

Be sure that we've told Laravel to use **Str** and **Request**:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

use App\Post;
use Response;
use Illuminate\Support\Str;

class PostsController extends Controller
{
```

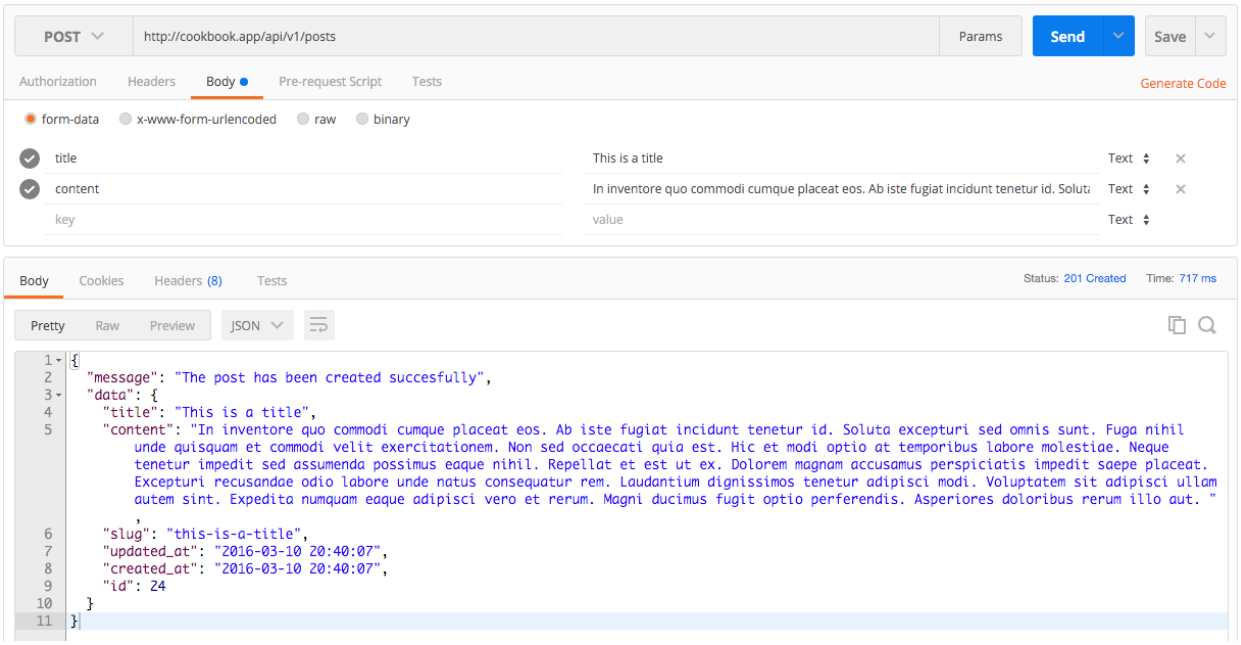
Now open **Postman** and try to **send a POST request**:



Error message when creating post

If we only enter the title, there should be an **error message**. The **status code** is **422**.

If we enter everything correctly, we should be able to create a new post:



The post is created successfully

Updating a post

Now that we have created a new post. Let's look at how we can use **PUT request** to **update a post**.

The API endpoint should be **/posts/{postid}** and the action that we use is **update**.

Open **PostsController**, update the **update** action as follows:

```
public function update(Request $request, $id)
{
    if((!$request->title) || (!$request->content)){

        $response = Response::json([
            'error' => [
                'message' => 'Please enter all required fields'
            ]
        ], 422);
        return $response;
    }

    $post = Post::find($id);
    $post->title = $request->title;
    $post->content = $request->content;
    $post->slug = Str::slug($request->title, '-');
```

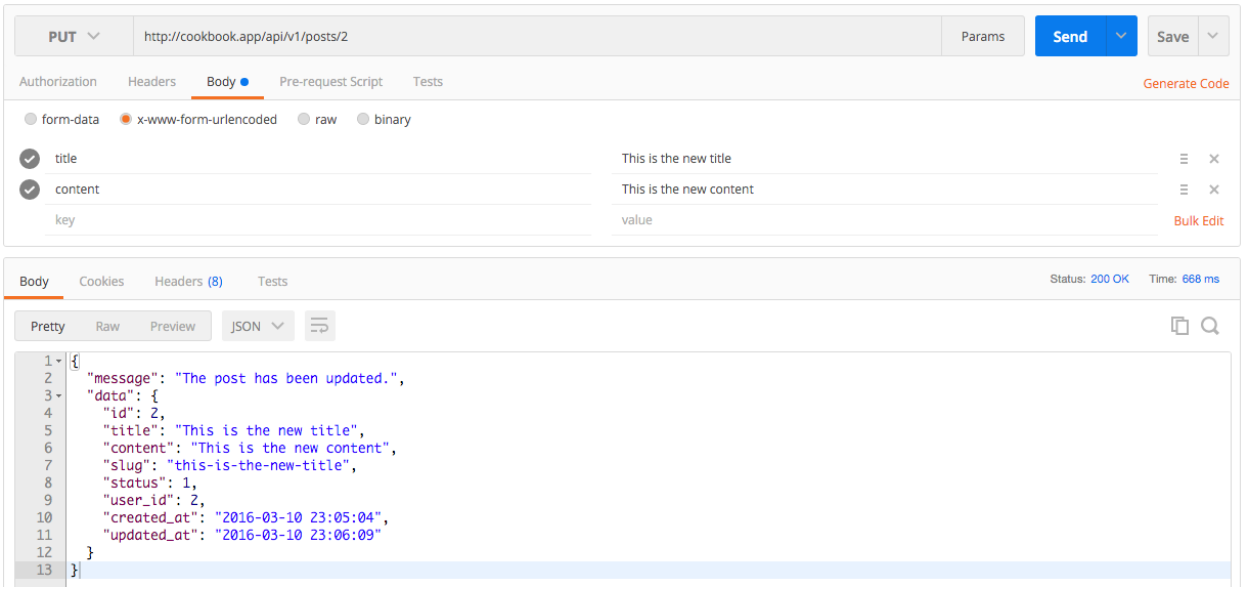
```
17      $post->save();
18
19      $response = Response::json([
20          'message' => 'The post has been updated.',
21          'data' => $post,
22      ], 200);
23
24      return $response;
25  }
```

Now open **Postman**. Select **PUT**.

Enter this URL: http://cookbook.app/api/v1/posts/2[54].

Choose **x-www-form-urlencoded**. Enter the title and content of your post.

Finally, hit **Send**.



The post is updated

The post is now updated!

Deleting a post

In our last section, we used **PUT** request to update our posts. We'll follow the same process that we used to **delete a post**, but this time, we use **DELETE** request.

The API endpoint should be **/posts/{postid}** and the action that we use is **destroy**.

Open **PostsController** and update the **destroy action** as follows:

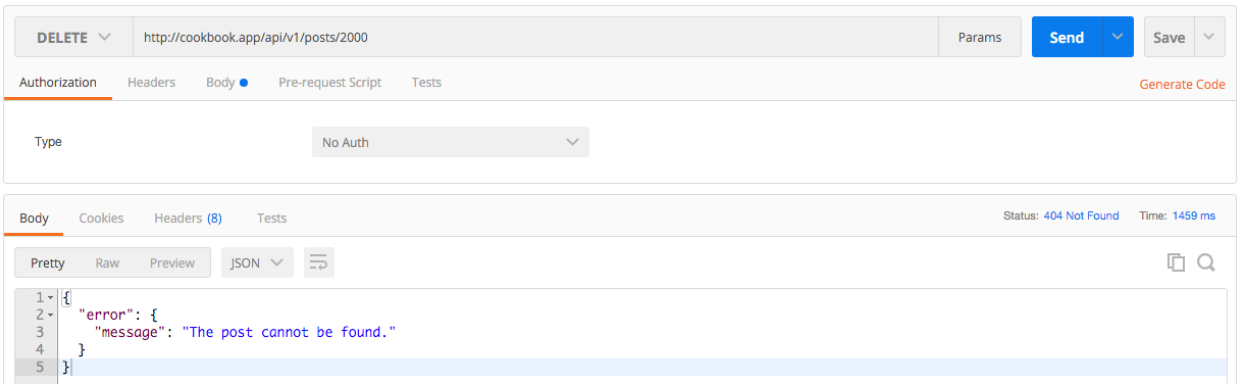
[54] http://cookbook.app/api/v1/posts/2

```php
public function destroy($id)
{
    $post = Post::find($id);

    if(!$post) {
        $response = Response::json([
            'error' => [
                'message' => 'The post cannot be found.'
            ]
        ], 404);

        return $response;
    }

    Post::destroy($id);

    $response = Response::json([
        'message' => 'The post has been deleted.'
    ], 200);

    return $response;
}
```

Now open **Postman**. Select **DELETE**.

Enter this URL: http://cookbook.app/api/v1/posts/2000[55].

Hit **Send**.

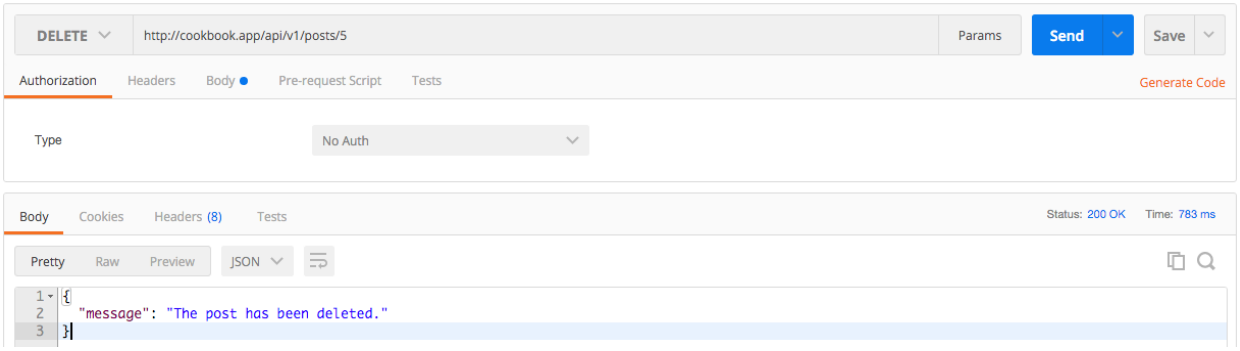


The post is updated

Because the post's id is **2000**, we should receive an error message.

[55] http://cookbook.app/api/v1/posts/2000

Now let's try to run again, but this time, enter **5** as the **post's id**.



The post is updated

The post is now deleted successfully.

Adding CORS

CORS[56] stands for **Cross-Origin Resource Sharing**, which is a mechanism that allows modern browsers to send and receive restricted data (images, fonts, files, etc.) from **a domain** other than the one that made the request.

Simply put, if we **don't enable CORS**, we **can't access our API** from other applications.

To enable CORS, we may build a custom middleware[57] to add CORS header to our response. This is a simple method, but the simplest one is to use a popular package called laravel-cors[58].

To install the package, run this **Composer** command:

```
composer require barryvdh/laravel-cors
```

Once installed, open **config/app.php** and add the **CorsServiceProvider** to our **providers** array:

```
Barryvdh\Cors\ServiceProvider::class,
```

Next, open **routes.php** and add the **cors middleware** to our **api middleware group**:

```
Route::group(['prefix' => 'api/v1', 'middleware' => ['api', 'cors']], function(){
    Route::resource('posts', 'PostsController');
});
```

That's it!

Our API is now working properly!

[56] https://en.wikipedia.org/wiki/Cross-origin_resource_sharing
[57] http://learninglaravel.net/laravel-51-easily-enable-cors
[58] https://github.com/barryvdh/laravel-cors

Chapter 10 Wrap-up

> **Tag:** Version 0.8 - Recipe 10[59]

Congratulations! There we have it! A Laravel application that can be used as a backend service for mobile applications or AJAX-based websites.

Now let's move onto Chapter 2. We will learn some front end recipes to improve user experience.

In the future, I'll add more backend recipes, so that we can learn more about API and other Laravel features.

This is just a beginning.

[59] https://github.com/LearningLaravel/cookbook/releases/tag/v0.8

Chapter 2: Front End Recipes

Introduction

Whether you are a beginner or intermediate web developer, if you wish to make good interactive web applications, then this chapter is for you.

In this chapter, you'll be getting some recipes about front-end web technologies and popular front-end tools. These recipes cover best practices and modern techniques for front-end development such as: integrating Twitter Bootstrap, AJAX loading, notifications, file uploads, cropping images and many more.

By the end, you should have a better understanding of how to work with AJAX, Jquery, front end frameworks and responsive design. You can apply these techniques to build beautiful applications and add that interactivity to any site you work on.

List Of Recipes

Frontend recipes

- Recipe 201 - Notifications
- Recipe 202 - Integrating Buttons With Built-in Loading Indicators
- Recipe 203 - Create A Registration Page Using AJAX and jQuery
- Recipe 204 - Create A Login Page Using AJAX And jQuery
- Recipe 205 - Upload Files Using AJAX And jQuery
- Recipe 206 - Cropping Images Using jQuery

(More recipes will be added later)

Recipe 201 - Notifications

What will we learn?

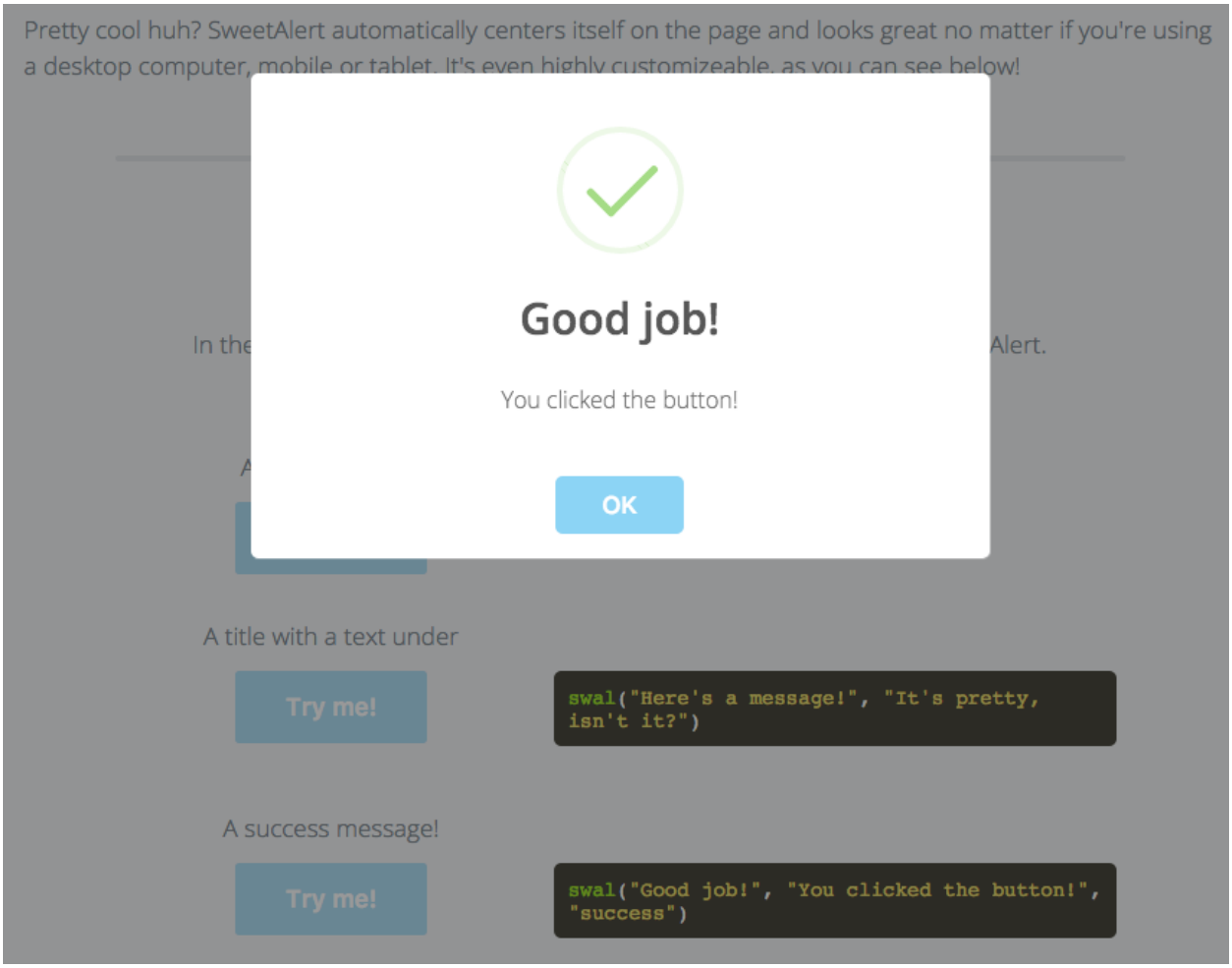
This recipe shows you how to integrate notifications into your Laravel application.

Say hi to Sweet Alert

Nowadays, notifications become a very important functionality of our modern applications. By integrating good looking notifications into our system, we will attract more users' attention and our app will definitely look nicer.

There are many notifications libraries, but the most popular ones are: HumanJS[60], HubSpot Messaging Library[61] and Sweet Alert[62].

This recipe will focus on integrating **Sweet Alert** - which is an amazing library that aims to replace JavaScript's alert and prompt features.



SweetAlert

[60] http://wavded.github.io/humane-js
[61] http://github.hubspot.com/messenger/docs/welcome/
[62] http://t4t5.github.io/sweetalert

Installing Sweet Alert

Installing Sweet Alert is pretty easy! There is a Laravel package called Easy Sweet Alert Messages for Laravel[63]. We can use this package to easily show Sweet Alert notifications in our Laravel application.

First, open our **composer.json** file and add the following code into the **require** section:

```
1     "uxweb/sweet-alert": "~1.1"
```

Next, run **composer update** to install the package.

Open **config/app.php**, add the following code to the **providers** array:

```
1 UxWeb\SweetAlert\SweetAlertServiceProvider::class,
```

Then find the **aliases** array and add:

```
1     'Alert' => UxWeb\SweetAlert\SweetAlert::class,
```

Next, download the latest version of Sweet Alert[64].

> **Note:** You may also use Sweet Alert 2[65].

Once downloaded, **unzip (decompress)** the file and go to **sweetalert-master/dist**.

Copy the **sweetalert.min.js** file to your **public/js** directory. Create the **js** directory if you don't have one.

Copy the **sweetalert.css** file to your **public/css** directory. Create the **css** directory if you don't have one.

Last step, open our **master layout** (resources/views/layouts/app.blade.php). Find:

```
1 <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css\
2 /bootstrap-theme.min.css">
```

Add below:

[63] https://github.com/uxweb/sweet-alert
[64] https://github.com/t4t5/sweetalert/archive/master.zip
[65] https://github.com/limonte/sweetalert2

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

Find:

```
</body>
```

Add above:

```
    <script src="/js/sweetalert.min.js"></script>
    @include('sweet::alert')
```

Our master layout should look like this:

```
<html>
<head>
    <title> @yield('title') </title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6\
/css/bootstrap.min.css">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6\
/css/bootstrap-theme.min.css">
    <link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.4.0/css/fo\
nt-awesome.min.css" rel='stylesheet'
    type='text/css'>
    <link rel="stylesheet" href="/css/sweetalert.css">

    <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.mi\
n.js"></script>
</head>
<body>

@include('shared.navbar')

@yield('content')

    <script src="/js/sweetalert.min.js"></script>
    @include('sweet::alert')
</body>
</html>
```

Sweet Alert is now ready to use!

If we want to customize the alert message partial, run:

```
php artisan vendor:publish
```

A **Sweet Alert view** is now generated in our **resources/views/vendor/sweet/** directory.

You can change the sweet alert configuration to your liking. Available options can be found at the Sweet Alert documentation[66].

Our first Sweet Alert message

Here's the moment we've been waiting for. Let's create our first Sweet Alert notification.

Open **routes.php** and find:

```
return view('home');
```

Add above:

```
Alert::info('Welcome to our website', 'Hi! This is a Sweet Alert message!');
```

Done! Head over to our home page and refresh the page:

[66] http://t4t5.github.io/sweetalert

Our first Sweet Alert message

This is how we show our first notification! Very simple, isn't it?

We've just used **Sweet Alert Facade** to display the notification. Alternatively, we may use **Sweet Alert Helper** to accomplish the same result:

Find:

```
1  Alert::info('Welcome to our website', 'Hi! This is a Sweet Alert message!');
```

Replace with:

```
1  alert()->info('Welcome to our website', 'Hi! This is a Sweet Alert message');
```

Here is a list of Facade's methods that we can use:

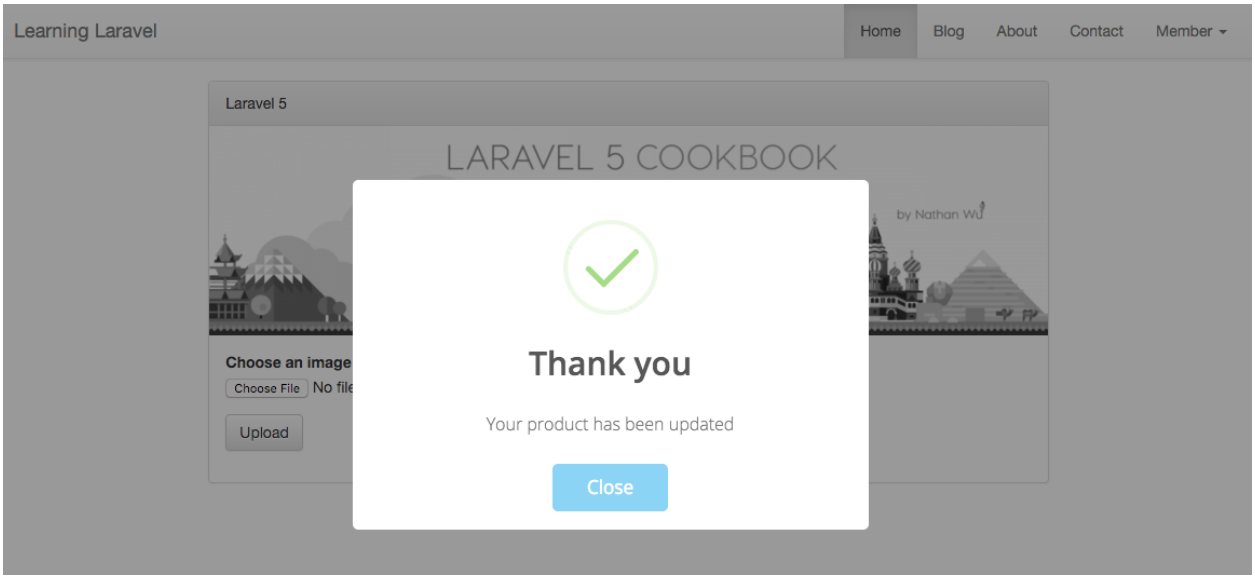
```php
Alert::message('Message', 'Optional Title');
Alert::basic('Basic Message', 'Mandatory Title');
Alert::info('Info Message', 'Optional Title');
Alert::success('Success Message', 'Optional Title');
Alert::error('Error Message', 'Optional Title');
Alert::warning('Warning Message', 'Optional Title');

Alert::basic('Basic Message', 'Mandatory Title')->autoclose(3500);

Alert::error('Error Message', 'Optional Title')->persistent('Close');
```

A list of Helper's methods:

```php
alert()->message('Message', 'Optional Title');
alert()->basic('Basic Message', 'Mandatory Title');
alert()->info('Info Message', 'Optional Title');
alert()->success('Success Message', 'Optional Title');
alert()->error('Error Message', 'Optional Title');
alert()->warning('Warning Message', 'Optional Title');

alert()->basic('Basic Message', 'Mandatory Title')
    ->autoclose(3500);

alert()->error('Error Message', 'Optional Title')
    ->persistent('Close');
```

Now let's try to show different notifications:

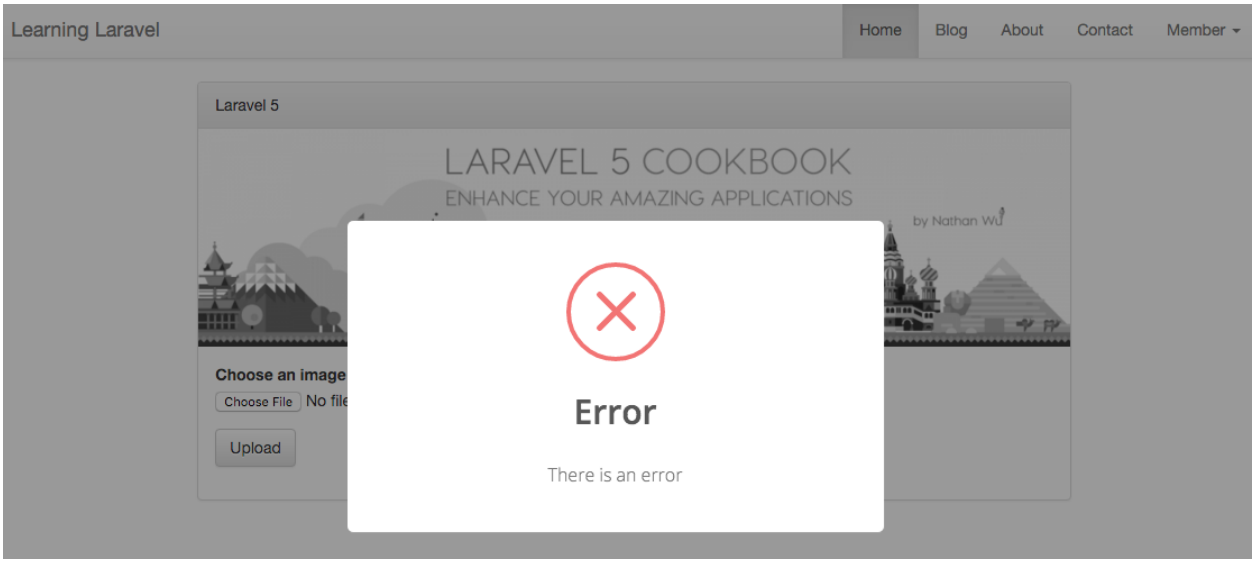
```php
alert()->success('Your product has been updated', 'Thank you')
    ->persistent('Close');
```



A successful notification

When adding **persistent('Your Custom Text')**, users must click the button to close the notification.

```
1  Alert::error('There is an error', 'Error')->autoclose(2000);
```



An error notification

When using **autoclose('time')**, the notification will be closed automatically after the defined time has passed.

Recipe 201 Wrap-up

Tag: Version 0.9 - Recipe 201[67]

As you see, Sweet Alert is really a good package.

Using the techniques above will be a good foundation to build beautiful notifications for our applications.

In the next recipes, we will be using Sweet Alert to provide textual feedback to our users.

Recipe 202 - Integrating Buttons With Built-in Loading Indicators

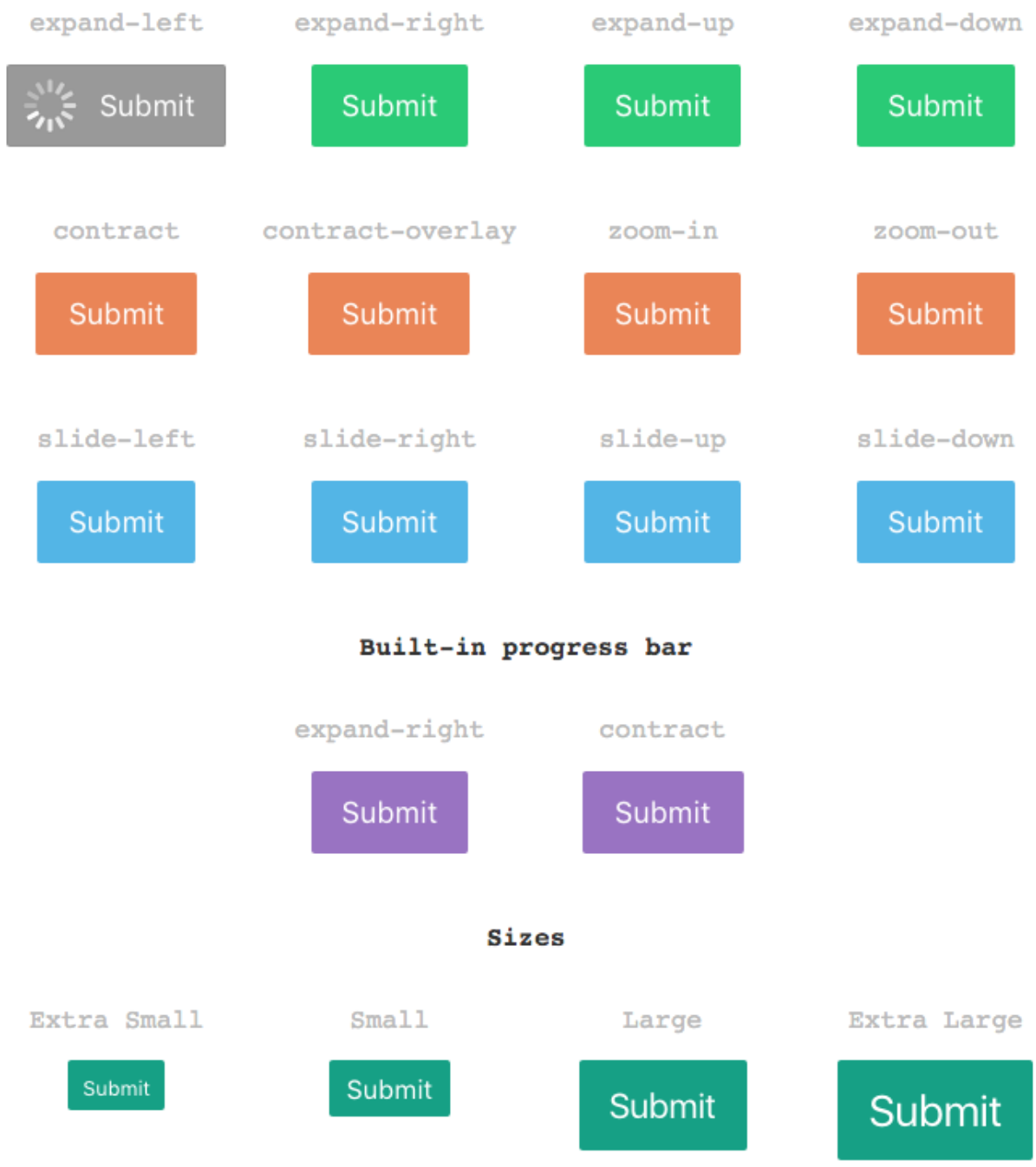
What will we learn?

This recipe shows you how to place a spinner directly inside a button and create some cool buttons with loading indicators.

Installing Ladda

When building modern applications, it's important to provide some creative loading effects to improve user experience. In this section, I'll show you how to install Ladda[68] - a popular Javascript/Jquery plugin that we can use to make **button loading effects**.

[67] https://github.com/LearningLaravel/cookbook/releases/tag/v0.9
[68] https://github.com/hakimel/Ladda



Ladda

You can test all the effects at:

http://lab.hakim.se/ladda[69]

Be sure to disable the **Sweet Alert notification** if you don't want to see it:

[69] http://lab.hakim.se/ladda

```
//  Alert::error('There is an error', 'Error')->autoclose(2000);
```

Now, let's install the plugin!

First, download the latest version of Ladda[70].

Once downloaded, **unzip (decompress)** the file and go to **Ladda-1.0.0/dist**.

> **Note**: Your version of Ladda could be different.

Copy the **spin.min.js** file to your **public/js** directory.

Copy the **ladda.min.js** file to your **public/js** directory.

Copy the **ladda-themeless.min.css** file to your **public/css** directory.

Copy the **ladda.min.css** file to your **public/css** directory.

> **Note**: Create the **css** and **js** directory if you don't have.

Next, open our **master layout** (resources/views/layouts/app.blade.php). Find:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css\
/bootstrap-theme.min.css">
```

Add below:

```
<link rel="stylesheet" href="/css/ladda-themeless.min.css">
```

Find:

```
</body>
```

Add above:

```
<script src="/js/spin.min.js"></script>
<script src="/js/ladda.min.js"></script>
<script src="/js/custom_script.js"></script>
```

Create a new file called **custom_script.js** and place it at **public/js/custom_script.js**.

Copy the following code into the file:

[70] https://github.com/hakimel/Ladda/archive/master.zip

```
Ladda.bind( 'input[type=submit]', { timeout: 10000 } );

// Bind normal buttons
Ladda.bind( '.ladda-button', { timeout: 10000 } );

// Bind progress buttons and simulate loading progress
Ladda.bind( '.ladda-button', {
    callback: function( instance ) {
        var progress = 0;
        var interval = setInterval( function() {
            progress = Math.min( progress + Math.random() * 0.1, 1 );
            instance.setProgress( progress );

            if( progress === 1 ) {
                instance.stop();
                clearInterval( interval );
            }
        }, 200 );
    }
} );
```

This is how we attach a spinner to the desired button.

> **Note:** You may use jQuery instead. If you use jQuery, be sure to use the **ladda.jquery.min.js** file. Read the Ladda documentation[71] to know about that method. Feel free to use Gulp, Elixir or other tools to minify the custom_script.js file.

Well done! Ladda is now ready to use.

Use Ladda to create loading buttons

Since we already have Ladda installed, let's try to change the **Register button** of our **Register** page.

Open the **register view (resources/views/register.blade.php)**, and find the button:

```
<button type="submit" class="btn btn-primary">
```

We just need to change it to:

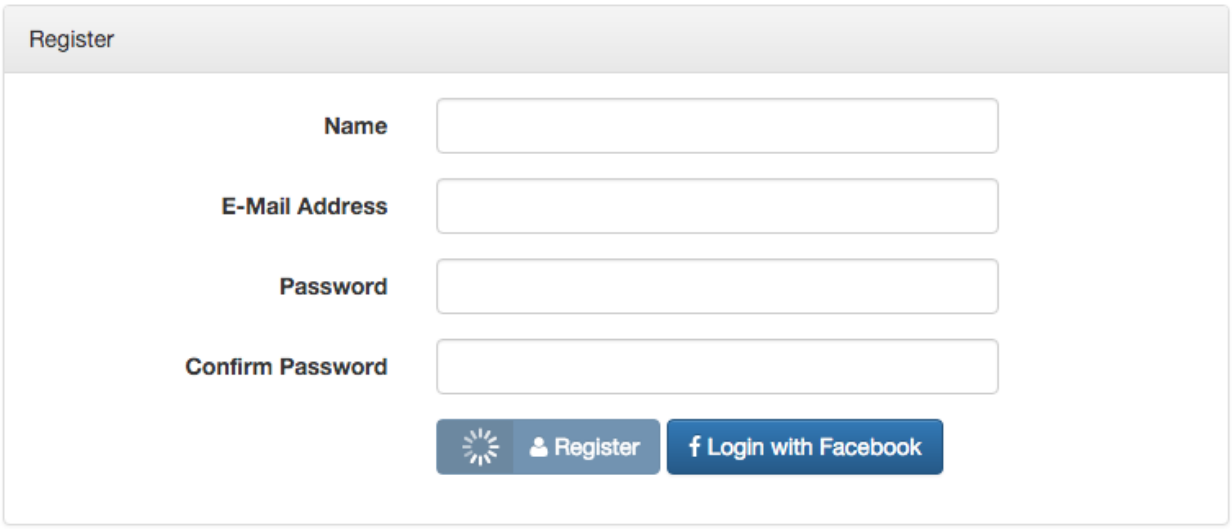
[71] https://github.com/hakimel/Ladda

```
<button type="submit" class="btn btn-primary ladda-button" data-style="expand-le\
ft">
```

As you see, we can choose **the effect** by setting the **data-style** attribute:

```
data-style="expand-left"
```

That's it! Go ahead and click the **Register** button, you should see the loading effect:



Register button

Of course that we can put the spinner inside our **Login With Facebook** button as well:

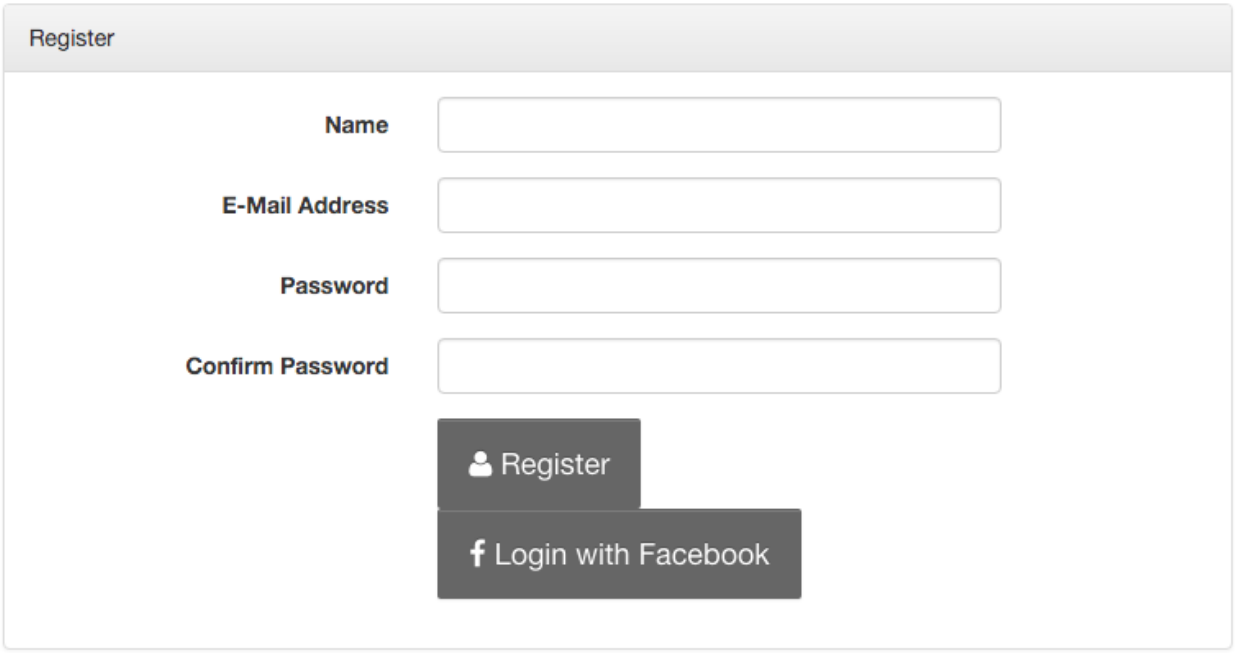
```
<a href="/login/facebook">
    <div class="btn btn-md btn-primary ladda-button" data-style="expand-left">
        <i class="fa fa-facebook"></i> Login with Facebook </div>
</a>
```

If you like the style of original Ladda buttons that we just see in the demo. Open our **master layout**, and change:

```
<link rel="stylesheet" href="/css/ladda-themeless.min.css">
```

To:

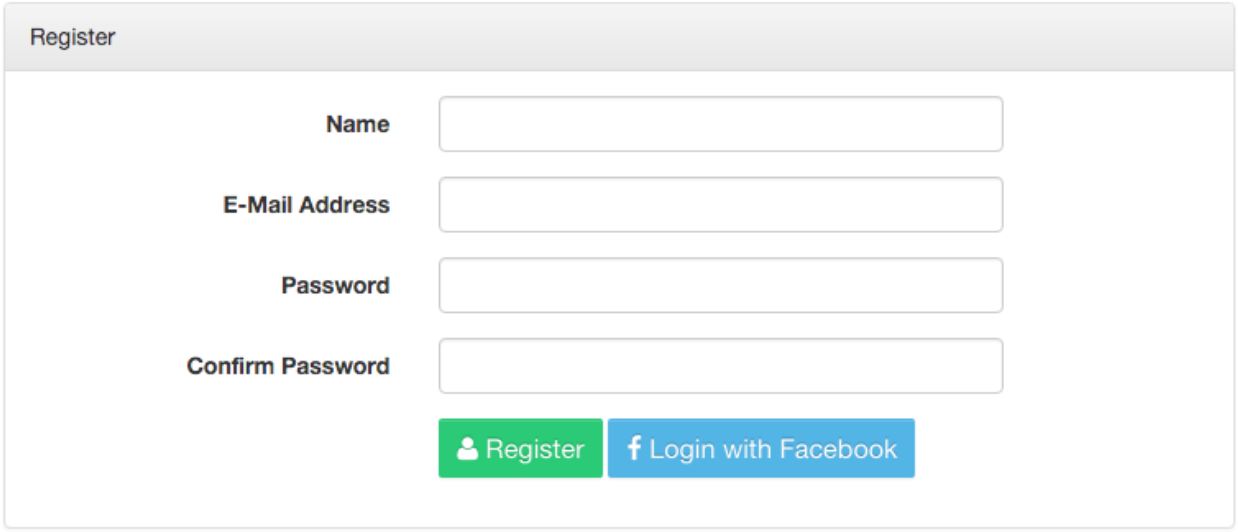
```
<link rel="stylesheet" href="/css/ladda.min.css">
```



Ladda buttons

Using **ladda.min.css**, we may change buttons' **size** and **color** by using the **data-size** and **data-color** attribute.

```
<button type="submit" class="btn btn-primary ladda-button" data-style="expand-le\
ft" data-size="s" data-color="green">
    <i class="fa fa-btn fa-user"></i> Register
</button>

<a href="/login/facebook">
    <div class="btn btn-md btn-primary ladda-button" data-style="expand-left" da\
ta-size="s" data-color="blue">
    <i class="fa fa-facebook"></i> Login with Facebook </div>
</a>
```



Ladda buttons

Here are all attributes that we can use:

- **data-style**: one of the button styles, full list in demo [required]
- **data-color**: green/red/blue/purple/mint
- **data-size**: xs/s/l/xl, defaults to medium
- **data-spinner-size**: 40, pixel dimensions of spinner, defaults to dynamic size based on the button height
- **data-spinner-color**: A hex code or any named CSS color.
- **data-spinner-lines**: 12, the number of lines the for the spinner, defaults to 12

Very cool, isn't it?

Recipe 202 Wrap-up

> Tag: Version 0.10 - Recipe 202[72]

This should give you a sample of how to use Ladda. Let's try to change other buttons by yourself to create some effects that fit your needs.

Using loading effects is very important when working with AJAX, because it's a great way to inform users that we're processing their requests.

[72] https://github.com/LearningLaravel/cookbook/releases/tag/v0.10

Recipe 203 - Create A Registration Page Using AJAX and jQuery

What will we learn?

This recipe shows you how to create a user registration system using AJAX and jQuery.

Building a registration form

When talking about AJAX forms, people usually think that they're very complicated. Don't worry, they are much simpler than they often seem. You can build AJAX forms even if you don't know much about jQuery and AJAX.

Basically, an AJAX registration form is very similar to the normal registration form that we already have, we only need to add some AJAX features to make better user experience.

For learning purposes, let's create different routes for our **AJAX registration page**. Open **routes.php**, add:

```
Route::get('users/register', 'Auth\AuthController@getRegister');
Route::post('users/register', 'Auth\AuthController@postRegister');
```

> **Note:** You may use different routes, different actions or different controllers if you want.

Next, open our **AuthController (app/Http/Controllers/Auth/AuthController)** and update the **getRegister** action as follows:

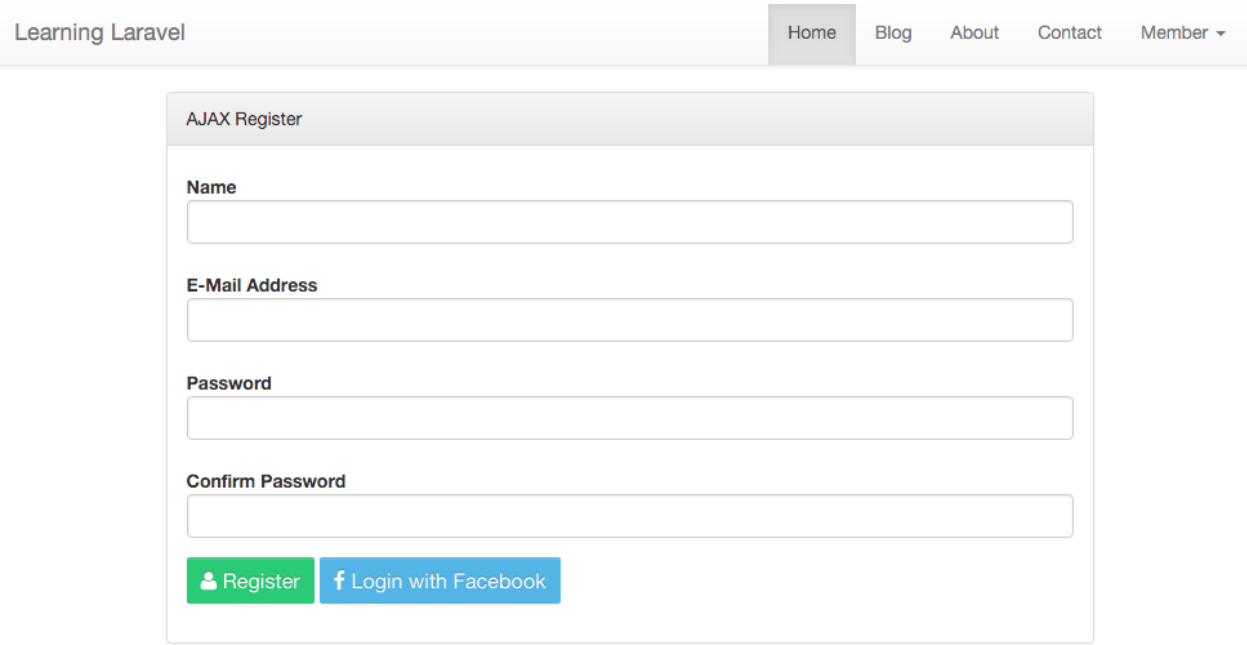
```
public function getRegister() {
    return view('auth/ajax_register');
}
```

By now you should be a pro at handling views, so let's create a new view called **ajax_register (resources/views/auth/ajax_register.blade.php)**:

```
@extends('layouts.app')

@section('content')
    <div class="container">
        <div class="row">
            <div class="col-md-8 col-md-offset-2">
                <div class="panel panel-default">
                    <div class="panel-heading">AJAX Register</div>
                    <div class="panel-body">
                        <form class="form-horizontal" id="registration" method="\
POST" action="{{ url('/users/register') }}">
                            {!! csrf_field() !!}

                            <div class="form-group">
                                <label class="col-md-4 control-label">Name</labe\
l>

                                <div class="col-md-6">
                                    <input type="text" class="form-control" name\
="name">
                                </div>
                            </div>

                            <div class="form-group">
                                <label class="col-md-4 control-label">E-Mail Add\
ress</label>

                                <div class="col-md-6">
                                    <input type="email" class="form-control" nam\
e="email">

                                </div>
                            </div>

                            <div class="form-group">
                                <label class="col-md-4 control-label">Password</\
label>

                                <div class="col-md-6">
                                    <input type="password" class="form-control" \
name="password" id="password">

```

```
                                </div>
                            </div>

                            <div class="form-group">
                                <label class="col-md-4 control-label">Confirm Pa\
ssword</label>

                                <div class="col-md-6">
                                    <input type="password" class="form-control" \
name="password_confirmation">
                                </div>
                            </div>

                            <div class="form-group">
                                <div class="col-md-6 col-md-offset-4">
                                    <button type="submit" class="btn btn-primary\
 ladda-button" data-style="expand-left"
                                     data-size="s" data-color="green">
                                        <i class="fa fa-btn fa-user"></i> Regist\
er
                                    </button>
                                    <a href="/login/facebook"> <div class="btn b\
tn-md btn-primary ladda-button"
                                    data-style="expand-left" data-size="s" data-\
color="blue">
                                        <i class="fa fa-facebook"></i> Login wit\
h Facebook </div></a>
                                </div>
                            </div>
                        </form>
                    </div>
                </div>
            </div>
        </div>
    </div>
@endsection
```

Of course, this view is used to display our AJAX registration form. As you can see, it's just a simple HTML form. Because this will be an AJAX form, we don't need to use **session** or the **errors variable** here.



Our new registration form

We can now access the form at http://cookbook.app/users/register[73]

Adding inline validation to our registration form

Using AJAX forms, we will need to find out a way to display the input validation as our users type. That means the page should not be refreshed, and users can see the generic feedback immediately. That feature which we want is called **Javascript form validation** or **inline validation**.

Luckily, there are many Javascript libraries that we can use to integrate **inline validation** into our form.

Here are popular (and free) Javascript inline validation libraries:

- Parsley[74]
- Validate.js[75]
- jQuery Validation Plugin[76]
- Verify.js[77]
- gvalidator[78]

[73] http://cookbook.app/users/register
[74] http://parsleyjs.org
[75] http://rickharrison.github.io/validate.js/
[76] http://jqueryvalidation.org
[77] http://verifyjs.com
[78] https://code.google.com/archive/p/gvalidator

Parsley is the most popular one, so we will use it to add inline validation to our registration form.

To install Parsley, you may choose one of the following methods:

Method 1: Using a CDN. Open our master layout (app.blade.php) and find:

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

Add below:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/parsley.js/2.3.5/parsley.min\
.js"></script>
```

Method 2: You can download Parsley (version 2.3.5) here[79].

Once downloaded, put the file at **public/js/parsley.min.js**.

Next, open our **master layout (app.blade.php)** and find:

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

Add below:

```
<script src="/js/parsley.min.js"></script>
```

Done! We now have Parsley installed!

To use Parsley, we just need to add **data-parsley-validate** to the form that we want to be validated.

Open the **ajax_register** view and find:

```
<form class="form-horizontal" id="registration" method="POST" action="{{ url('us\
ers/register') }}">
```

Change to:

```
<form class="form-horizontal" id="registration" method="POST" action="{{ url('us\
ers/register') }}" data-parsley-validate>
```

It's now time to add some validation rules to our form.

Find all input fields:

[79] http://parsleyjs.org/dist/parsley.min.js

```
<input type="text" class="form-control" name="name">

<input type="email" class="form-control" name="email">

<input type="password" class="form-control" name="password" id="password">

<input type="password" class="form-control" name="password_confirmation">
```

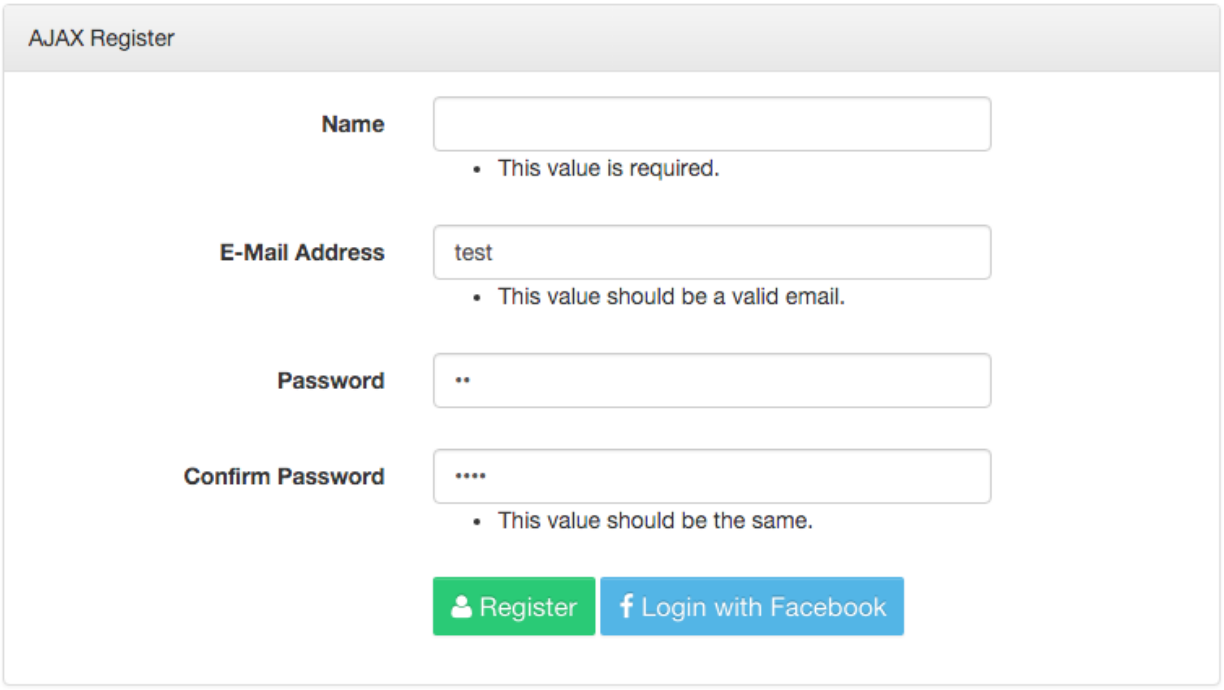
Change to:

```
<input type="text" class="form-control" name="name" required>

<input type="email" class="form-control" name="email" required>

<input type="password" class="form-control" name="password" id="password" requir\
ed>

<input type="password" class="form-control" name="password_confirmation" data-pa\
rsley-equalto="#password" required>
```

You might have noticed that we're adding the **required attribute** to our input fields. Users must enter all the required fields before submitting the form.

We also use **data-parsley-equalto="#password"** to make sure that the value of our **password_confirmation field** must be the same with the **password field**'s value.

Let's give our brand new inline validation system a try.



Our new registration form

If we enter wrong values and click the **Register** button. We should see some errors immediately. Parsley also detects the **email field** automatically and validate the field for us!

Amazing! Right?

The great thing is, we can **customize** all Parsley's **classes and elements** in the DOM when it validates.

Let's create a new **app.css** stylesheet and place it at **public/css/app.css.**. Add the following:

```
input.parsley-success,
select.parsley-success,
textarea.parsley-success {
    color: #468847;
    background-color: #DFF0D8;
    border: 1px solid #D6E9C6;
}

input.parsley-error,
select.parsley-error,
textarea.parsley-error {
    color: #B94A48;
    background-color: #F2DEDE;
    border: 1px solid #EED3D7;
```

```
}

.parsley-errors-list {
    margin: 2px 0 3px;
    padding: 0;
    list-style-type: none;
    font-size: 0.9em;
    line-height: 0.9em;
    opacity: 0;

    transition: all .3s ease-in;
    -o-transition: all .3s ease-in;
    -moz-transition: all .3s ease-in;
    -webkit-transition: all .3s ease-in;
}

.parsley-errors-list.filled {
    opacity: 1;
}
```

You can find these css rules at:

http://parsleyjs.org/src/parsley.css[80]

Finally, open our **master layout** and find:

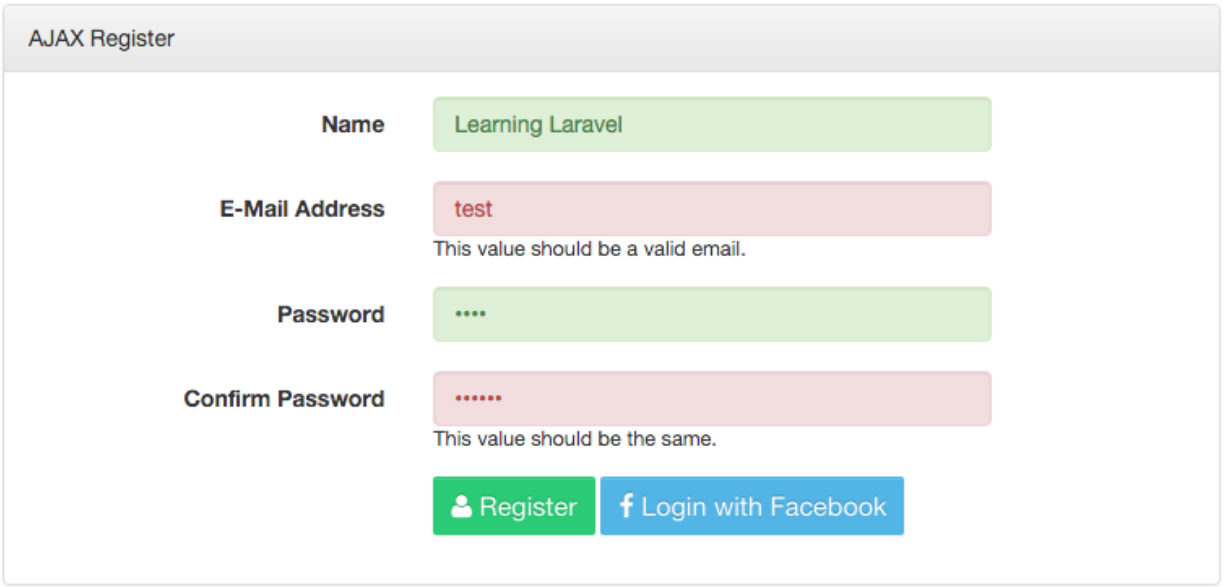
```
<link rel="stylesheet" href="/css/sweetalert.css">
```

Add below:

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

It's time to refresh our registration form to see the changes.

[80] http://parsleyjs.org/src/parsley.css



Our new registration form

We now have a beautiful registration form!

Optional: To access our AJAX registration page easier, open our **navbar view** (resources/views/shared/-navbar.blade.php) and find:

```
<li><a href="{{ url('/register') }}">Register</a></li>
```

Add below:

```
<li><a href="{{ url('/users/login') }}">AJAX Login</a></li>
<li><a href="{{ url('/users/register') }}">AJAX Register</a></li>
```

We can now access our AJAX registration and AJAX login page via the main menu.

Using AJAX and jQuery to submit the form

Now you know how to use client side JavaScript to validate the user input in web forms. There's just one more thing to do: **submitting the form using AJAX and jQuery**.

Definitely, this is the hardest part, especially if you don't know much about jQuery or Javascript. But stay with me, I'll try my best to make this simple enough. If you've made it this far in the book, you can do it!

First, let's learn how to use jQuery.

As before, we'll use the **custom_script.js** file. If you don't have one, create a new one and put it at **public/js/custom_script.js**. Be sure that you have the following code at the end of our **master layout (app.blade.php)**:

```
<script src="/js/custom_script.js"></script>
</body>
</html>
```

jQuery has a statement known as the **ready event**:

```
$( document ).ready(function() {

    // Our code will be here.

});
```

We have to put our code inside the **ready event**. When the document is ready, our code will run without waiting for other assets (images, files, etc.) to load.

Alternatively, you may use:

```
window.onload = function() {

    // Our code will be here.

};
```

or simply use:

```
 $(function() {
    // Our code will be here.
};
```

> **Note**: Please note that the last two methods are not recommended.

Choose one of the methods above, and put the code at the end of our **custom_script.js** file.

Next, we need to find our registration form by using the following:

```
$( document ).ready(function() {

    var form = $('#registration');

});
```

As you see, we just use **jQuery ID Selector** to select our registration form. Please note that our registration form must have the **id attribute**:

```
<form class="form-horizontal" id="registration" method="POST" action="{{ url('us\
ers/register') }}" data-parsley-validate>
```

Once the form is selected, we use **e.preventDefault()** method to **prevent** the **submit button** (the Register button) from **submitting the form** using the default action. Simply put, our browser should understand that we want to use the **Register button** to do other things. If we try to click the button now, it does nothing.

```
$( document ).ready(function() {

    var form = $('#registration');

    form.submit(function(e){
       e.preventDefault();
    });
});
```

This final step is interesting. We will use jQuery's **$.ajax()** function to send an **asynchronous HTTP request**:

```
$(document).ready(function () {

    var form = $('#registration');

    form.submit(function (e) {
        e.preventDefault();

        $.ajax({
                url: form.attr('action'),
                type: "POST",
                data: form.serialize(),
                dataType: "json"
            })
            .done(function (response) {
                // If the request succeeds, do something
            })
            .fail(function () {
                 // If the request fails, do something
            });
    });
});
```

Let's take a look deeper at this **$.ajax() function**:

The **url** parameter is the **URL** that we want to reach. We use **form.attr('action')** to get the value of our **form's action attribute**.

We're sending POST request, so the **type** is **POST**.

The **form.serialize()** is used to serialize the form data[81].

We know that we would get **a JSON object** in response, so the **dataType** should be **json**.

Once our requests are sent, we'll receive a response from the server. We'll use the **done()** and **fail()** method to handle it.

Here is the full code:

```
$(document).ready(function () {

    var form = $('#registration');

    form.submit(function (e) {
        e.preventDefault();

        $.ajax({
                url: form.attr('action'),
                type: "POST",
                data: form.serialize(),
                dataType: "json"
            })
            .done(function (response) {
                if (response.success) {
                    swal({
                        title: "Hi " + response.name,
                        text: response.success,
                        timer: 2000,
                        showConfirmButton: false,
                        type: "success"
                    });
                    window.location.replace(response.url);
                } else {
                    swal("Oops!", response.errors, 'error');
                }
            })
            .fail(function () {
```

[81] http://www.formget.com/javascript-serialize

```
29                 swal("Fail!", "Cannot register now!", 'error');
30             });
31     });
32 });
```

Let's see the code line by line.

If the **request succeeds**, we use **Sweet Alert** to display **a successful notification**:

```
swal({
    title: "Hi " + response.name,
    text: response.success,
    timer: 2000,
    showConfirmButton: false,
    type: "success"
});
```

We then redirect users to another place:

```
window.location.replace(response.url);
```

If we **can't register** a new member, we use **Sweet Alert** to display the errors:

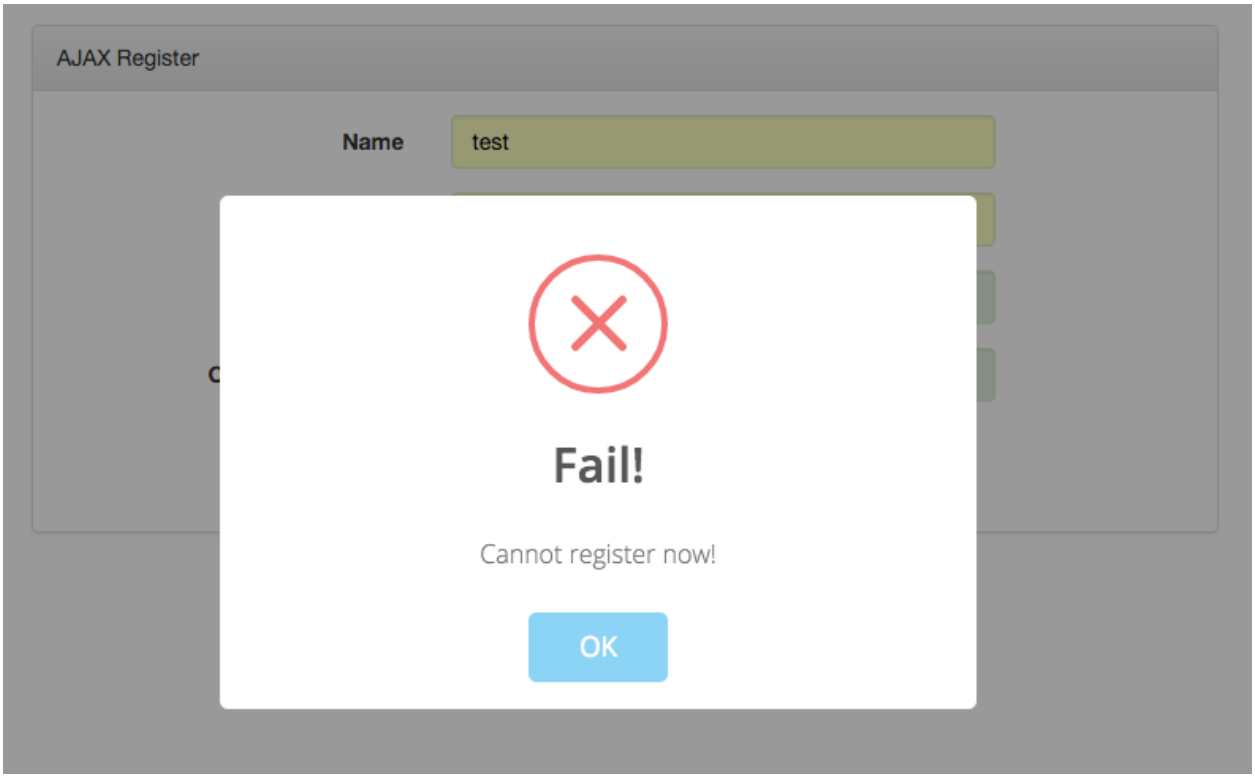
```
swal("Oops!", response.errors, 'error');
```

If the **request fails** (server problems) we also use **Sweet Alert** to trigger **error messages**.

```
swal("Fail!", "Cannot register now!", 'error');
```

Our "**frontend part**" is now complete!

Give it a try:



Our new registration form

Because we haven't built **the backend (server)** yet, we should see **an error message**.

Let's build the backend!

Building backend to handle AJAX requests

To start off, open our **AuthController** (app/Http/Controllers/Auth/AuthController) and update the **postRegister action** as follows:

```
1  public function postRegister(Request $request) {
2
3      $validator = Validator::make($request->all(), [
4          'email' => 'required|email|unique:users,email',
5          'name' => 'required|min:2',
6          'password' => 'required|alphaNum|min:6|same:password_confirmation',
7      ]);
8  }
```

You may notice that we've just created some **validation rules**. If you're not familiar with this, please take a look at the documentation[82]

[82] https://laravel.com/docs/master/validation#available-validation-rules

```
**Note:** You may create a RegisterFormRequest to validate the form or change th\
e rules if you want.
```

Next, if the **validation fails**, an error response will be generated to notify users. If the **form is valid**, a successful response will be generated, a new user will be created and we will send the user to a preferred location (dashboard, for example):

```
public function postRegister(Request $request) {

    $validator = Validator::make($request->all(), [
        'email' => 'required|email|unique:users,email',
        'name' => 'required|min:2',
        'password' => 'required|alphaNum|min:6|same:password_confirmation',
    ]);

    if ($validator->fails()) {
        $message = ['errors' => $validator->messages()->all()];
        $response = Response::json($message,202);
    } else {

        // Create a new user

        $user = new User([
            'name' => $request->name,
            'email' => $request->email,
            'facebook_id' => $request->email
        ]);
        $user->save();

        Auth::login($user);

        $message = ['success' => 'Thank you for joining us!', 'url' => '/', 'nam\
e' => $request->name];
        $response = Response::json($message,200);
    }
    return $response;
}
```

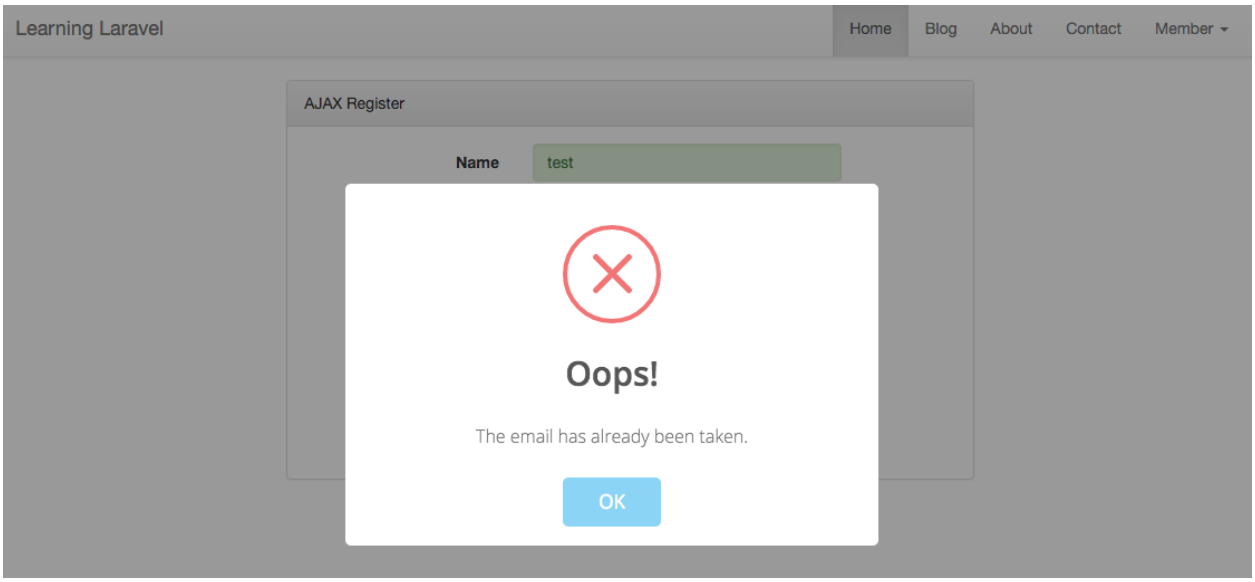
This might all be a lot to take in all at once, but the code is pretty easy.

Notice that we also use **Auth::login** to log the user in.

If we've done our job properly, we now have a working AJAX registration form!

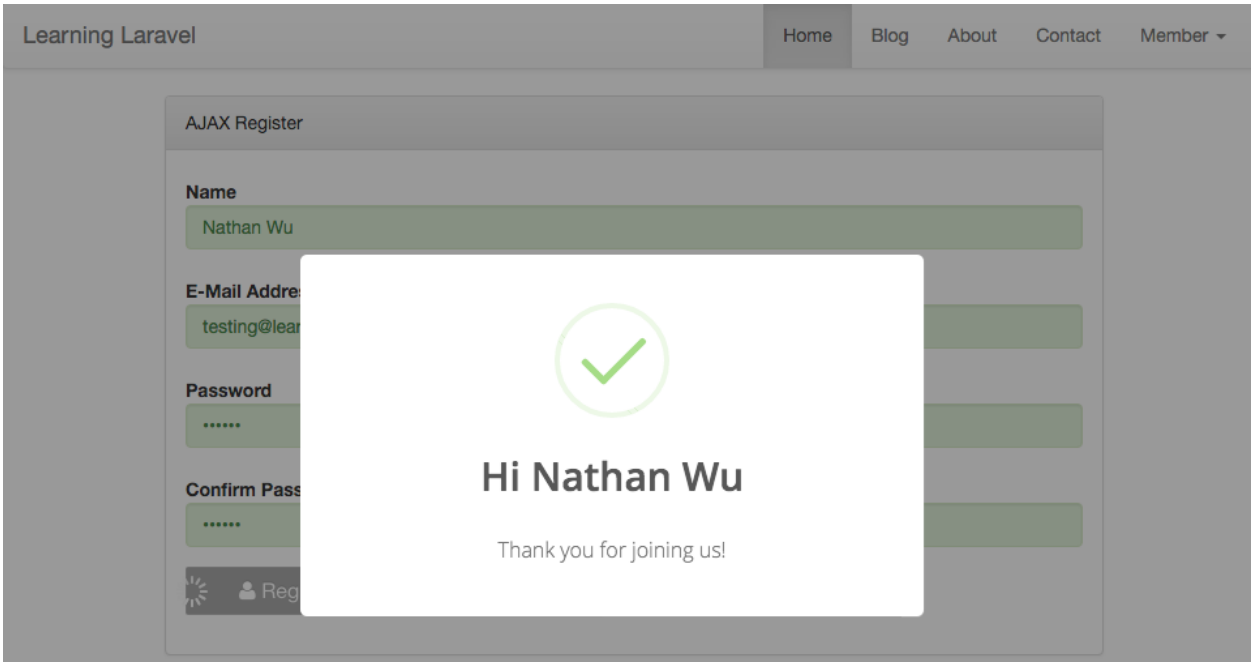
Let's check the form in our browser:

If we enter wrong credentials, an error notification should appear:



Our new registration form

If everything is fine, a new user should be created, we see a successful message and we're redirected to another location.



Our new registration form

Recipe 203 Wrap-up

Tag: Version 0.11 - Recipe 203[83]

Congratulations! By now you should have a good grasp of how to build an AJAX registration form.

This technique can be used to build many other AJAX forms. Go ahead and try to build another form to test your skill.

You may try to build the login form as well. I know that you can do it!

Recipe 204 - Create A Login Page Using AJAX And jQuery

What will we learn?

This recipe shows you how to create an AJAX login page using AJAX and jQuery.

Building a login form

So far we've built a registration form. It turns out that we can do the same thing to create an AJAX login form.

First of all, open **routes.php** and add these routes:

```
1 Route::get('users/login', 'Auth\AuthController@getLogin');
2 Route::post('users/login', 'Auth\AuthController@postLogin');
```

Next, open our **AuthController** (app/Http/Controllers/Auth/AuthController) and update the **getLogin action** as follows:

```
1 public function getLogin()
2 {
3     return view('auth/ajax_login');
4 }
```

Create a new view called **ajax_login** (resources/views/auth/ajax_login.blade.php):

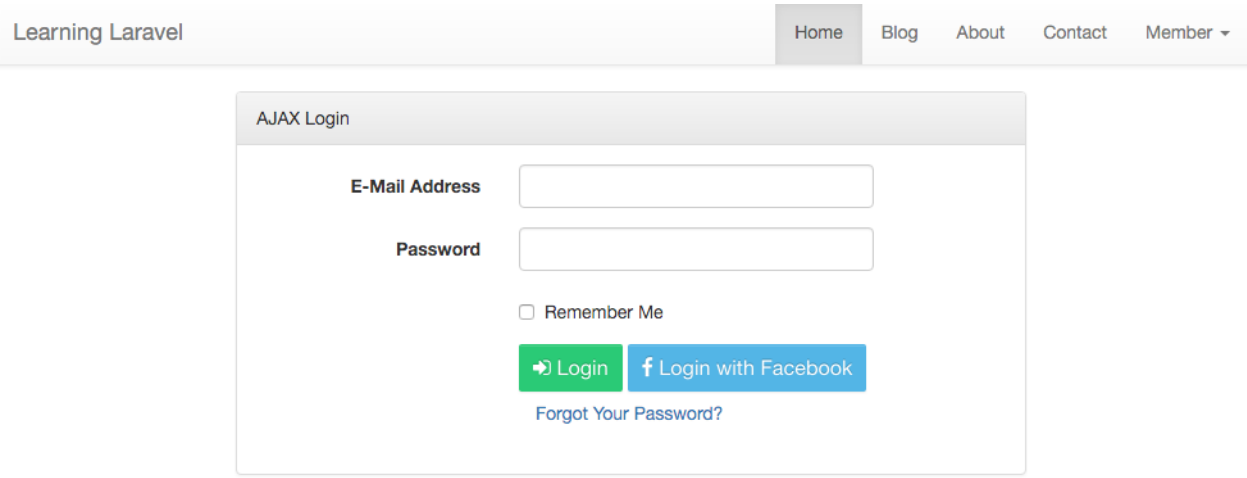
[83] https://github.com/LearningLaravel/cookbook/releases/tag/v0.11

```
@extends('layouts.app')

@section('content')
    <div class="container">
        <div class="row">
            <div class="col-md-8 col-md-offset-2">
                <div class="panel panel-default">

                    <div class="panel-heading">AJAX Login</div>
                    <div class="panel-body">

                        <form class="form-horizontal" id="login" method="POST" a\
ction="{{ url('/login') }}">
                            {!! csrf_field() !!}

                            <div class="form-group">
                                <label class="col-md-4 control-label">E-Mail Add\
ress</label>

                                <div class="col-md-6">
                                    <input type="email" class="form-control" nam\
e="email">
                                </div>
                            </div>

                            <div class="form-group">
                                <label class="col-md-4 control-label">Password</\
label>

                                <div class="col-md-6">
                                    <input type="password" class="form-control" \
name="password">
                                </div>
                            </div>

                            <div class="form-group">
                                <div class="col-md-6 col-md-offset-4">
                                    <div class="checkbox">
                                        <label>
                                            <input type="checkbox" name="remembe\
r"> Remember Me
                                        </label>
```

```
                                    </div>
                                </div>
                            </div>

                            <div class="form-group">
                                <div class="col-md-6 col-md-offset-4">
                                    <button type="submit" class="btn btn-primary\
 ladda-button" data-style="expand-left"
                                            data-size="s" data-color="green">
                                        <i class="fa fa-btn fa-sign-in"></i> Log\
in
                                    </button>
                                    <a href="/login/facebook">
                                        <div class="btn btn-md btn-primary ladda\
-button" data-style="expand-left"
                                             data-size="s" data-color="blue"><i \
class="fa fa-facebook"></i> Login with
                                            Facebook
                                        </div>
                                    </a>
                                    <a class="btn btn-link" href="{{ url('/passw\
ord/reset') }}">Forgot Your
                                        Password?</a>
                                </div>
                            </div>
                        </form>
                    </div>
                </div>
            </div>
        </div>
    </div>
@endsection
```

We can now access the form at http://cookbook.app/users/login[84].

[84] http://cookbook.app/users/login



Our new login form

Adding inline validation to our login form

Similarly, we will use **Parsley** to add inline validation to our login form.

> **Note**: Please read the previous recipe to learn how to install and use Parsley if you don't know what Parsley is.

As you may already know, we need to add **data-parsley-validate** to the form that we want to be validated.

Open the **ajax_login** view and find:

```
<form class="form-horizontal" id="login" method="POST" action="{{ url('/login') \
}}">
```

Change to:

```
<form class="form-horizontal" id="login" method="POST" action="{{ url('/login') \
}}" data-parsley-validate>
```

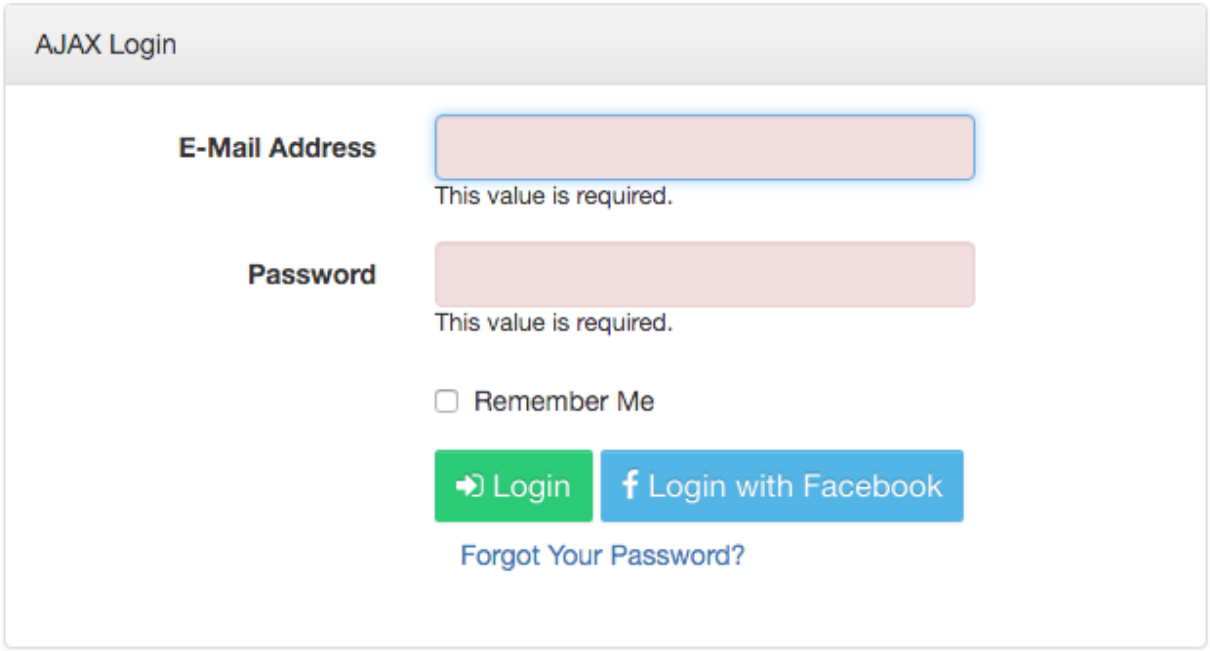
Find all input fields:

```
<input type="email" class="form-control" name="email">

<input type="password" class="form-control" name="password">
```

Update to:

```
<input type="email" class="form-control" name="email" required>

<input type="password" class="form-control" name="password" required>
```

Well done! We've integrated inline validation into our login form!



Inline validation

Using AJAX and jQuery to submit our login form

As before, we'll use the **custom_script.js** file. If you don't have one, create a new one and put it at **public/js/custom_script.js**. Be sure that you have the following code at the end of our **master layout** (app.blade.php):

```
<script src="/js/custom_script.js"></script>
</body>
</html>
```

We should put our code inside the **ready event**:

```
$( document ).ready(function() {

    // Registration form (previous recipe)

    // Our code will be here.

});
```

Next, we use **jQuery ID Selector** to select the login form.

```
var login_form = $('#login');
```

Once the form is selected, don't forget to use **e.preventDefault() method** to prevent the submit button (the Login button) from submitting the form using the default action.

```
var login_form = $('#login');

login_form.submit(function (e) {
    e.preventDefault();
});
```

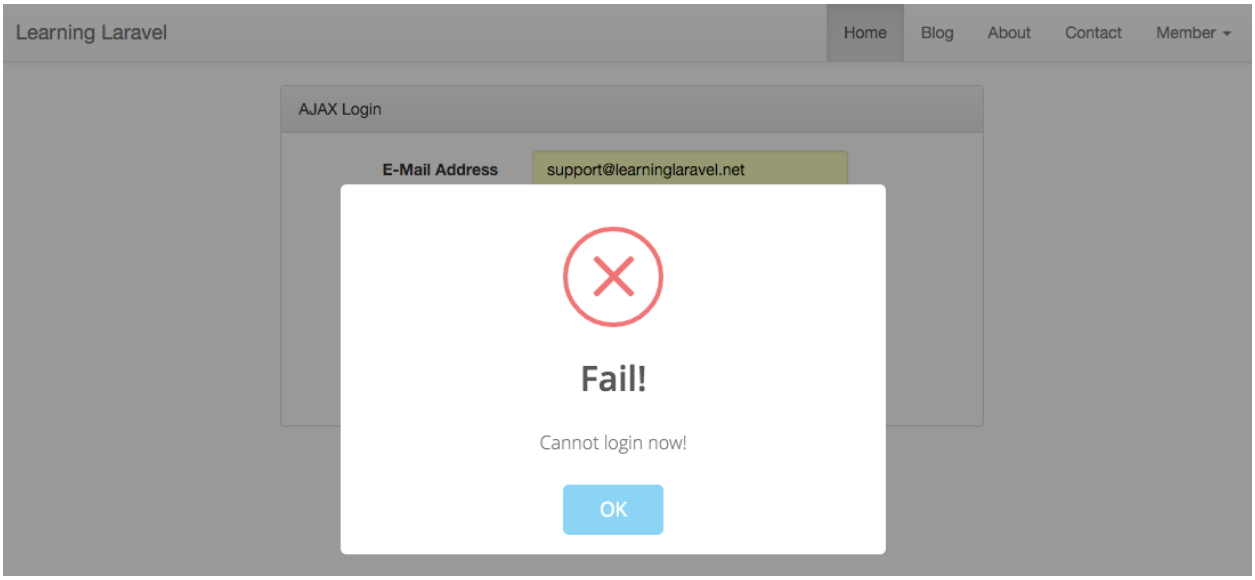
After that, we can use the **$.ajax()** function to submit the form:

```
var login_form = $('#login');

login_form.submit(function (e) {
    e.preventDefault();

    $.ajax({
            url: login_form.attr('action'),
            type: "POST",
            data: login_form.serialize(),
            dataType: "json"
        })
        .done(function (response) {
```

```
            if (response.success) {
                swal({
                    title: "Welcome back!",
                    text: response.success,
                    timer: 5000,
                    showConfirmButton: false,
                    type: "success"
                });

                window.location.replace(response.url);

            } else {
                swal("Oops!", response.errors, 'error');
            }
        })
        .fail(function () {
            swal("Fail!", "Cannot login now!", 'error');
        });
});
```

Just like we previously set up the registration form, we'll use **Sweet Alert** to display a **successful message** and we'll redirect the user to another location if the response from the server is **OK (200)**. If not, we also use Sweet Alert to display **error notifications**.

The form can be used to send our AJAX request now.



The form is working

Building the login backend

Here is the code for the **postLogin action**:

```
public function postLogin(Request $request)
{

    $validator = Validator::make($request->all(), [
        'email' => 'required|email',
        'password' => 'required',
    ]);

    if ($validator->fails()) {
        $message = ['errors' => $validator->messages()->all()];
        $response = Response::json($message, 202);
    } else {
        $remember = $request->remember? true : false;

        if (Auth::attempt(['email' => $request->email, 'password' => $request->p\
assword], $remember)) {

            $message = ['success' => 'Love to see you here!', 'url' => '/'];

            $response = Response::json($message, 200);
        } else {
            $message = ['errors' => 'Please check your email or password again.'\
];
            $response = Response::json($message, 202);
        }
    }

    return $response;
}
```

First, we use **Validator** to validate the form. If our validation rules pass, we use **Auth::attempt** to authenticate the user.

If the login credentials of the user are correct, we return a **successful response** with a **URL**. If not, we simply return an **error message**.

You may notice that we also use the **$remember** variable to store the value of the **Remember Me select box**.

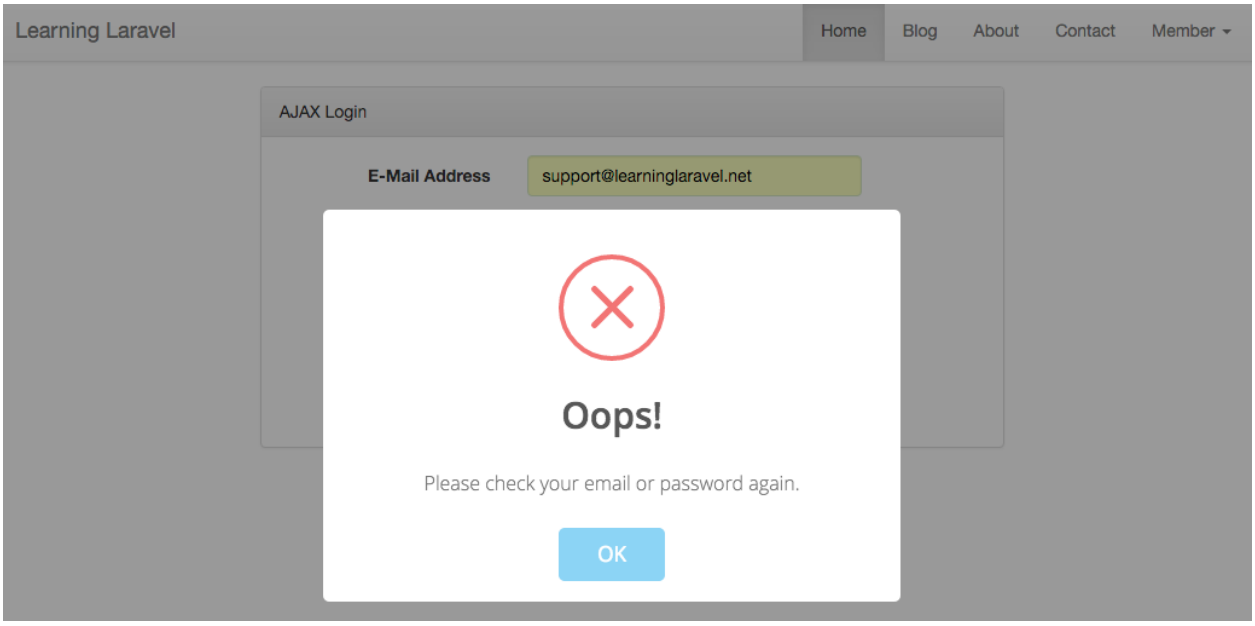
We use the variable for the **remember me** functionality in our application. When we pass the **$remember** variable (which is a boolean) as the **second argument** to the **attempt** method, if the value of the variable is **1 (yes)**, our app keeps the **user authenticated indefinitely**.

For more information about using the **Auth** facade, check this out:

https://laravel.com/docs/master/authentication#authenticating-users[85]

Let's give our new login form a try.

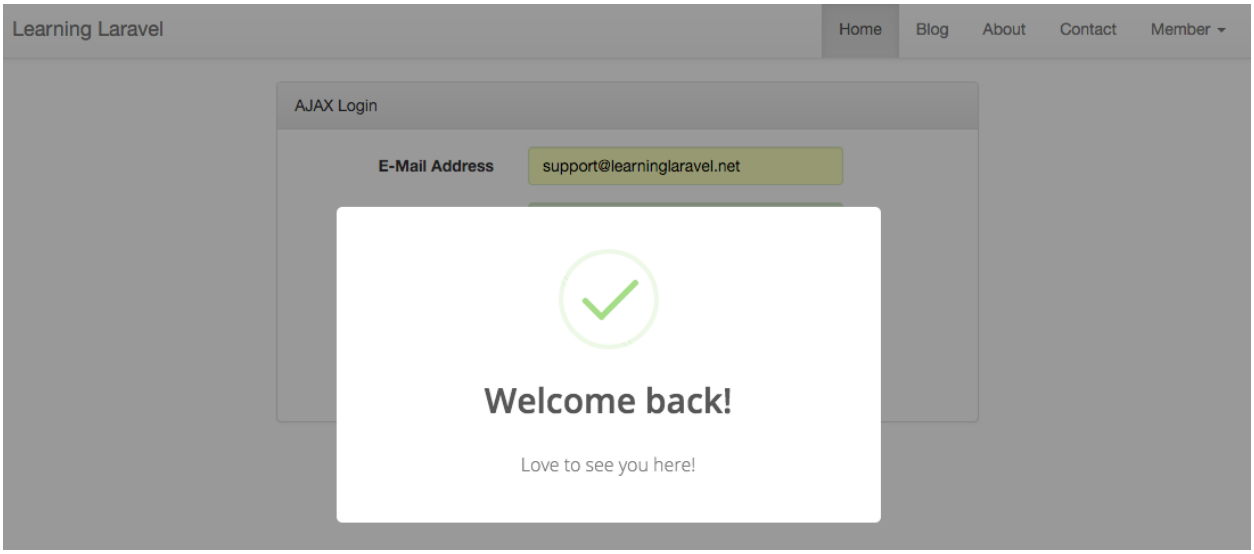
If we enter wrong information, an error notification should appear:



Error

If the credentials are correct, we will be able to log in!

[85] https://laravel.com/docs/master/authentication#authenticating-users



We can login now

Recipe 204 Wrap-up

Tag: Version 0.12 - Recipe 204[86]

By applying the techniques above, we can easily build an AJAX login page!

Although we have only dealt with users, these concepts can be applied to create many types of forms.

Recipe 205 - Upload Files Using AJAX And jQuery

What will we learn?

This recipe shows you how to upload images using AJAX and jQuery.

All about jQuery File Upload Plugin

We can find many open source file upload libraries, but it's very hard to get a library that works with any server-side platforms, supports multiple languages, easy to skin and have a good documentation.

Here is a list of the best file upload libraries:

- jQuery File Upload Plugin[87]

[86] https://github.com/LearningLaravel/cookbook/releases/tag/v0.12
[87] https://github.com/blueimp/jQuery-File-Upload

- DropzoneJS[88]
- Plupload[89]
- Uploadify[90]
- jQuery DROPAREA[91]
- jqUploader[92]

In this recipe, we'll learn about **jQuery File Upload Plugin**, which is the most popular jQuery file upload library.

Here are some of the most prominent jQuery File Upload's features:

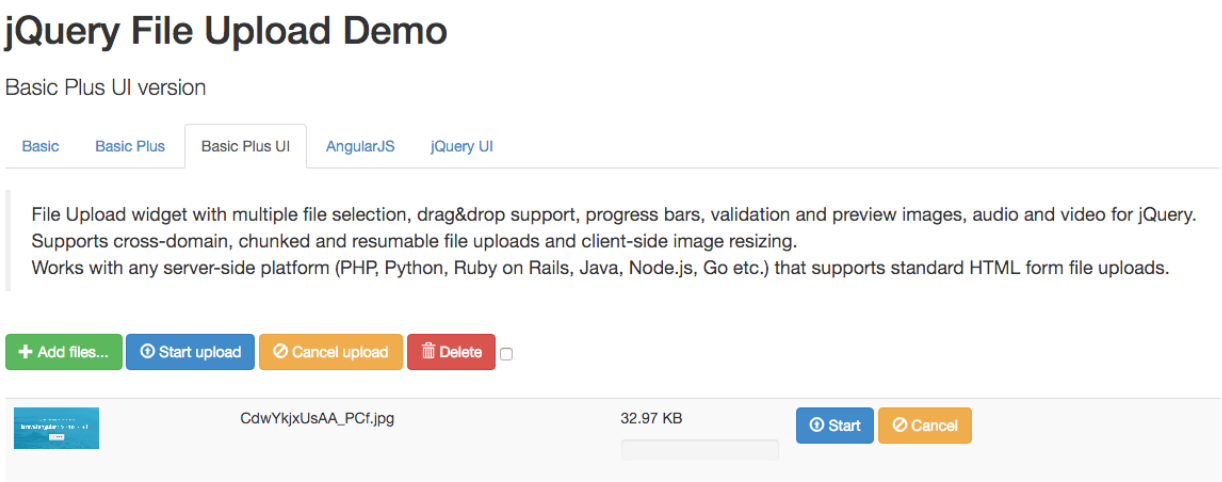
- Multiple file upload.
- Drag & Drop support.
- Upload progress bar.
- Resumable uploads.
- Chunked uploads.
- Preview images, audio and video.
- Graceful fallback for legacy browsers.
- Multipart and file contents stream uploads.
- Compatible with any server-side application platform.

You may view all jQuery File Upload's features and its documentation at:

https://github.com/blueimp/jQuery-File-Upload[93]

Be sure to check out the demo[94] to see how it works.

[88] http://www.dropzonejs.com
[89] http://www.plupload.com/example_queuewidget.php
[90] http://www.uploadify.com
[91] http://gokercebeci.com/dev/droparea
[92] http://pixeline.be/experiments/jqUploader
[93] https://github.com/blueimp/jQuery-File-Upload
[94] https://blueimp.github.io/jQuery-File-Upload



jQuery File Upload

Installing jQuery File Upload

To install jQuery File Upload, we have to download its latest version[95] first.

Unzip (decompress) the downloaded file, and go to the **js** directory.

jQuery File Upload comes with many files, but we only need these files:

- jquery.fileupload-image.js
- jquery.fileupload-process.js
- jquery.fileupload-ui.js
- jquery.fileupload.js
- jquery.iframe-transport.js
- vendor/jquery.ui.widget.js

> **Note**: You may use all the files if you want.

Place them all at our **public/js** directory.

Next, open the **master layout** (app.blade.php) and find:

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

Add below:

[95] https://github.com/blueimp/jQuery-File-Upload/archive/master.zip

```
<!-- The Load Image plugin is included for the preview images and image resizing\
 functionality -->
<script src="//blueimp.github.io/JavaScript-Load-Image/js/load-image.all.min.js"\
></script>
<!-- The Canvas to Blob plugin is included for image resizing functionality -->
<script src="//blueimp.github.io/JavaScript-Canvas-to-Blob/js/canvas-to-blob.min\
.js"></script>
<!-- jQuery File Upload Plugin -->
<script src="/js/jquery.ui.widget.js"></script>
<script src="/js/jquery.iframe-transport.js"></script>
<script src="/js/jquery.fileupload.js"></script>
<script src="/js/jquery.fileupload-process.js"></script>
<script src="/js/jquery.fileupload-image.js"></script>
```

Done! jQuey File Upload plugin is now ready to use!

But note that just because we can use these:

```
<script src="//blueimp.github.io/JavaScript-Load-Image/js/load-image.all.min.js"\
></script>
<script src="//blueimp.github.io/JavaScript-Canvas-to-Blob/js/canvas-to-blob.min\
.js"></script>
```

doesn't mean you should just use them in a production environment. We should download those files, and put them in the **public/js** directory.

Our master layout should look like this:

```
<script src="/js/load-image.all.min.js"></script>
<script src="/js/canvas-to-blob.min.js"></script>
<!-- jQuery File Upload Plugin -->
<script src="/js/jquery.ui.widget.js"></script>
<script src="/js/jquery.iframe-transport.js"></script>
<script src="/js/jquery.fileupload.js"></script>
<script src="/js/jquery.fileupload-process.js"></script>
<script src="/js/jquery.fileupload-image.js"></script>
```

Creating an upload form

We'll begin by creating a new form to upload images.

Because our **about page** is empty, we'll put the form there. Open **resources/views/about.blade.php** and update the code as follows:

```
@extends('layouts.app')
@section('title', 'About')

@section('content')

    <div class="container">
        <div class="content">
            <div class="title">About Page</div>
            <div>
                <div id="files" class="files">
                        <div id="testimage"><img src="/images/testimage.png" alt\
="test image"></div>
                </div>
                 <span class="btn btn-info btn-file">
                        Upload an image
                           <input id="fileupload" class="upload" type="file" na\
me="files[]">
                 </span>
                <div id="progress" class="progress" style="display:none;">
                    <div class="progress-bar progress-bar-success"></div>
                </div>
            </div>
        </div>
    </div>

@endsection
```

Let's see the code line by line.

First, a default image is placed at the top of the page:

```
<div id="files" class="files">
        <div id="testimage"><img src="/images/testimage.png" alt="test image"></\
div>
</div>
```

Currently, we don't have the **testimage.png** yet, so the image won't display.

(Optional) You may download the image below (or use any image that you like) and save it at **public/images/testimage.png**.

Learning Laravel 5 cover image[96]

Here is our upload button:

[96] http://learninglaravel.net/img/LearningLaravel5_cover.png

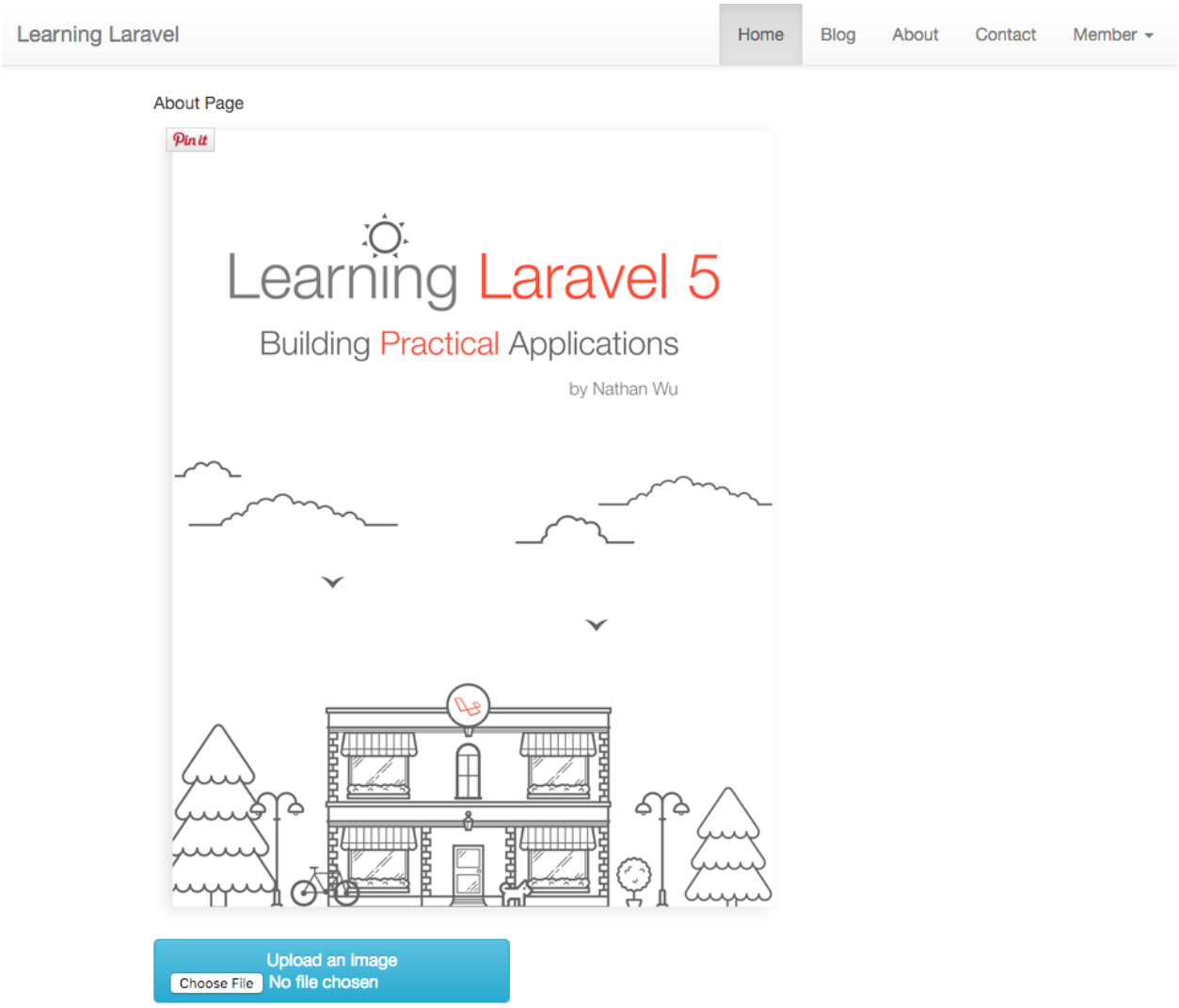
```
 <span class="btn btn-info btn-file">
        Upload an image
           <input id="fileupload" class="upload" type="file" name="files[]">
 </span>
```

As you see, we don't have to create a form here. A simple button is more than enough.

Lastly, we put the **progress bar** at the bottom:

```
<div id="progress" class="progress" style="display:none;">
    <div class="progress-bar progress-bar-success"></div>
</div>
```

Give it a try. You should see something like this:



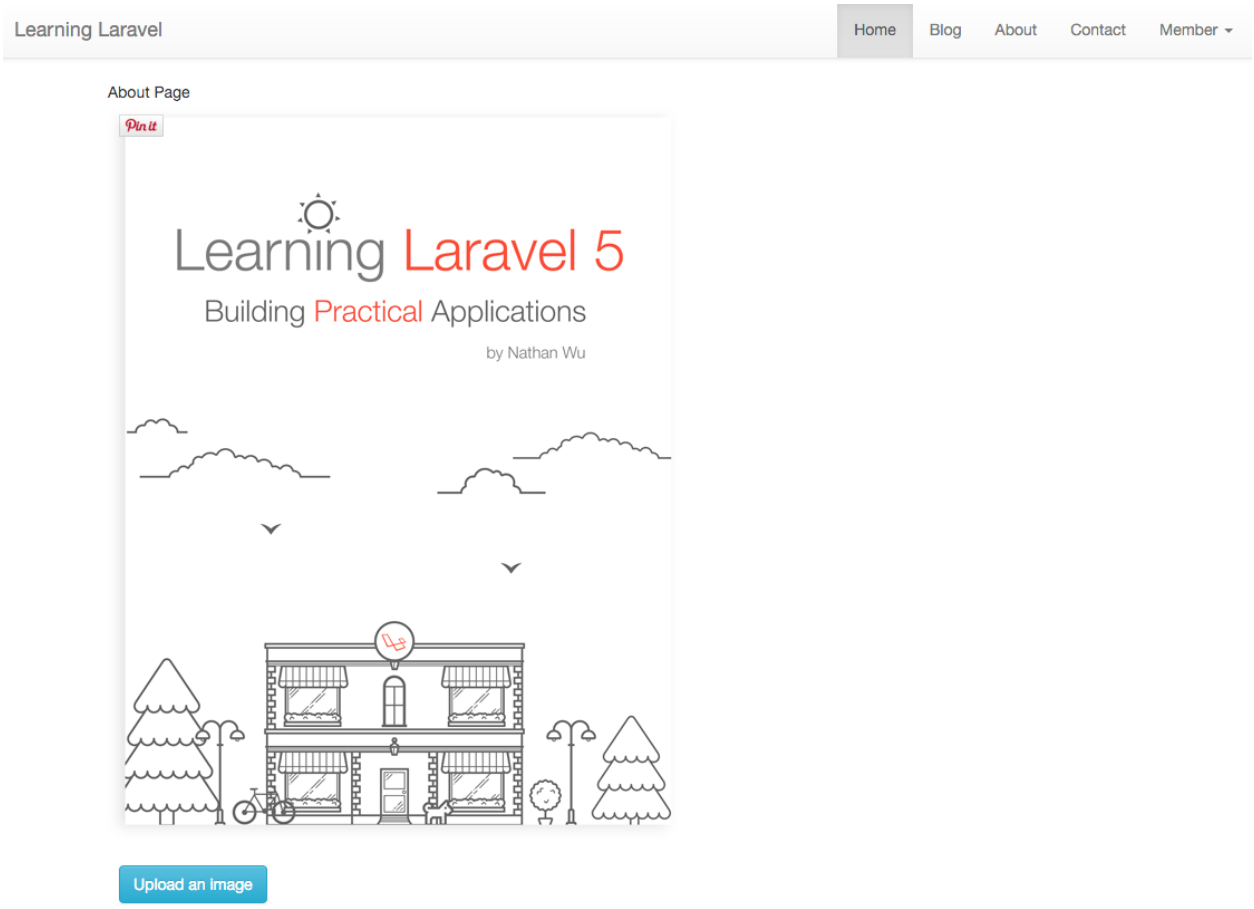
Upload form

You can also customize the style of the upload button and everything else to your liking. For instance, we may style the button in pure css[97]. Open our **public/css/app.css** file, add the following:

[97] http://geniuscarrier.com/how-to-style-a-html-file-upload-button-in-pure-css

```
.btn-file {
    position: relative;
    overflow: hidden;
    margin: 10px;
}
.btn-file input.upload {
    position: absolute;
    top: 0;
    right: 0;
    margin: 0;
    padding: 0;
    font-size: 20px;
    cursor: pointer;
    opacity: 0;
    filter: alpha(opacity=0);
}
```

Here is our new upload button:



The new upload button

Uploading images using jQuery File Upload

Now we'll use jQuery File Upload to upload an image.

First of all, open the **routes.php** file, add this route to the **web middleware group**:

```
1  Route::post('imageupload', 'ImagesController@storeImage');
```

This route is where the upload request is sent to.

Next, open the **master layout** (app.blade.php), find:

```
1  <script src="/js/jquery.fileupload-image.js"></script>
```

Add below:

```
 <meta name="_token" content="{{ csrf_token() }}" />

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

Because the **web middleware group** has the **CSRF middleware**, we have to **generate a CSRF token** and send it with our form. If we don't have a CSRF token, we will get a **500 internal server error**.

> **Note:** if you don't want to use the **CSRF feature** and generate the token, you may put the **imageupload** route outside of the web middleware group.

Create a new file called **upload.js** and place it at **public/js/upload.js**:

```
$.ajaxSetup({
    headers: {'X-CSRF-Token': $('meta[name=_token]').attr('content')}
});

$(function () {
    'use strict';

    var url = '/imageupload';

    $('#fileupload').fileupload({
        url: url,
        dataType: 'json',
        autoUpload: true,
        acceptFileTypes: /(\.|\/)(gif|jpe?g|png)$/i,
        singleFileUploads: true,
        maxFileSize: 999000,
        previewMaxWidth: 300,
        previewMaxHeight: 300,
        previewCrop: false
    }).on('fileuploadadd', function (e, data) {

        $('#progress').fadeIn();
        data.context = $('<div class="fileinfo"><div/>').appendTo('#files');
        $.each(data.files, function (index, file) {
            var node = $('<p/>')
                .append($('<span/>').text(file.name));
            node.appendTo(data.context);
        });
    }).on('fileuploadprocessalways', function (e, data) {

```

```
        var index = data.index,
            file = data.files[index],
            node = $(data.context.children()[index]);
        if (file.preview) {
            node
                .prepend('<br>')
                .prepend(file.preview);
        }
    }).on('fileuploadprogressall', function (e, data) {

        var progress = parseInt(data.loaded / data.total * 100, 10);
        $('#progress .progress-bar').css(
            'width',
            progress + '%'
        );
    }).on('fileuploaddone', function (e, data) {

        $('#files').empty();
        $.each(data.result.files, function (index, file) {
            if (file.url) {
                var currentTime = (new Date()).getTime();
                $('#files').append("<div id='testimage'><img src='" + file.url +\
 "?" + currentTime + "' /></div>");

                // reset the progress bar
                $('#progress').fadeOut();
                setTimeout(function () {
                    $('#progress .progress-bar').css('width', 0);
                }, 500);

            } else if (file.error) {
                var error = $('<span class="text-danger"/>').text(file.error);
                $(data.context.children()[index])
                    .append('<br>')
                    .append(error);
            }
        });
    }).on('fileuploadfail', function (e, data) {

        $.each(data.files, function (index) {
            var error = $('<span class="text-danger"/>').text('File upload faile\
d.');
```

```
            $(data.context.children()[index])
                .append('<br>')
                .append(error);
        });
    });
});
```

This may seem overwhelming at first, but the code is easy. Let's take a look deeper!

First, we use **$.ajaxSetop()** to add a default header to every request:

```
$.ajaxSetup({
    headers: {'X-CSRF-Token': $('meta[name=_token]').attr('content')}
});
```

This header **contains the CSRF token** that we have generated.

Next, we use the **fileupload** method to **initialize** the File Upload widget:

```
var url = '/imageupload';

$('#fileupload').fileupload({
    url: url,
    dataType: 'json',
    autoUpload: true,
    acceptFileTypes: /(\.|\/)(gif|jpe?g|png)$/i,
    singleFileUploads: true,
    maxFileSize: 999000,
    previewMaxWidth: 300,
    previewMaxHeight: 300,
    previewCrop: false
})
```

There are many options[98] that we can use to configure the plugin. For instance, the **url** option can be used to specify where the request is sent (**/imageupload**).

You may remove some options or add more option if you want.

jQuery File Upload also provides us some **callbacks**[99] that we can use to execute code during some events.

[98] https://github.com/blueimp/jQuery-File-Upload/wiki/Options

[99] https://github.com/blueimp/jquery-file-upload/wiki/options#callback-options

```
    .on('fileuploadadd', function (e, data) {

        $('#progress').fadeIn();
        data.context = $('<div class="fileinfo"><div/>').appendTo('#files');
        $.each(data.files, function (index, file) {
            var node = $('<p/>')
                .append($('<span/>').text(file.name));
            node.appendTo(data.context);
        });
    })
```

As you see, we use the **fileuploadadd** callback here. This callback is invoked as soon as files are added to the **fileupload** widget.

When files are added, we will display **a progress bar**, and add the **files' name** to the **#files** section.

```
.on('fileuploadprocessalways', function (e, data) {

        var index = data.index,
            file = data.files[index],
            node = $(data.context.children()[index]);
        if (file.preview) {
            node
                .prepend('<br>')
                .prepend(file.preview);
        }
    })
```

fileuploadprocessalways is the callback for the end of an individual file processing queue.

When the file is processed, we will display preview images on the page.

```
.on('fileuploadprogressall', function (e, data) {

        var progress = parseInt(data.loaded / data.total * 100, 10);
        $('#progress .progress-bar').css(
            'width',
            progress + '%'
        );
    })
```

When the file is being processed, we calculate the progress bar percentage here and change the width of the bar using CSS.

```
.on('fileuploaddone', function (e, data) {

        $('#files').empty();
        $.each(data.result.files, function (index, file) {
            if (file.url) {
                var currentTime = (new Date()).getTime();
                $('#files').append("<div id='testimage'><img src='" + file.url +\
 "?" + currentTime + "' /></div>");

                // reset the progress bar
                $('#progress').fadeOut();
                setTimeout(function () {
                    $('#progress .progress-bar').css('width', 0);
                }, 500);

            } else if (file.error) {
                var error = $('<span class="text-danger"/>').text(file.error);
                $(data.context.children()[index])
                    .append('<br>')
                    .append(error);
            }
        });
    })
```

When the file is **uploaded successfully**, we remove all the images in the **#files** section using:

```
$('#files').empty();
```

After that, we will insert a new image into the **#files** section.

Actually, we can just simply use the following code to display the image:

```
$('#files').append("<div id='testimage'><img src='" + file.url + "' /></div>");
```

However, some browsers have **browser cache**. When the new image has the same name with the old one, the image is not updated. We could not see the new image.

This is how we fix the issue:

```
var currentTime = (new Date()).getTime();
$('#files').append("<div id='testimage'><img src='" + file.url + "?" + currentTi\
me + "' /></div>");
```

As you see, we may add **a current timestamp** at the end of the image URL. For example, the URL of the image is changed to **images/testimage.png?1458728374846**.

```
   // reset the progress bar
    $('#progress').fadeOut();
    setTimeout(function () {
        $('#progress .progress-bar').css('width', 0);
    }, 500);
```

Next, we **reset the progress bar** and **hide it**.

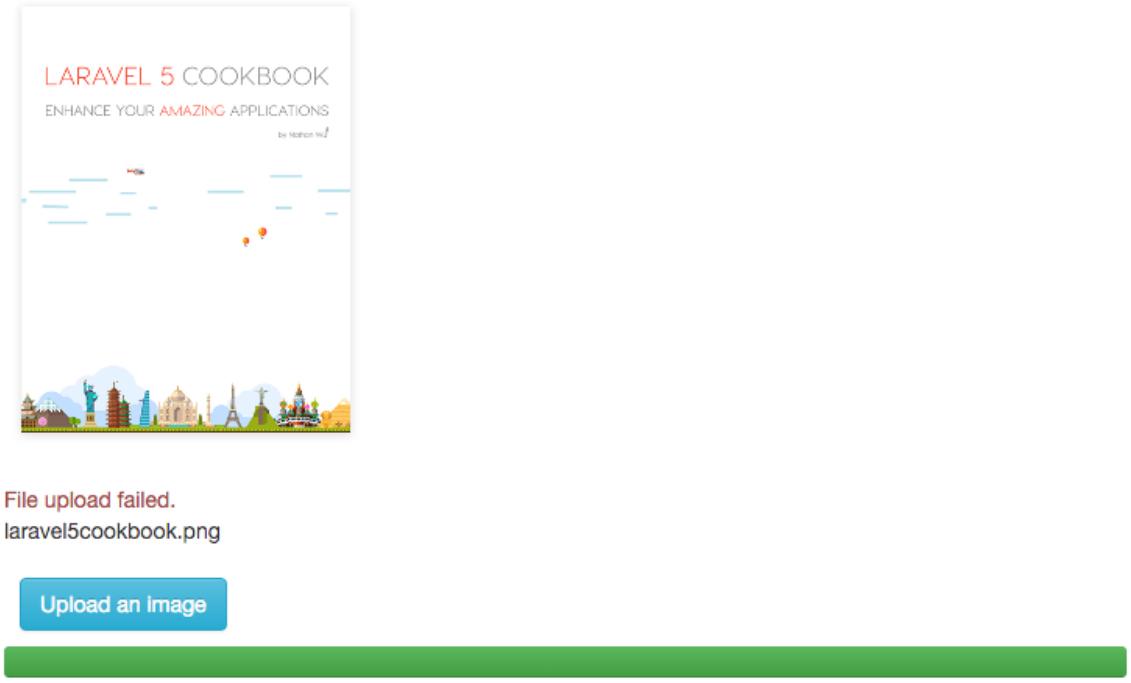
```
} else if (file.error) {
                    var error = $('<span class="text-danger"/>').text(file.error\
);
                    $(data.context.children()[index])
                        .append('<br>')
                        .append(error);
                }
            });
```

If the file has some errors, we insert the error messages to the page.

```
}).on('fileuploadfail', function (e, data) {

        $.each(data.files, function (index) {
            var error = $('<span class="text-danger"/>').text('File upload faile\
d.');
            $(data.context.children()[index])
                .append('<br>')
                .append(error);
        });
    });
```

Finally, if the request fails, all that left to do is to display an error message.

Here's what our form should now look like when we try to upload a new image:



The request fails

If you still feel confused, try to remove some callbacks and modify some options to get a better understanding of what's really going on behind the scenes.

> **Note**: Please note that we may also use Sweet Alert to display the messages as well. Some callbacks (such as **fileuploadprocessalways**) can be removed.

Building the backend

The process we'll follow will be pretty similar to how we've built the image upload backend in **Recipe 6**.

> **Note**: We will use **Intervention Image** in this section. If you don't have the package installed, please read the Recipe 6.

If you don't have the **ImagesController** yet, let's generate a new one:

```
1  php artisan make:controller ImagesController
```

Once we have the **ImagesController** file, open it and add this **storeImage** action:

```
public function storeImage()
{

    $files = Input::file('files');

    $json = array(
        'files' => array()
    );

    foreach ($files as $file) {

        $destination = 'images';
        $size = $file->getSize();
        $filename = 'testimage';
        $extension = 'png';
        $fullName = $filename . '.' . $extension;
        $pathToFile = $destination . '/' . $fullName;
        $upload_success = Image::make($file)->encode('png')->save($destination .\
 '/' . $fullName);

        if ($upload_success) {
            $json['files'][] = array(
                'name' => $filename,
                'size' => $size,
                'url' => $pathToFile,
                'message' => 'Uploaded successfully'
            );
            return Response::json($json);
        } else {
            $json['files'][] = array(
                'message' => 'error uploading images',
            );
            return Response::json($json, 202);
        }
    }
}
```

As you notice, we set the image's name as **testimage.png**. After that, We also use **Intervention Image** to convert the image to PNG and move it to our **public/images** directory:

```
$destination = 'images';
$size = $file->getSize();
$filename = 'testimage';
$extension = 'png';
$fullName = $filename . '.' . $extension;
$pathToFile = $destination . '/' . $fullName;
$upload_success = Image::make($file)->encode('png')->save($destination . '/' . $\
fullName);
```

If the image is **uploaded successfully**, we return a **JSON object** containing a **files array**:

```
if ($upload_success) {
    $json['files'][] = array(
        'name' => $filename,
        'size' => $size,
        'url' => $pathToFile,
        'message' => 'Uploaded successfully'
    );
    return Response::json($json);
}
```

> **Note**: even if only one file is uploaded, the response should always be a **JSON object** containing a **files** array.

If we **can't upload** the image, **an error message** is returned:

```
else {
                $json['files'][] = array(
                    'message' => 'error uploading images',
                );
                return Response::json($json, 202);
            }
```

Our backend should now be working perfectly!

Let's try to upload an image:

Upload an image

We should see a **progress bar** and **a preview image** while uploading. Our new image appears without reloading the page.

If you've done it right, your image should now be uploaded successfully.

Here is a little tip. If you don't want to set the image's name or its extension, you may use the following:

```
$destination = 'images';
$time = time();
$formatTime = date("Y-m-d_h-m", $time);
$filename = $formatTime . '_' . str_random(8);
$extension = $file->getClientOriginalExtension();
$size = $file->getSize();
$fullName = $filename . '.' . $extension;
$pathToFile = $destination . '/' . $fullName;
$upload_success = Image::make($file)->save($destination . '/' . $fullName);
```

This time we generate the image's name automatically and preserve the original image's extension.

You may also insert the image link into your database to keep track of it. When you have the links, you can display images anywhere on your site.

Recipe 205 Wrap-up

> Tag: Version 0.13 - Recipe 205[100]

So we've seen how to go from integrating jQuery File Upload to uploading images asynchronously.

As you see, the plugin is is very customizable. Using the techniques, you can build some useful features such as: uploading the site's cover, changing user's profile picture, etc.

This just covers the basics of what you can do with jQuery File Upload, be sure to explore its features more!

Recipe 206 - Cropping Images Using jQuery

What will we learn

This recipe shows you how to upload and crop images using jQuery.

All about Cropper

For **cropping images** using jQuery, there are many popular plugins:

- Cropper[101]
- Croppic[102]

[100] https://github.com/LearningLaravel/cookbook/releases/tag/v0.13
[101] https://github.com/fengyuanchen/cropper
[102] http://www.croppic.net

- Cropit[103]
- JCrop[104]
- Cropimg[105]
- jQuery Guillotine Plugin[106]

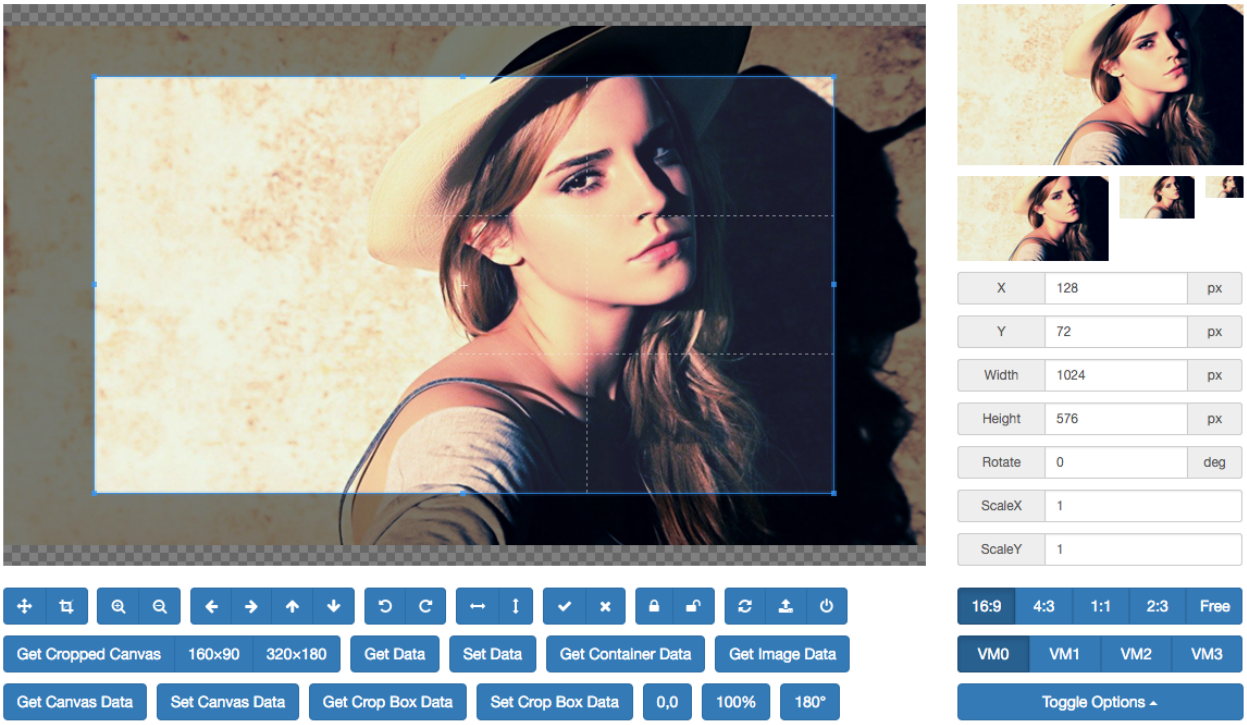
Currently, it's hard to find a better plugin than **Cropper**.

Cropper has many features and it is still strongly maintained. At the time of writing this section, the last commit to Github was less than 2 weeks ago.

Here are some of Cropper's **prominent features**:

- It has 39 options, 27 methods and 7 events
- Supports touch (mobile)
- Supports zooming
- Supports rotating
- Supports scaling (flipping)
- Supports multiple croppers
- Supports to crop image in the browser-side by canvas
- Cross-browser support

[103] http://scottcheng.github.io/cropit
[104] http://deepliquid.com/projects/Jcrop/demos.php
[105] http://requtize.github.io/cropimg
[106] https://github.com/matiasgagliano/guillotine



Cropper demo

You can check out the demo page of Cropper[107].

Be sure to read Cropper documentation[108] to know more about its features and what we can do with it.

Installing Cropper

Let's get started by downloading the latest release of Cropper[109].

Unzip (decompress) the downloaded file, and go to the **cropper-master/dist** directory.

Copy the **cropper.min.css** file to our **public/css** directory.

Copy the **cropper.min.js** file to our **public/js** directory.

Open our **master layout (app.blade.css)**, find:

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

Add above:

[107] http://fengyuanchen.github.io/cropper
[108] https://github.com/fengyuanchen/cropper
[109] https://github.com/fengyuanchen/cropper/archive/master.zip

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

Find:

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

Add below:

```
<script src="/js/cropper.min.js"></script>
<script src="/js/crop.js"></script>
```

Create a new file called **crop.js** and place it inside our **public/js** directory. This is our custom Javascript file.

Cropper is now ready to use!

Cropping an image using Cropper

Just for testing purposes, we'll be placing our **image cropping form** at the contact page.

Open **views/contact.blade.php**, here is the very beginnings of the file:

```
@extends('layouts.app')
@section('title', 'Contact')

@section('content')
    <div class="container">
        <div class="content">
            <div class="title">Contact Page</div>
                <div class="img-container">
                    <img id="image" src="/images/testimage.png">
                </div>
        </div>
    </div>
@endsection
```

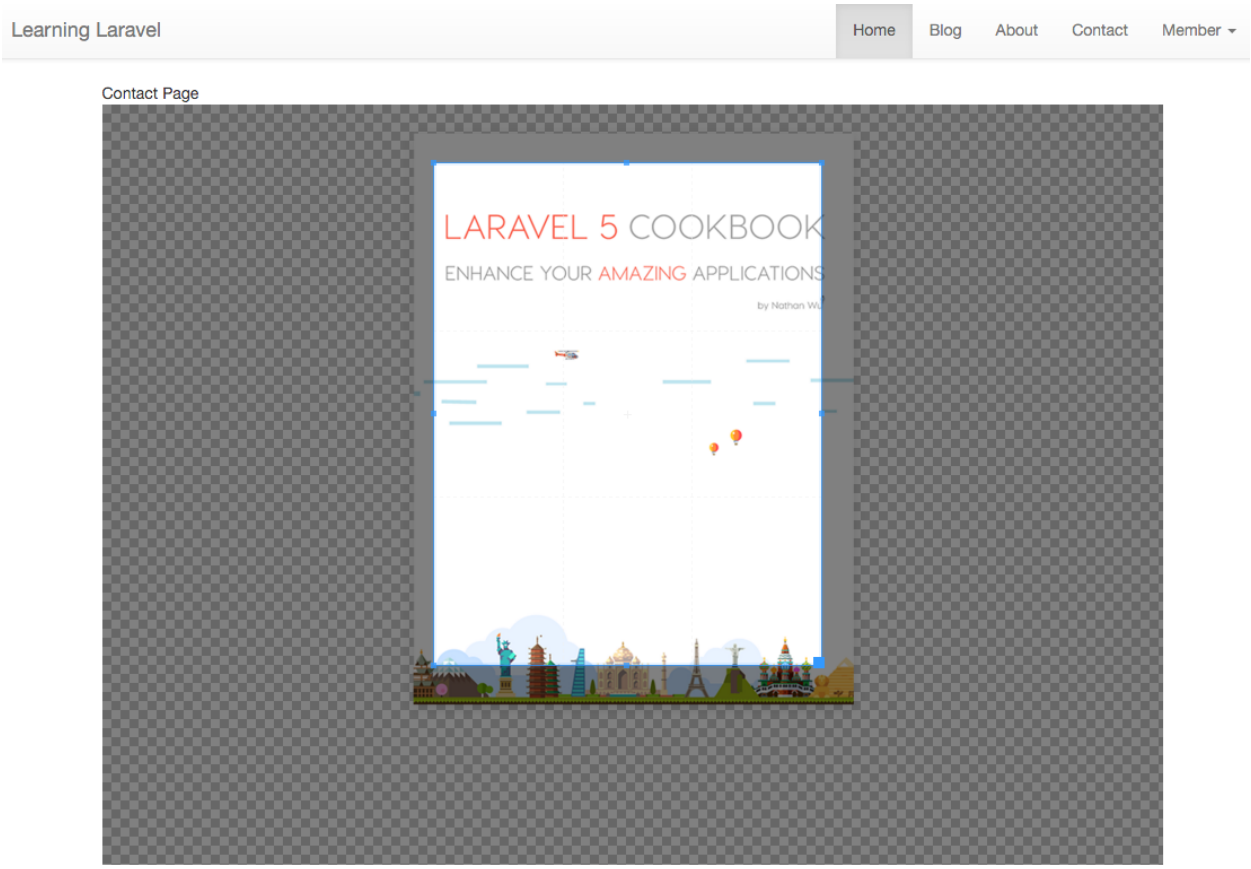
As you see, we just create a **normal image** and put it inside **a wrapper (img-container)**. The size of **the cropper** inherits from the size of the wrapper, so be sure to always wrap the image with a visible block element.

We can display a **new cropper** by adding the following to our **crop.js** file:

```
$(function () {

    'use strict';

    var $image = $('#image');
    $image.cropper();

});
```

We use jQuery to find the image. After that, we use **$image.cropper()** to initialize the cropper.

Now we can see the cropper in our browser!

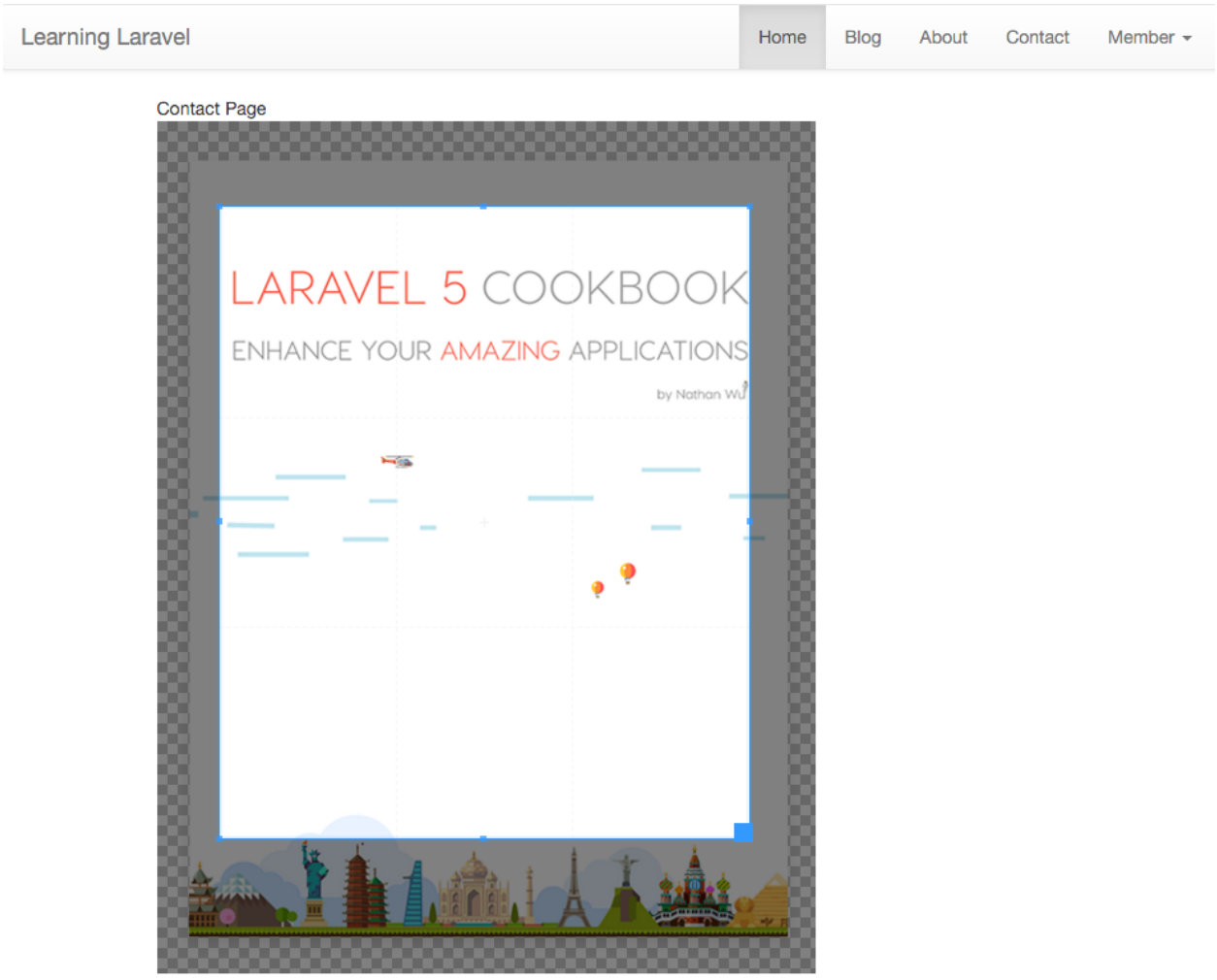


New cropper

As mentioned earlier, the size of the cropper is the wrapper's size. Let's change the cropper's size by **adding these CSS rules** to our **app.css** file:

```
img {
    max-width: 100%;
}

.img-container {
    margin-bottom: 20px;
    max-width: 516px;
}
```

Here is the new cropper:



New cropper with new size

Feel free to modify its size to your liking.

Now our cropper doesn't do much so let's give it a button to crop the image. Open **contact.blade.php**, find:

```
<div class="img-container">
```

Add above:

```
<button type="button" id="crop-btn" class="btn btn-primary">
    Crop Image
</button>
<div class="image-data"></div>
```

You should see a **Crop Image button** above our image.

Open the **crop.js** file, find:

```
$image.cropper();
```

Add below:

```
var croppingData = {};
$('#crop-btn').click(function() {
    croppingData = $image.cropper("getCroppedCanvas");
    $('.image-data').html(croppingData);
});
```

Cropper has a method called **getCroppedCanvas** that we can use to **get a canvas drawn the cropped image** when clicking the **Crop Image** button:

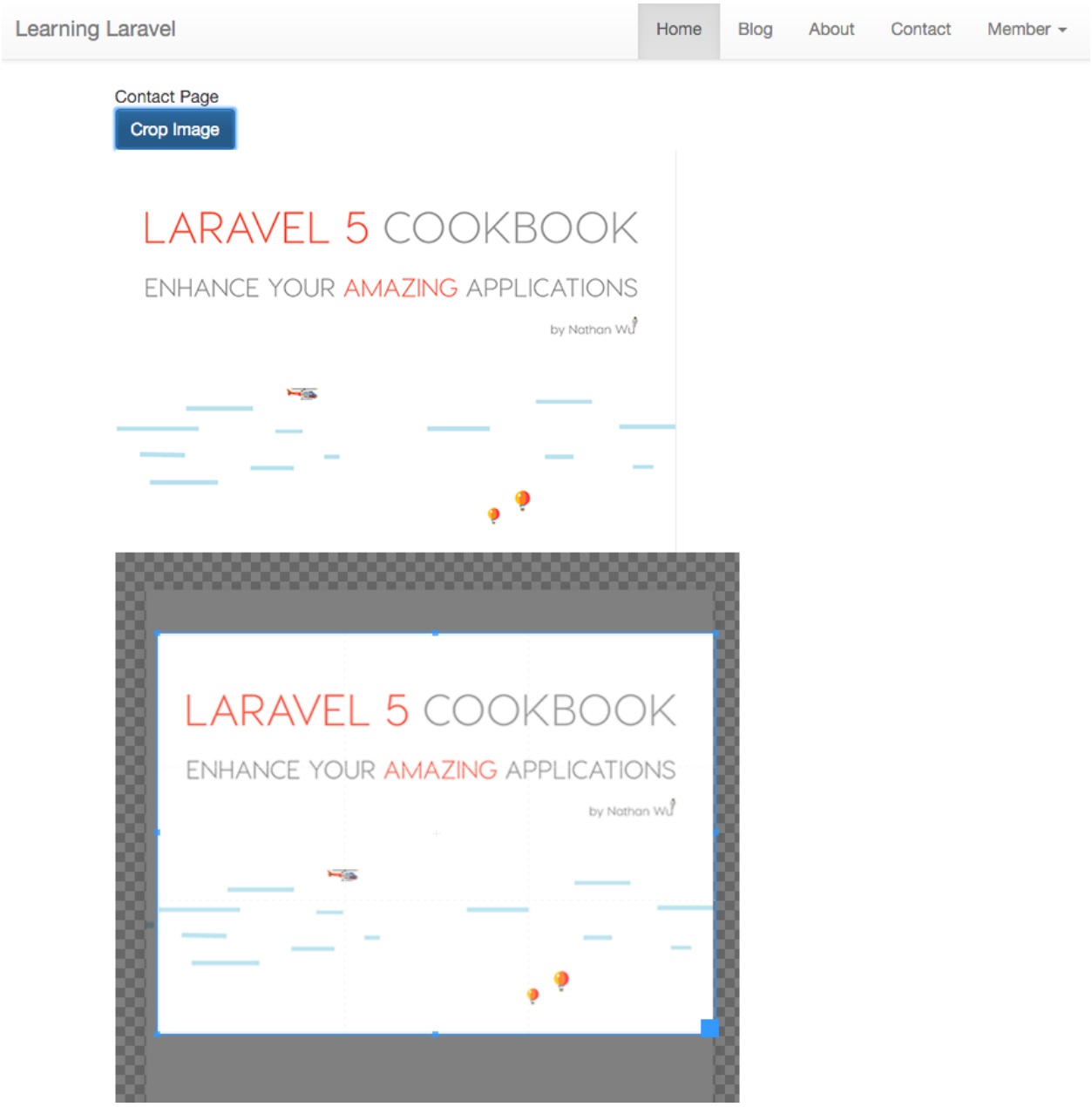
```
$('#crop-btn').click(function() {
    croppingData = $image.cropper("getCroppedCanvas");
});
```

Once having the **cropping data**, we can display the **cropped image** in the **image-data** section:

```
$('.image-data').html(croppingData);
```

Let's give it a try.

Change the cropped area position and then click the **Crop Image** button:



Click the crop image button

Great! Every time we click the **Crop Image** button, we should see a new cropped image immediately!

Uploading and cropping an image

Although we have only dealt with a single test image, this is the foundation for how we can add more related features to our application.

Let's say we wanted to let our users upload an image to our site and crop it. Here are the steps for building this feature:

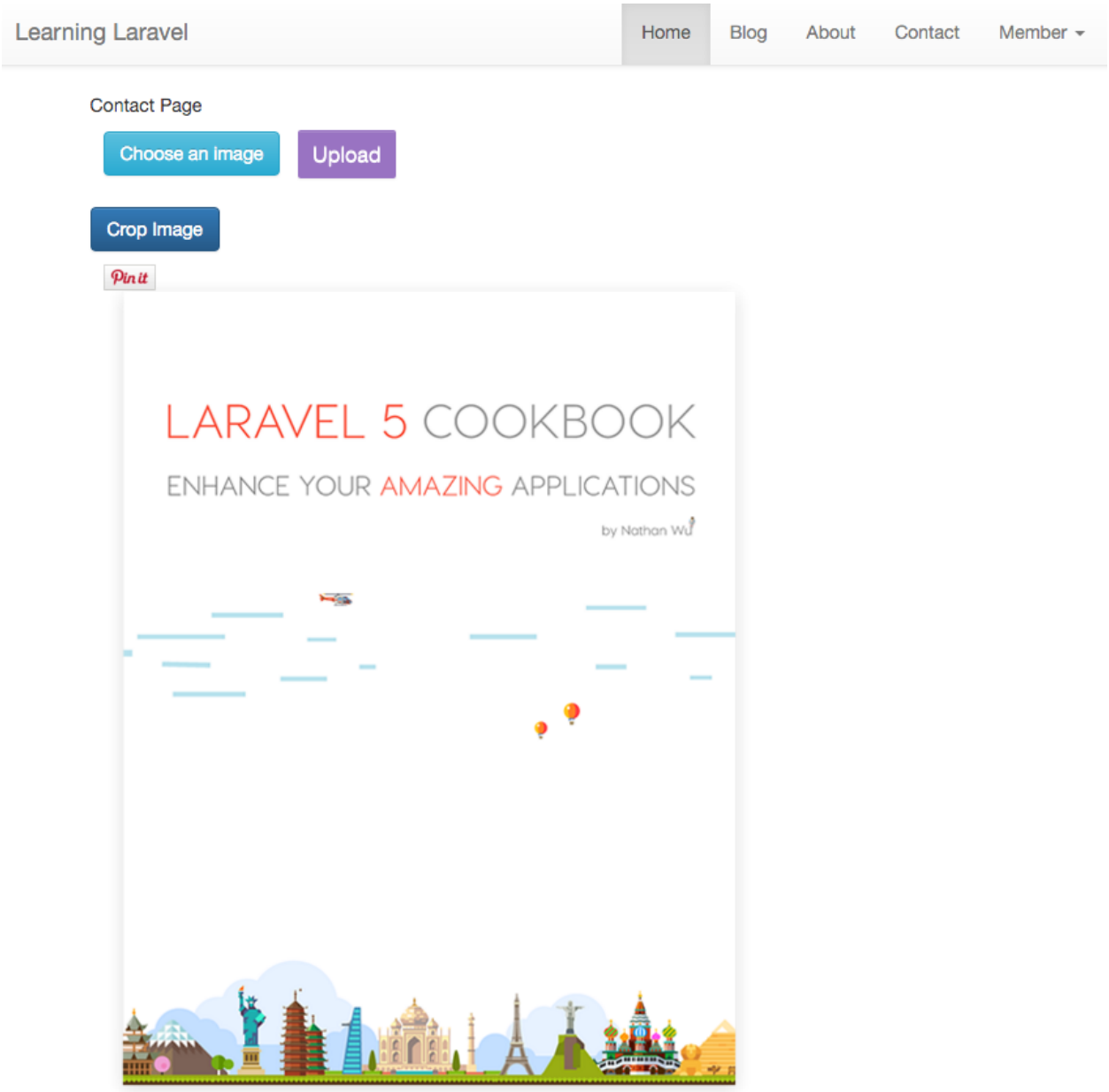
First of all, open our contact view (**contact.blade.php**), find:

```
<button type="button" id="crop-btn" class="btn btn-primary">
```

Add above:

```
<form method="POST" action="/cropimage" enctype="multipart/form-data">

    {!! csrf_field() !!}

    <input type="hidden" id="cropped-image" name="cropped-image" value="">

    <span class="btn btn-info btn-file">
    Choose an image
    <input id="uploaded-image" class="upload" type="file" name="uploaded-image" \
onchange="PreviewImage();"/>
    </span>

    <button type="submit" class="btn btn-default ladda-button" data-style="expan\
d-left" data-size="s" data-color="purple">
    Upload</button>

</form>
```

Now we should have a nice little form with some buttons:



Form buttons

Of course, we don't want users to click the **Crop Image** button if they don't choose any image yet. Let's temporarily hide the button!

Open our **app.css** file (public/css/app.css), and add:

```
#crop-btn {
    display:none;
}
```

Our **Crop Image** button should now be hidden.

When users click the **Choose an image** button, they can choose an image that they want to crop.

Once an image is selected, we will display the image inside a crop box. We can add this functionality by creating a new Javascript function called **PreviewImage**.

In our **crop.js**, add the function:

```
function PreviewImage() {
    var oFReader = new FileReader();
    oFReader.readAsDataURL(document.getElementById("uploaded-image").files[0]);
    oFReader.onload = function(oFREvent) {
        $('#crop-btn').show();
        $("#image").cropper('destroy');
        document.getElementById("image").src = oFREvent.target.result;
        $("#image").cropper();
    };
}
```

Here is how the **PreviewImage** function works:

```
    var oFReader = new FileReader();
    oFReader.readAsDataURL(document.getElementById("uploaded-image").files[0]);
    oFReader.onload = function(oFREvent) {
```

We use **FileReader** to get the **chosen image**.

```
$('#crop-btn').show();
```

Next, we display the **#crop-btn** button (**Crop Image** button).

```
$("#image").cropper('destroy');
```

Be sure that there is no **cropper** on the page by **destroying** it. If we don't do this step, when users select an image again, the new image will not be displayed.

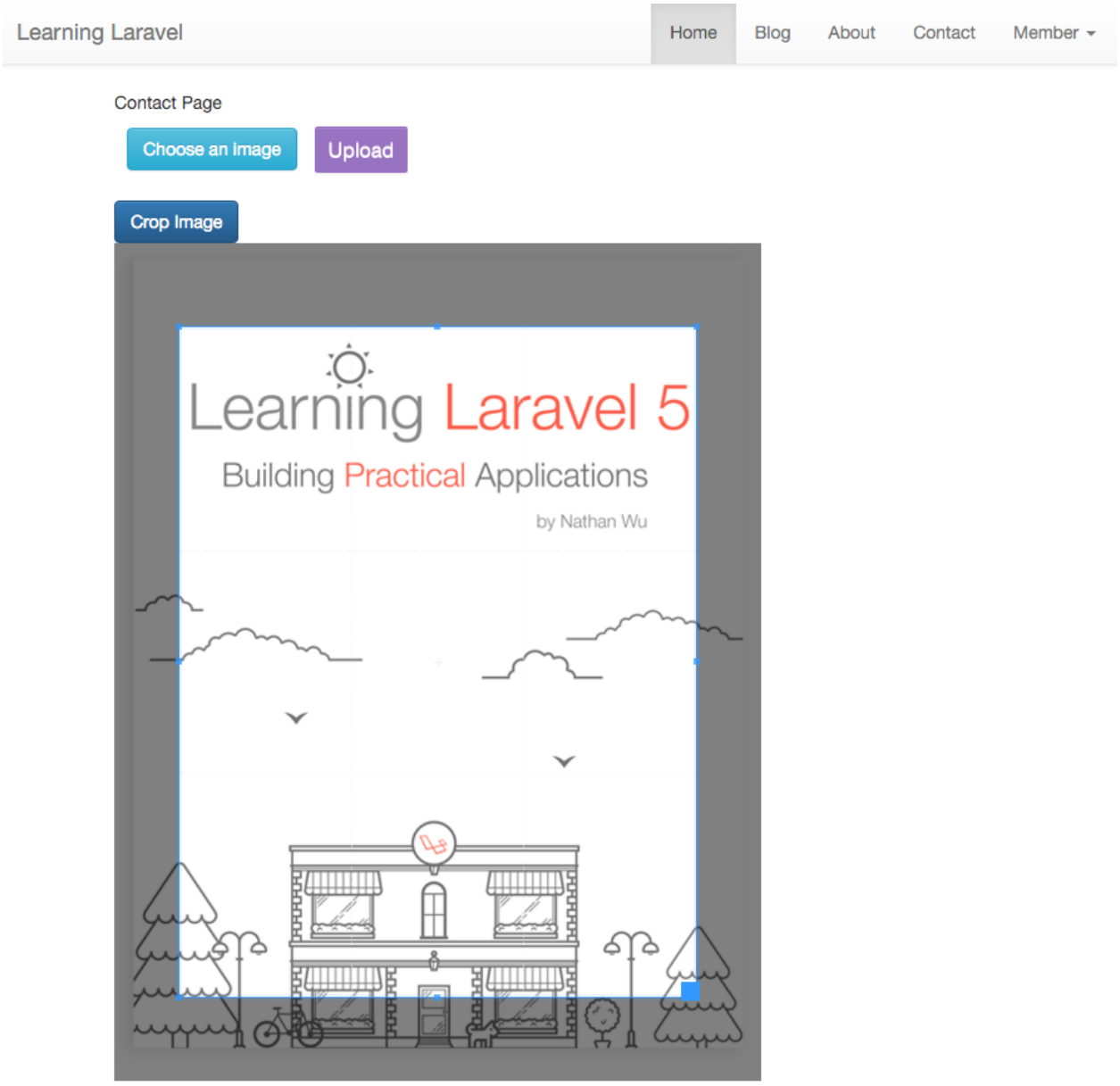
```
document.getElementById("image").src = oFREvent.target.result;
$("#image").cropper();
```

Finally, we **replace** the old image with the chosen one and **display a new cropper**.

The thing to notice here is that we must use the **onchange** event to trigger the **PreviewImage** function when a user selects an image:

```
<span class="btn btn-info btn-file">
Choose an image
<input id="uploaded-image" class="upload" type="file" name="uploaded-image" onch\
ange="PreviewImage();"/>
</span>
```

Go ahead and test the form:



Choose an image

If we click the **Choose an image** button and pick an image, then a cropper with our image will be displayed on the page!

The last thing we need to do is create **a hidden input** to **send our image data** to our server (backend) and **another action** to handle that request.

```
<input type="hidden" id="cropped-image" name="cropped-image" value="">
```

As you may notice, our form has the **hidden input** already, so just open the **crop.js** and update as follows:

```
$('#crop-btn').click(function() {
    croppingData = $image.cropper("getCroppedCanvas");
    $('.image-data').html(croppingData);
    $('#cropped-image').val(croppingData.toDataURL();
});
```

When we use **croppingData.toDataURL()**, the image will be converted to base64[110]. Simply put, base64 encoded data is a string of characters that contains our image data. We can decode that base64 data later to create a new image.

Next, add this route to our **web middleware group**:

```
Route::post('cropimage', 'ImagesController@storeCroppedImage');
```

Here is the **storeCroppedImage** action:

```
public function storeCroppedImage()
{
    $files = Input::all();

    if ($files['cropped-image'] != "") {

        $file = $files['cropped-image'];

        $destination = 'images';
        $filename = 'testimage';
        $extension = 'png';
        $fullName = $filename . '.' . $extension;

        $image = Image::make($file)->encode('png')->save($destination . '/' . $f\
```

[110] https://en.wikipedia.org/wiki/Base64

```
ullName);

        Alert::success('Image has been cropped successfully!', 'Success!')->auto\
close(2000);

        return redirect('/contact');

    } else if(isset($files['uploaded-image']) ) {

        $file = $files['uploaded-image'];
        $destination = 'images';
        $filename = 'testimage';
        $extension = 'png';
        $fullName = $filename . '.' . $extension;
        $image = Image::make($file)->encode('png')->save($destination . '/' . $f\
ullName);

        Alert::success('Image has been uploaded successfully!', 'Success!')->aut\
oclose(2000);

        return redirect('/contact');

    } else {

        Alert::error('There is an error', 'Error')->autoclose(2000);

        return redirect('/contact');
    }
}
```

Let's take a look deeper at this action so that we can see how it works.

```
    if ($files['cropped-image'] != "") {

        $file = $files['cropped-image'];

        $destination = 'images';
        $filename = 'testimage';
        $extension = 'png';
        $fullName = $filename . '.' . $extension;

        $image = Image::make($file)->encode('png')->save($destination . '/' . $f\
```

```
ullName);

        Alert::success('Image has been cropped successfully!', 'Success!')->auto\
close(2000);

        return redirect('/contact');

    }
```

We'll start by checking if our **cropped-image** field is empty. If it's not empty, we use **Intervention Image** to create a new image. We also use **Sweet Alert** to display a successful message. After that, the user will be redirected to the contact page.

```
    } else if(isset($files['uploaded-image']) ) {

        $file = $files['uploaded-image'];
        $destination = 'images';
        $filename = 'testimage';
        $extension = 'png';
        $fullName = $filename . '.' . $extension;
        $image = Image::make($file)->encode('png')->save($destination . '/' . $f\
ullName);

        Alert::success('Image has been uploaded successfully!', 'Success!')->aut\
oclose(2000);

        return redirect('/contact');

    }
```

If we don't get the **cropped-image** but we still get the **uploaded-image**, that means our users have uploaded an image but they haven't cropped it. We still save the image, display a successful message and redirect them back to the contact page.

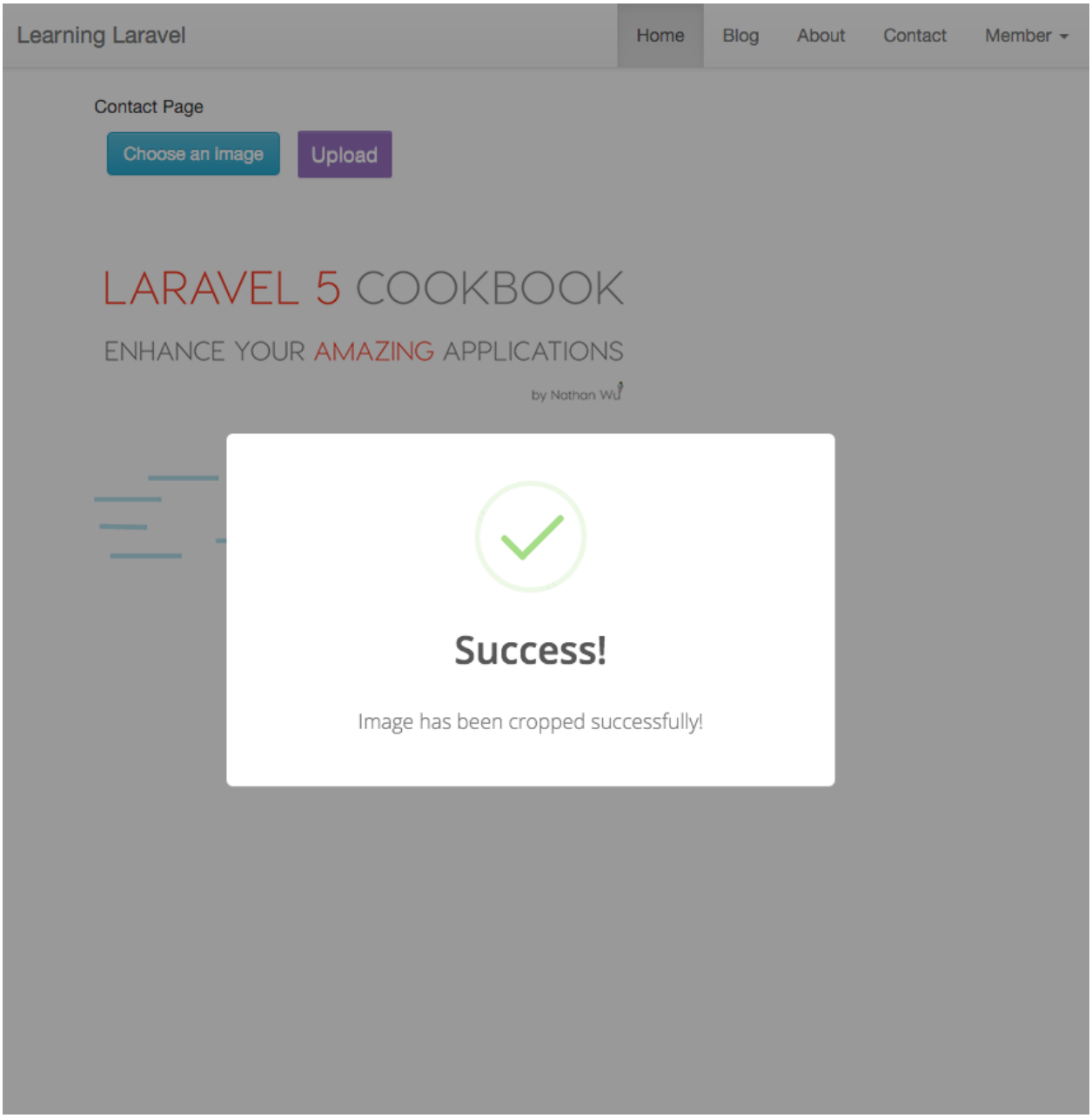
```
    } else {

        Alert::error('There is an error', 'Error')->autoclose(2000);

        return redirect('/contact');
    }
```

Lastly, if there is no image or there is an error, we just simply display an error message and redirect users back to our contact page.

Be sure that our **ImagesController** has all the required classes:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Requests\ImageFormRequest;

use Image;
use Illuminate\Support\Facades\Input;
use Response;
use Alert;

class ImagesController extends Controller
```

Well done! Go ahead and test your application! Let's make sure that everything is working properly.



Crop the image successfully

> **Note**: Sometimes, you may not see any changes. This is the **browser cache issue**. You have to manually refresh the page to clear your browser cache and see the new image. There are many ways to solve this issue: using a different image name, using Javascript to reload the page automatically, adding a timestamp to the image's name, redirecting users to a different page, etc.

Additionally, you may also set cropper options using **$().cropper(options)**. For example, you may

set some options as follows:

```
$("#image").cropper({
          aspectRatio: 200/200,
          resizable: true,
          zoomable: false,
          rotatable: false,
        });
```

Be sure to check out the documentation[111] to know more about other options.

Recipe 206 Wrap-up

> Tag: Version 0.14 - Recipe 206[112]

Congratulations! Now that you know the theory behind the cropping image functionality.

Don't forget to take advantage of all features of Cropper plugin to enhance your application. There is much more that you can now build using this technique. For example, you can use Cropper to get the height, width and x/y coordinates of the crop box, then crop the image at the backend.

Remember, this is just a beginning.

Have fun coding!

> The chapter is now complete. However, more recipes will be added later. Feedback from our readers is always welcome. Please leave your testimonials at http://learninglaravel.net/laravel[113]

[111] https://github.com/fengyuanchen/cropper
[112] https://github.com/LearningLaravel/cookbook/releases/tag/v0.14
[113] http://learninglaravel.net/laravel

Chapter 3: Deployment Recipes

Introduction

After learning some tricky topics to successfully build a full stack application, it's time to deploy your app. This chapter contains some helpful recipes about working with Heroku, Digital Ocean, etc.

Deploy your applications blazingly fast using GIT and secret techniques are also discussed in the book!

List Of Recipes

Deployment recipes

- Recipe 301 - Deploying your applications using DigitalOcean (PHP 7 and Nginx)
- Recipe 302 - Deploying your applications using Heroku
- Recipe 303 - Deploying your applications blazingly fast using GIT.

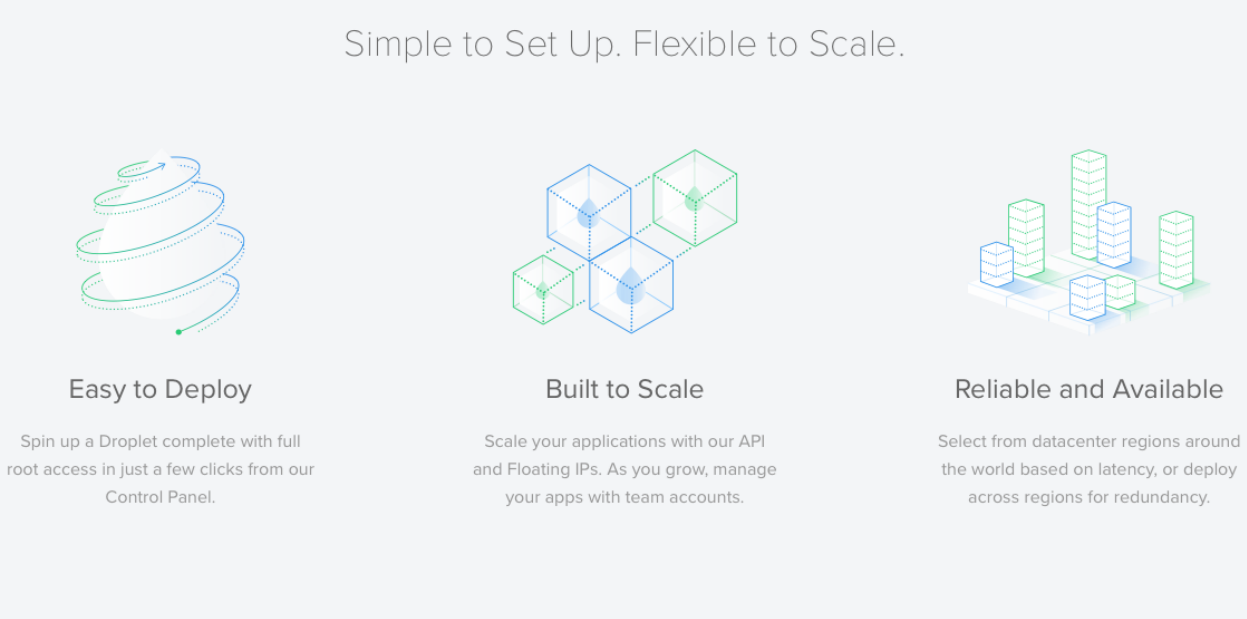
Recipe 301 - Deploying your applications using DigitalOcean (PHP 7 and Nginx)

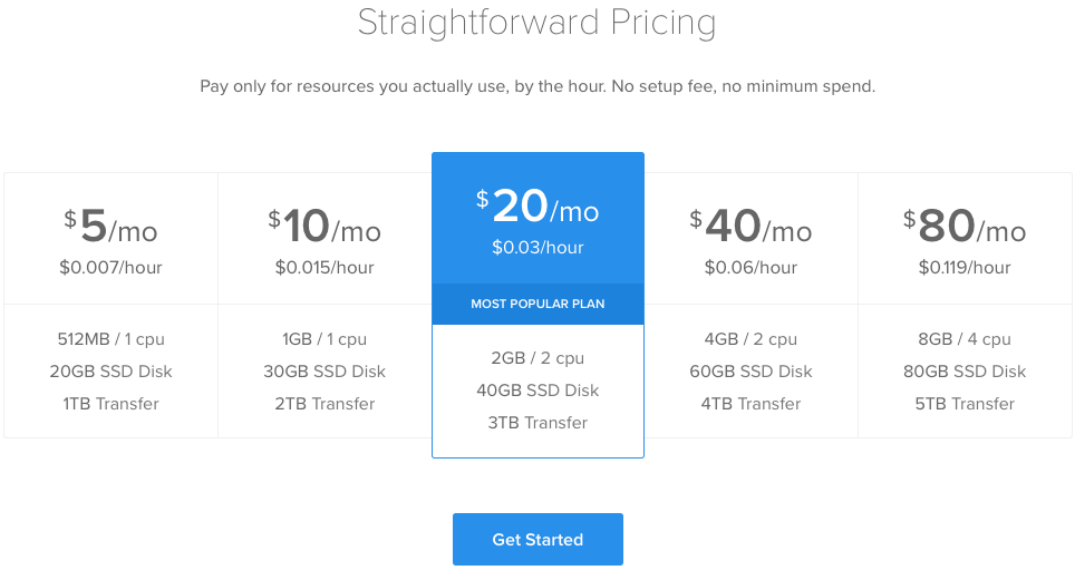
What will we learn?

This recipe shows you how to deploy your Laravel applications using DigitalOcean. In this recipe, we'll use PHP 7 (which is twice as fast as PHP 5) and Nginx 1.9.x.

All about 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**. Millions of amazing sites across the web are hosted on DigitalOcean!





DigitalOcean

If you're not a DigitalOcean member yet, you'll need to **register a new account** at DigitalOcean. You can use the link below to get **$10 for free**, that means you can **use their $5 cloud server for two months**.

Register a new DigitalOcean account and get $10 for free![114]

[114] https://www.digitalocean.com/?refcode=5f7e95cb014e

> **Note**: You will need to provide your credit card information or Paypal to activate your account.

Little tip: Learning Laravel also has a **freebies** section[115], you may find some useful coupons there.

Creating a new droplet (VPS)

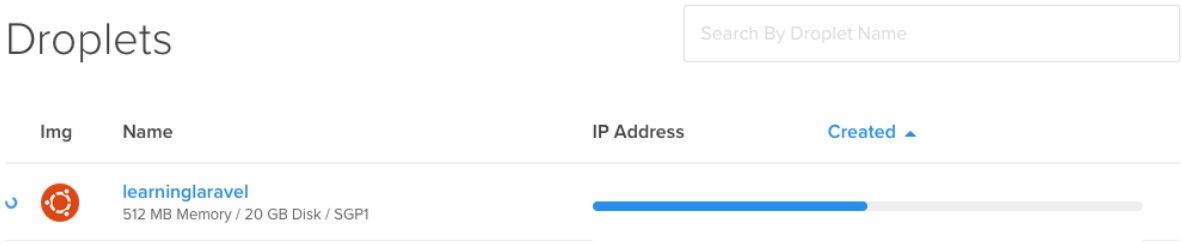
After your account has been activated. You will need to create a "**droplet**", which is a cloud server. Click on the **Create Droplet button** or go to:

https://cloud.digitalocean.com/droplets/new[116]

Follow these steps:

- At the **Choose an Image** section, be sure to choose **Ubuntu 14.04 x64**.
- Select your droplet size and region that you like (**$5/month** or **$10/month** is ok).
- At the **Choose a hostname** section, **name your Droplet** (For example, learninglaravel).
- You may skip other settings.
- Click "**Create**" to create your first cloud server!

> **Note**: There are some newer versions of Ubuntu (14.10, 15.04, etc.), but the 14.04 is an LTS (Long Term Support) version, that means we will receive updates and support for at least five years. Ubuntu 14.04 also has more compatible plugins. By the way, you may try to use a newer version if you want.



Creating a new droplet

Wait for a few seconds...

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.

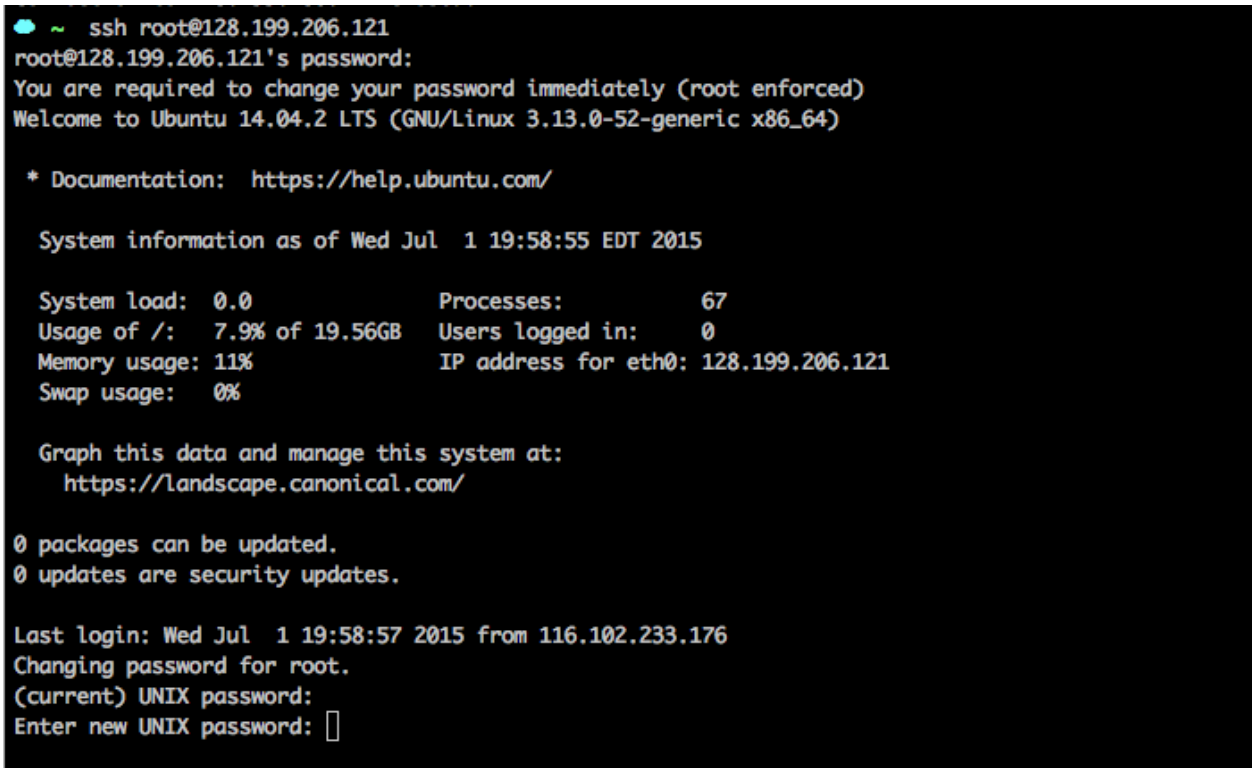
[115] http://learninglaravel.net/topics/freebies
[116] https://cloud.digitalocean.com/droplets/new

```
Droplet Name: learninglaravel
IP Address: 128.199.206.121
Username: root
Password: yourPassword
```

Installing PHP 7, Nginx and other packages

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

```
ssh root@yourIPAddress
```



Change your password

The first time you login, it will ask you to **change the password**. Enter **the current Unix password** again, and then **enter your new password** to change it.

Finally, run this command to check and update all current packages to the latest version:

```
apt-get update && apt-get upgrade
```

Say **Y (Yes)** if it asks you anything.

We're now ready to install PHP 7, Nginx 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**.

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 g\
it php7.0-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-\
opcache 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-mbs\
tring 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
```

```
43 php-redis - PHP extension for interfacing with Redis
44 php-memcached - memcached extension module for PHP5, uses libmemcached
45 php-apcu-bc - APCu Backwards Compatibility Module
46 php-amqp - AMQP extension for PHP
47 php7.0-bz2 - bzip2 module for PHP
48 php-rrd - PHP bindings to rrd tool system
49 php-uuid - PHP UUID extension
50 php-memcache - memcache extension module for PHP5
51 php-gmagick - Provides a wrapper to the GraphicsMagick library
52 php-smbclient - PHP wrapper for libsmbclient
53 php-zmq - ZeroMQ messaging bindings for PHP
54 php-igbinary - igbinary PHP serializer
55 php-msgpack - PHP extension for interfacing with MessagePack
56 php-geoip - GeoIP module for PHP
57 php7.0-bcmath - Bcmath module for PHP
58 php7.0-mbstring - MBSTRING module for PHP
59 php7.0-soap - SOAP module for PHP
60 php7.0-xml - DOM, SimpleXML, WDDX, XML, and XSL module for PHP
61 php7.0-zip - Zip module for PHP
62 php-tideways - Tideways PHP Profiler Extension
63 php-yac - YAC (Yet Another Cache) for PHP
64 php-mailparse - Email message manipulation for PHP
65 php-oauth - OAuth 1.0 consumer and provider extension
66 php-propro - propro module for PHP
67 php-raphf - raphf module for PHP
68 php-solr - PHP extension for communicating with Apache Solr server
69 php-stomp - Streaming Text Oriented Messaging Protocol (STOMP) client module for\
70  PHP
71 php-gearman - PHP wrapper to libgearman
```

> **Note**: You may remove some packages that you don't use and install them later when you need.

Once installed, you can now visit your **Nginx server** via the **IP address** (Be sure to use your droplet's IP address):

http://128.199.206.121[117]

[117] http://128.199.206.121

Creating a new droplet

We can check the installed version of PHP by using this command:

```
1 php -v
```

```
root@learninglaravel:~# php -v
PHP 7.0.4-7+deb.sury.org~trusty+2 (cli) ( NTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
    with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2016, by Zend Technologies
root@learninglaravel:~#
```

PHP 7's running

After that, we need to edit the **server block** (aka **virtual hosts**) file. Open it:

```
1 sudo nano /etc/nginx/sites-available/default
```

Find:

```
1 root /usr/share/nginx/html;
```

This is the path to your Laravel application, we don't have a Laravel application yet, but let's change it to:

```
1 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;
```

Change to:

```
index index.php index.html index.htm;
```

Find:

```
location / {
        # First attempt to serve request as file, then
        # as directory, then fall back to displaying a 404.
        try_files $uri $uri/ =404;
        # Uncomment to enable naxsi on this location
        # include /etc/nginx/naxsi.rules
}
```

Change to:

```
location / {
        # First attempt to serve request as file, then
        # as directory, then fall back to displaying a 404.
        # try_files $uri $uri/ =404;
        try_files $uri/ $uri /index.php?$query_string;
        # Uncomment to enable naxsi on this location
        # include /etc/nginx/naxsi.rules
}
```

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
```

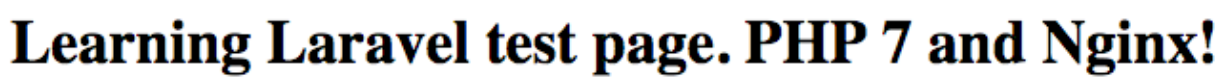
Here is the content of the **index.html** file:

```
<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:



PHP 7's running

Well done! You now have a working PHP 7 installation.

Installing Composer and 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**:

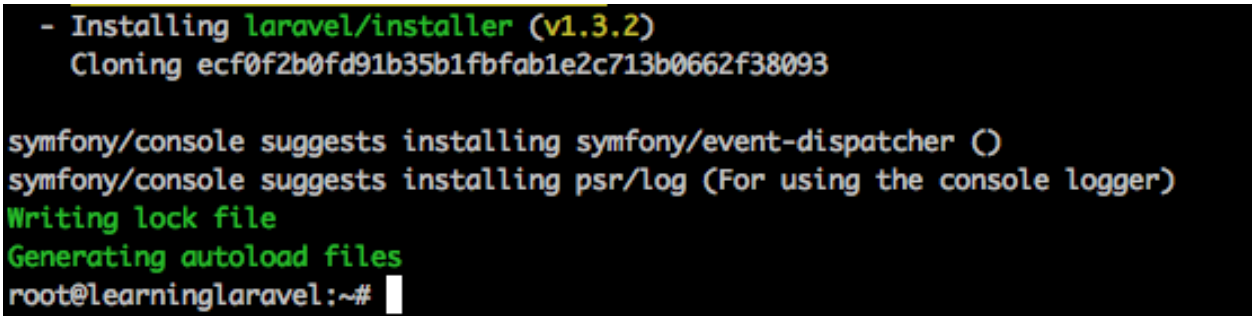
```
curl -sS https://getcomposer.org/installer | php
```

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"
```



Installing Laravel Installer

If you read the Laravel docs, you may see this:

"Make sure to place the ~/.composer/vendor/bin directory in your PATH so the laravel executable can be located by your system."

Let's do that by running these commands:

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

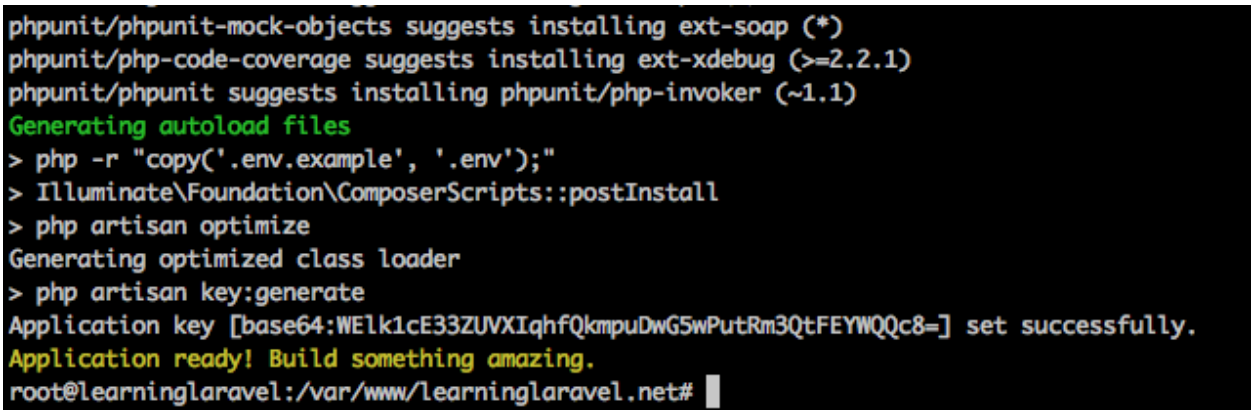
Once finished, we're finally at the part that we've been waiting for: **Installing Laravel**!

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:

```
laravel new laravel
```



Installing Laravel

This is a pretty standard process. I hope you understand what we've done. If you don't, please read Learning Laravel 5 book's Chapter 1[118].

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:

[118] http://learninglaravel.net/laravel5/installing-laravel

```
sudo nano /etc/nginx/sites-available/default
```

Find:

```
root /var/www/learninglaravel.net/html;
```

Change to:

```
root /var/www/learninglaravel.net/laravel/public;
```

Finally, restart Nginx:

```
service nginx restart
```

Go ahead and visit your Laravel app in browser:

Laravel is running

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
```

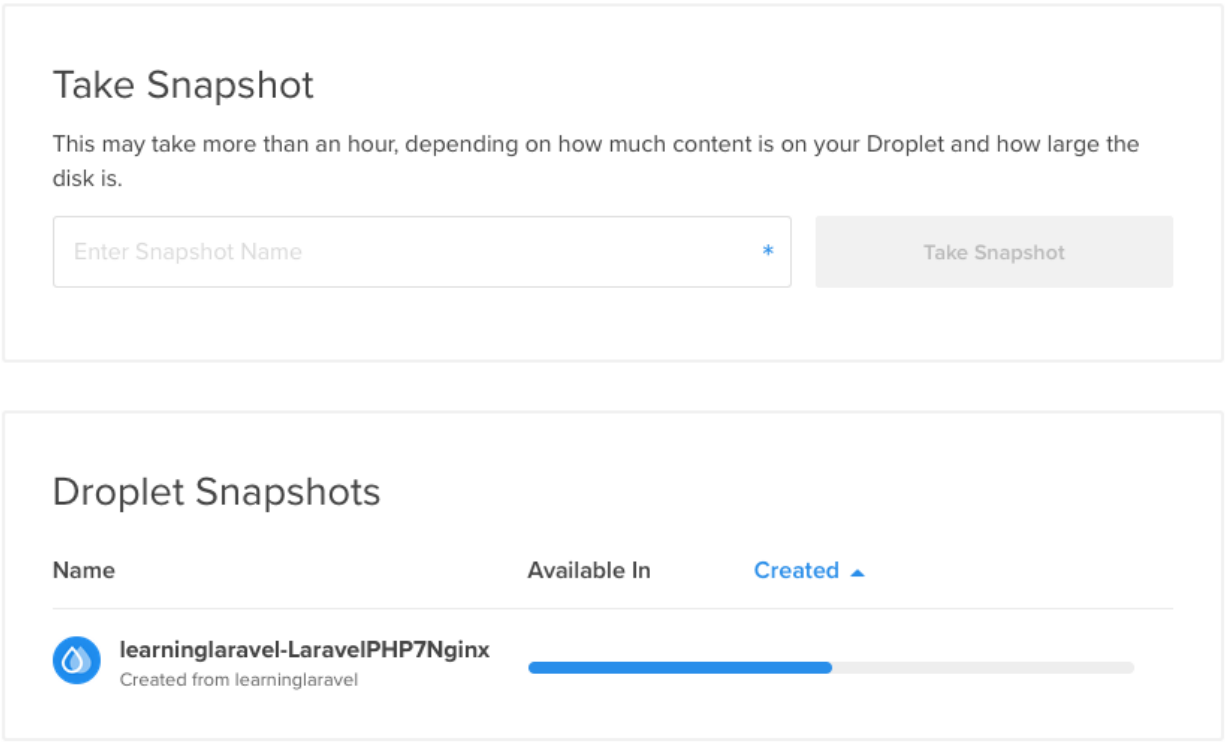
Take a snapshot of your application

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**.



Take a snapshot

Enter a name and then **take a snapshot**!

You may use this snapshot to restore your VPS later by using the **Restore Droplet** functionality.

Tips

Here are some little tips when using a droplet:

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:

```
1  server_name localhost;
```

Modify to:

```
1  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
```

Recipe 301 Wrap-up

Wonderful! We now have a Nginx web server running PHP 7!

The great thing is, Laravel 5.2 fully supports PHP 7! The performance of our sites should be improved.

Please note that this technique can be used to install Laravel on other Ubuntu servers, that means you can use other VPS services as well.

Recipe 302 - Deploying your applications using Heroku

What will we learn?

This recipe shows you how to install Laravel on Heroku for free.

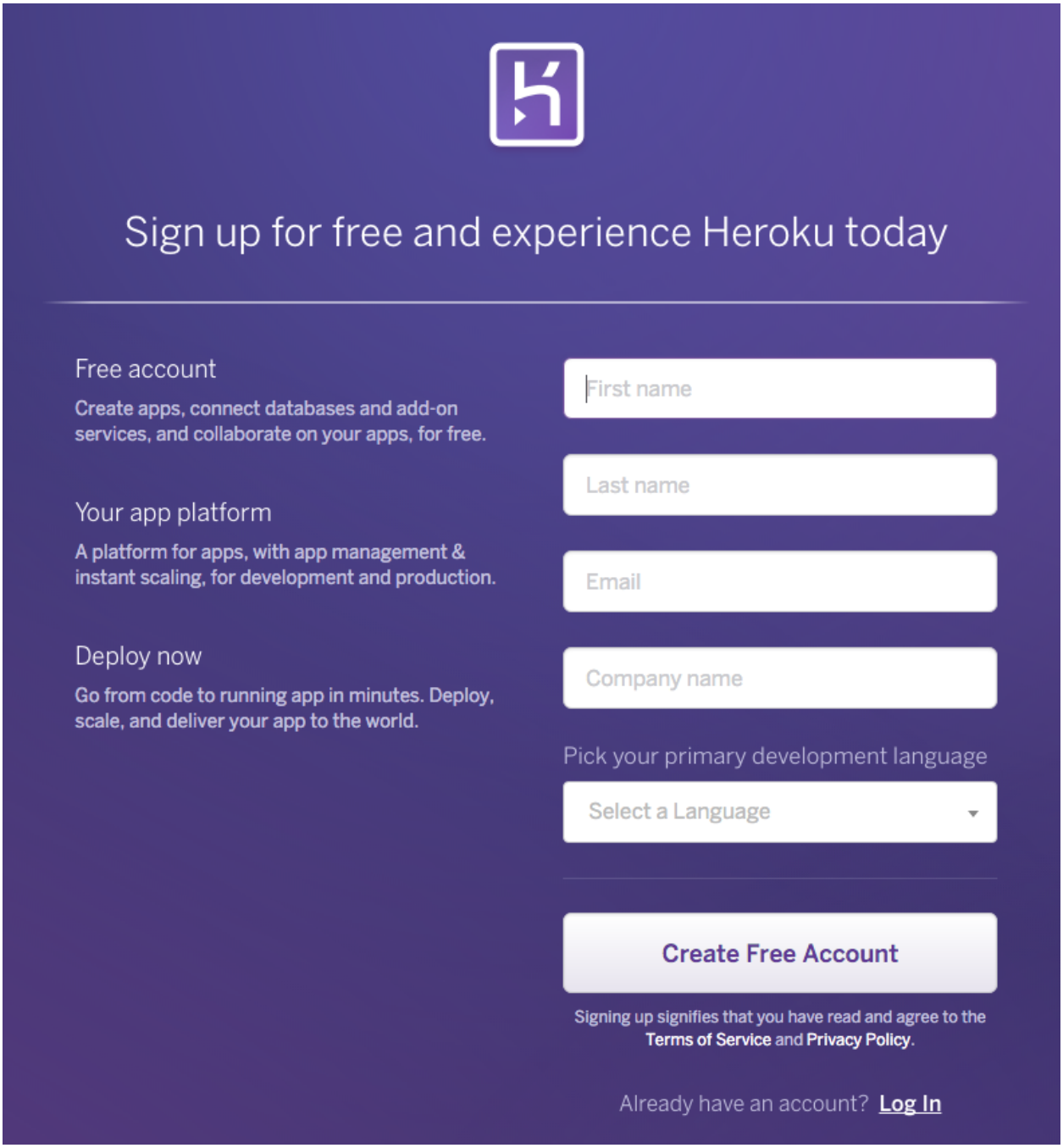
All about Heroku

Heroku is a popular cloud hosting service that supports PHP, Ruby, Java, and many other languages. One of the best features of Heroku is, we can register a Heroku account and deploy our applications to Heroku for free.

About the pricing, DigitalOcean is actually cheaper and I personally prefer DigitalOcean, but Heroku is still a good option if we just want to quickly test your applications on a production environment or show our projects to our friends and colleagues.

To use Heroku, we need to Register a Heroku account[119] first.

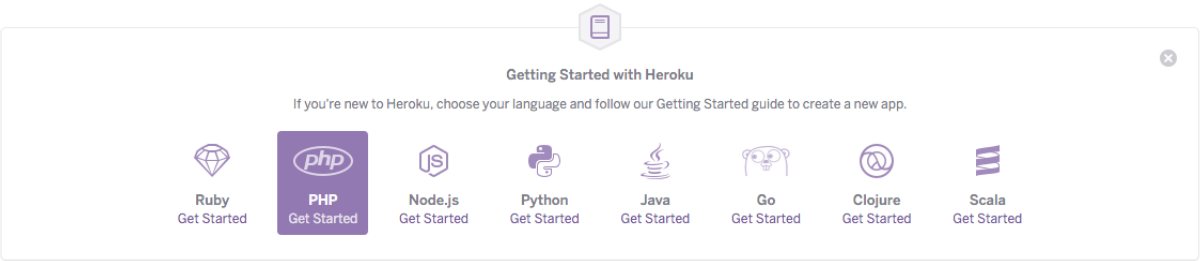
[119] https://signup.heroku.com



Register a Heroku account

Once having an account, we can choose **PHP** to get started.

> **Note**: Keep in mind that you can choose other languages later.



Register a Heroku account

Next, we must download and install Heroku Toolbelt[120], which is **a terminal utility** that provides you access to the **Heroku Command Line Interface (Heroku CLI)**.

You may choose and download **Heroku Toolbelt** for your system at the devcenter[121] or at the Heroku Toolbelt page[122].

Once installed, we can use the **heroku** command from our **command shell**:

```
1 heroku
```

[120] https://toolbelt.heroku.com
[121] https://devcenter.heroku.com/articles/getting-started-with-php#set-up
[122] https://toolbelt.heroku.com

```
heroku-cli: Adding dependencies... 10.2 MB/10.2 MB
heroku-cli: Adding dependencies... 4.34 MB/4.34 MB
heroku-cli: Installing core plugins... done
Usage: heroku COMMAND [--app APP] [command-specific-options]

Primary help topics, type "heroku help TOPIC" for more details:

  addons    #  manage add-on resources
  apps      #  manage apps (create, destroy)
  auth      #  authentication (login, logout)
  config    #  manage app config vars
  domains   #  manage domains
  logs      #  display logs for an app
  ps        #  manage dynos (dynos, workers)
  releases  #  manage app releases
  run       #  run one-off commands (console, rake)

Additional topics:

  2fa           #  manage two-factor authentication settings
  access        #  CLI to manage access in Heroku Applications
  buildpacks    #  manage the buildpack for an app
  certs         #  manage ssl endpoints for an app
  drains        #  display drains for an app
  features      #  manage optional features
  fork          #  clone an existing app
  git           #  manage local git repository for app
  help          #  list commands and display help
  keys          #  manage authentication keys
  labs          #  manage optional features
  local         #  run heroku app locally
  login         #  login with your Heroku credentials.
  logout        #  clear your local Heroku credentials
  maintenance   #  manage maintenance mode for an app
  members       #  manage membership in organization accounts
  orgs          #  manage organization accounts
  pg            #  manage heroku-postgresql databases
  pgbackups     #  manage backups of heroku postgresql databases
  pipelines     #  manage collections of apps in pipelines
  plugins       #  manage plugins to the heroku gem
  regions       #  list available regions
  spaces        #  manage heroku private spaces
```

The first time we use the **heroku** command, Heroku installs some **dependencies and plugins** for us, and then we'll see **a list of commands**.

Done! We can now use Heroku to deploy our applications.

Creating a new Laravel application

Just for testing purposes, we'll create a new Laravel application and then deploy it to Heroku later.

First, **SSH** into our **Homestead**:

```
vagrant ssh
```

Then navigate to our Code directory.

```
cd Code
```

Now let's create a new **laraheroku** app:

> **Note**: Feel free to change the name of the app to your liking.

```
laravel new laraheroku
```

Great! We should have a new Laravel application!

```
You are running composer with xdebug enabled. This has a major impact on runtime performance. See https://getcomposer.org/xdebug
> php artisan key:generate
Application key [base64:trchbNOb9jbqH8rz03/kLhMIybDIIcxHZi4zKMPx5tc=] set successfully.
Application ready! Build something amazing.
```

Heroku command

Now we need to write down or just remember **the application key**. We'll need this key later.

```
base64:trchbNOb9jbqH8rz03/kLhMIybDIIcxHZi4zKMPx5tc=
```

Last step, we'll need to **initialize a new Git repository**.

Be sure that we're at the **laraheroku's root**:

```
cd laraheroku
```

Initializing a new Git repo by using the following:

```
git init
git add .
git commit -m "My new laraheroku app"
```

Our code is ready to go!

Delploying to Heroku

We'll need to create a **Procfile**, which is a configuration file that tells **Heroku** about our applications' settings. Our Laravel applicationâ€™s root is the **public/ subdirectory**, so we have to **create a new Procfile** to serve the application from **/public**.

To begin, be sure that we're at the **laraheroku's root**.

Creating a new **Procfile** and add "**web: vendor/bin/heroku-php-apache2 public**" to the file by using the following:

```
echo web: vendor/bin/heroku-php-apache2 public/ > Procfile
```

Next, we'll add the new file to our Git repository:

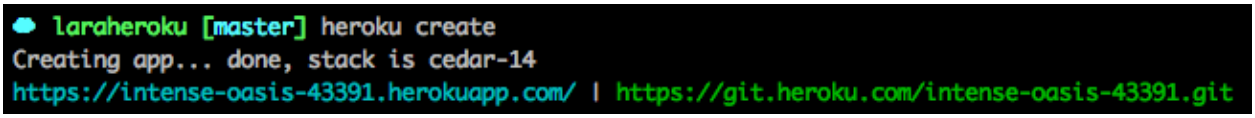
```
git add .
git commit -m "Procfile for Heroku"
```

Now we create a new Heroku application that we can push to, using this command:

```
heroku create
```

> **Important**: We've installed Heroku Toolbelt on **our system** (not **on Homestead**), so be sure that we run the **heroku create** command on our system (for example: ~/**Code/laraheroku**).

You may need to enter your **Heroku's credentials**.



Heroku command

As you see, a random name was automatically chosen for our application. https://intense-oasis-43391.herokuapp.com[123] is my **application URL**.

Heroku automatically detects our application is written in PHP. However, we should tell Heroku about that again, because sometimes it may not work as expected:

[123] https://intense-oasis-43391.herokuapp.com

```
heroku buildpacks:set heroku/php
```

Before deploying our app for the first time, we must set **a Laravel encryption key**, which is **the application key** used by Laravel to encrypt user sessions and other information.

We may use **heroku config:set APP_KEY=** command to do this:

```
heroku config:set APP_KEY=base64:eXJbtMeuhk3LtwIa7Xh4z1mEPQ4dgn3nT20aIsTZEkM=
```

> **Note**: Replace **base64:eXJbtMeuhk3LtwIa7Xh4z1mEPQ4dgn3nT20aIsTZEkM=** with your key.

We should have the key already when creating our new Laravel application. If you don't have the key, you can generate a new one by running this Artisan command (on Homestead):

```
php artisan key:generate --show
```

Finally, we can deploy our application to Heroku by pushing our files to the **Heroku Git remote** (https://git.heroku.com/intense-oasis-43391.git):

```
git push heroku master
```

```
remote: -----> Compressing...
remote:        Done: 13.3M
remote: -----> Launching...
remote:        Released v3
remote:        https://intense-oasis-43391.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy.... done.
To https://git.heroku.com/intense-oasis-43391.git
```

Deploy to Heroku

Head over to your Heroku application:

> **Note**: https://intense-oasis-43391.herokuapp.com[124] is my application, your **application's URL should be different**.

You may also use this Heroku command to open your application in a new window:

[124] https://intense-oasis-43391.herokuapp.com

```
1 heroku open
```

A new Laravel app

Congratulations! Your application is now running on Heroku!

Recipe 302 Wrap-up

This concludes our exploration of the Heroku cloud application platform. As you see, the deployment process is very straight forward and simple.

In addition, you may install a database by reading the Heroku ClearDB (MySQL alternative)[125] or Heroku Postgres[126] documentation.

Using Git to deploy our application is really great, right?

We can do the same thing when using our own server (DigitalOcean droplet or other VPS services). Let's learn about this technique in the next recipe.

[125] https://devcenter.heroku.com/articles/cleardb

[126] https://devcenter.heroku.com/articles/heroku-postgresql

Recipe 303 - Deploying your applications blazingly fast using GIT

What will we learn?

This recipe shows you how to deploy your applications using Git.

Creating a Git remote

In this section, I'll show you how to create a **Git remote**, which is a Git repository for our project. We can put this Git repository anywhere.

Absolutely, we can push changes to a Git remote and pull changes from it. That means, we don't need to use FTP to upload or download our files manually anymore. Git handles all the tedious processes for us. Additionally, Git automatically compresses all our files, so we can deploy our applications much faster.

Let's get started by creating a new Git repository first.

Login to your server via SSH:

```
1 ssh root@yourIPAddress
```

> **Note**: Let's assume that we're using DigitalOcean here.

Navigate to our site directory:

```
1 cd /var/www/learninglaravel.net
```

I will create a new directory called **repos** and put my Git repository there:

```
1 mkdir repos
2 cd repos
```

Now we can initialize a new Git repository by using the following:

```
1 git init --bare --shared learninglaravel.git
```

Next, go back to our site directory:

```
cd /var/www/learninglaravel.net
```

Assuming that our Laravel app will be installed at **/var/www/learninglaravel.net/laravel**, we'll use **Git clone** to clone the **learninglaravel.git** repository:

```
git clone /var/www/learninglaravel.net/repos/learninglaravel.git laravel
```

If you already have the **laravel** directory, you should remove it by running this command:

```
rm -r laravel
```

Now on our **local machine** or **Homestead**, go to our application directory:

```
cd Code/laravel
```

> **Note**: If you don't have the **laravel** directory yet, be sure to create a new Laravel application and name it **laravel**.

We can add a new **git remote called learninglaravel** here:

```
git remote add learninglaravel root@yourIPAddress:/var/www/learninglaravel.net/r\
epos/learninglaravel.git
```

> **Note**: if you're using **a different site name**, be sure to replace **learninglaravel** with your **site name**. Replace **yourIPAddress** with **your real server IP address** as well.

That's it for now!

Deploying our application to a VPS using Git

Once we have **a Git remote (learninglaravel)**, we can push our files to the server using Git:

```
git add .
git commit -a -m "Push files to the server"
git push learninglaravel master
```

Now our **learninglaravel.git** repository should contain all the files.

On our server, navigate to the **laravel** directory:

```
cd /var/www/learninglaravel.net/laravel
```

Next, we can use **git pull** to fetch **learninglaravel.git** repository and merge the changes into the **laravel** repository:

```
git pull origin master
```

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
```

> **Note**: We only need to do this one time.

To ensure that everything is working fine, let's visit our site:

A new Laravel app

Perfect! We've used Git to deploy our application!

Next time, if we make any changes or we want to upload files to the server, we can simply use these commands (On our local machine/homestead):

```
git add .
git commit -a -m "Update files"
git push learninglaravel master
```

And then use **git pull** to merge the changes (on our server):

```
git pull origin master
```

Recipe 303 Wrap-up

It took a bit of work, but we finally deploy our application to our server.

This technique is really great and it saves me a lot of time. Faithfully, I haven't used FTP in a long time. Git does the job better and faster.

Actually, you may use some third party services to deploy your applications, but Git is free and it's very easy to use.

Remember that, this is just a basic technique, you can do a lot more with Git.

I hope you enjoy reading my books a much as I enjoy writing them.

Happy learning and good luck!

Please send me your valuable feedback (support@learninglaravel.net) and leave your testimonial or review here[127].

[127] http://learninglaravel.net/laravel