LEARNING LARAVEL

THE EASIEST WAY

Learning Laravel: The Easiest Way

Jack Vo

©2014 Jack Vo

Contents

PART 1: BASIC INFORMATION

Welcome to Learning Laravel

Introduction

Hi! My name is Jack Vo. It's great to know that you're reading my book. I'm a designer and web/mobile game developer. I have over 7 years of development experience on projects ranging from small size applications through complex enterprise solutions.

I have developed a wide range of websites and mobile applications using HTML, PHP, Drupal, Wordpress, Corona SDK and Unity 3D. I've been also reading many books and watching many video tutorials about programming. Therefore, I'm sure that I can bring to you a book, that helps you learning Laravel easily.

In my opinion, Laravel documentation is good. However, if you're a new Laravel developer, it's not easy to follow. I will try to guide you through all the troubles and give you my best experiences! After reading this book, you can develop Laravel web applications fast and efficiently.

I write this book for beginners, developers of all levels. However, if you've known Laravel already, this book is still a good resource for you. Let's think it as a clean and clear alternative documentation.

The book is free, and it can be downloaded at the Learning Laravel website. If you love this book, feel free to donate money to help us continue to support the Laravel community.

A Special Thanks

Thank you for reading this book.

If you want to help me finding typos and other issues, or want to give some feedback, feel free to contact us at:

www.twitter.com/LearningLaravel[1]

or

www.facebook.com/LearningLaravel[2]

or

www.LearningLaravel.net[3]

Here's a list of people who have helped me to bring this book to you:

[1] http://www.twitter.com/LearningLaravel
[2] http://www.facebook.com/LearningLaravel
[3] http://www.LearningLaravel.net

- **Janie** - my special one.
- **Taylor Otwell** - without him, there is no Laravel Framework, there is no Learning Laravel book as well.
- **Jeffrey Way** - a great contributor to the community, I've learned a lot from him.
- **Peter Armstrong** - thanks for the awesome Leanpub!
- **You** - yes, you, thank you for supporting me.

Once again, thank you very much.

Translations

If you're interested in translating this book into a different language, please contact me at:

support@learninglaravel.net[4]

I will offer a 50/50 split of the the profits from the translated copy.

Structure of The Book

A note about this book

Please note that I may change the structure of this book and some contents in the future (adding and removing things) so make sure to keep a copy of it if you like that version:

Here's how things are organized:

PART 1: BASIC INFORMATION

- Welcome to Learning Laravel

This section will provide you a basic information about the book.

- Introducing Laravel 4

You want to know about the history of Laravel? Why should we choose it as our PHP framework? Let's find out! If you don't want, just skip this section, don't worry, you won't lose anything.

PART 2: BUILDING APPLICATIONS WITH LARAVEL

- Chapter 1 - Building Our First Website

We dive into building some simple web applications right away. It's the best way to learn Laravel. While some other programming books teach you the basic things first, I know all of us love to do something while we're learning.

[4] mailto:support@learninglaravel.net

- Chapter 2 - Building A To-do List Application

We have a fully responsive home page from Chapter 1. We will use it as a template for our To-do list application. In this chapter, we learn more about Laravel special features, such as: Blade Template, Schema Builder, Eloquent ORM, Controllers, Composer and Artisan.

PART 3 - AN ALTERNATIVE LARAVEL DOCUMENTATION

If you're a Laravel programmer, you can read this section as a documentation and learn more about it. If you're a beginner, don't worry, you will be a Laravel programmer soon. There are many design philosophies and principles in this section as well.

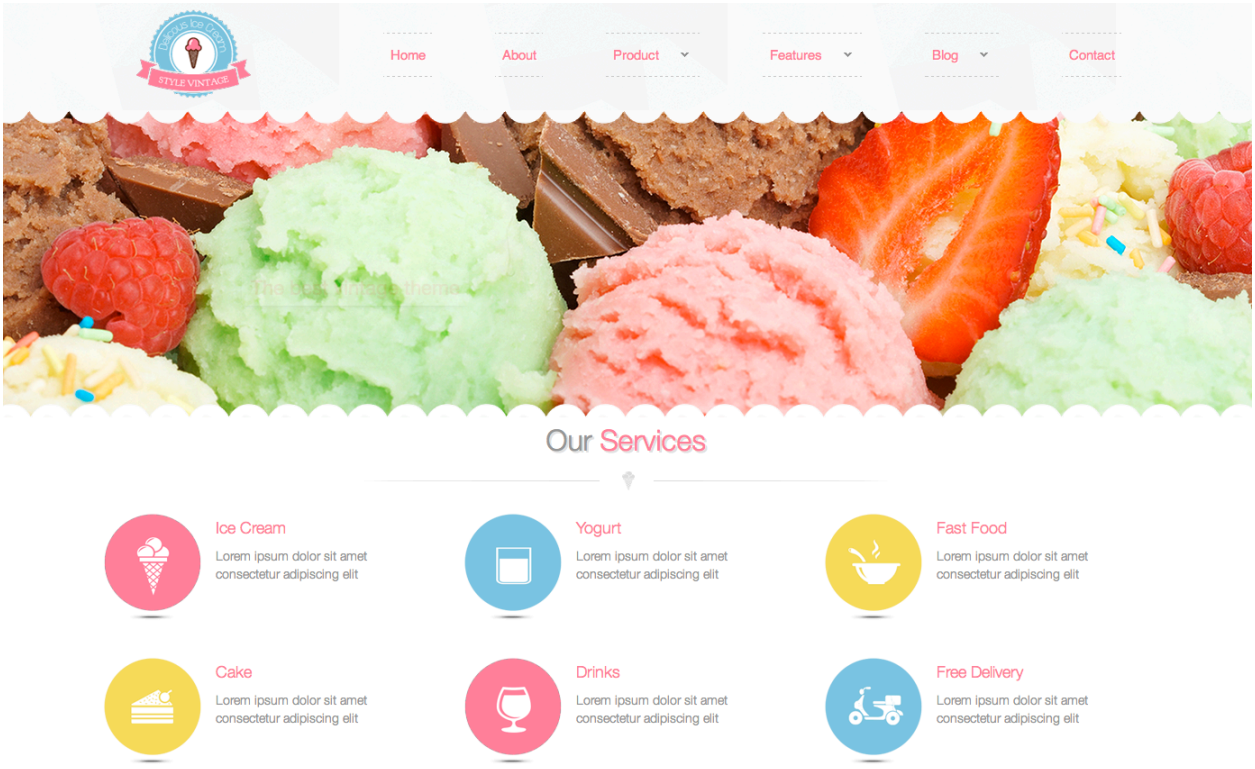
PART 4: LARAVEL CHEAT SHEET

If you want a cheat sheet for Laravel. Here it is.

PART 5: BUILDING A COMPLETE CMS FROM SCRATCH

- Chapter 2 - Building A Responsive Website From Scatch

Here we'll build a dynamic website using our knowledges from the book. In the end, our application looks like this:



Style Vintage Theme

This is one of my themes on Themeforest. If you love the theme, you can buy it at the link below right away. If you worry about the price, you can also send me a message, I will give you a big

discount:

Style Vintage Drupal Theme[5]

APPENDICES

- Basic HTML5, CSS3, Twitter BootStrap and PHP knowledge

If you don't know PHP, you don't even know how to code, this section is for you! You can skip this part if you like as well because it's for absolute beginners.

Revision History

A note about revision history

One important thing about the book is that, it's published while in progress. This means that the book is available in an incomplete state, but will grow over time into a complete title.

All **future updates** will be provided for **FREE**.

The current version of this book is **0.2**.

Version 0.1: Starting to write Part 1,2.

[5] http://themeforest.net/item/style-vintage-vintage-responsive-drupal-7-theme/5383210?ref=StyleMultimedia

Introducing Laravel 4

Say “Hi” to Laravel 4

Laravel 4 is an open source PHP framework, which is created by Taylor Otwell. I’ve been developing web applications for a long time, so I know that, developing a whole website from scratch can be very difficult and tedious. There are many potential bugs, and you have to think all complicated logic by yourself. A lot of works to do and it could take a lot of time. Fortunately, Laravel has come and saved us! Many developers around the globe are using its beautiful, clean code to create their great web applications.

You can start to build a Laravel application within a few minutes! It’s always a fun process. Laravel gives you right tools and nice ideas to help you build your website faster, more stable and very easy to maintain.

What can you create using Laravel? Well, a lot of things! From simple blogs to nice CMSs (Content Management System), eCommerce solutions, large scale business applications, social websites and more.

Laravel History

In 2011, Taylor Otwell, a great web developer has created an open source PHP framework, he called it Laravel. For only just 2 years, many developers around the world have been developing and using Laravel to build their applications. Laravel has come to version 4.1 today. It has many features such as built in session management, database management, Composer, Eloquent ORM and many more.

Laravel is a full stack framework, it means that you can develop web application completely from scratch using its amazing database wrapper called Eloquent ORM and its own templating engine called Blade. Many problems in the process of creating web application have been solved by Laravel. Laravel is a great tool, a great time saver to help you build things faster and faster. There are many reasons for using Laravel to develop web applications. One of the reasons is, Laravel has a welcoming and supportive community. Unlike Symfony or Zend framework, you can easily find many code snippets, tutorials, courses about Laravel. Even though the Laravel 4 has just been released a few months ago.

Laravel is not only Taylor Otwell’s product. It’s the product of a big community. It’s an open source framework, thus hundred developers worldwide have been providing many new features, bug fixes, ideas. You can easily ask questions in the community forum, or through Laravel IRC channel. We’re also building a learning Laravel center for you. If you’re not our member yet, feel free to register here:

Learning Laravel Official Website[6]

If you're a mobile developer, you find a right way to develop your web backend application. Laravel supports JSON very well.

The syntax of Laravel is very clean and easy to follow. The methods, functions are well defined. Sometimes you can even guess them without looking at the documentation. You can also create your own rules, your own way to write your code. Laravel gives you a lot of freedom. You can also maintain your code or upgrade it to a new version easily.

Laravel is a MVC Framework?

MVC (Model-View-Controller) pattern is very popular and many developers are using it for their application today. Laravel also loves the MVC. You can find folders called models-views-controllers inside Laravel. If you don't know about MVC, Laravel will help you to master it easily by just developing application with it.

So what is MVC? Basically, it's a architecture pattern that enforces seperation between models (information), controller (user's interaction) and view (models' display). Simply put, it helps to seperate your applications to many small parts in an organized structure. The main benefits of using MVC pattern is that, it helps you to change, extend and maitain your applications easily.

Want to learn more about MVC?

Don't worry about it too much, you can learn more about it in the later chapter.

Places to Learn Laravel

Laravel is a fast growing PHP framework. There are many places, books, tutorials to learn about it. You can find them here. I will try to update this section frequently.

- **Websites/Blogs**:

Learning Laravel[7] - our Official Twitter page for you to learn more about Laravel.

Tuts+ Premium[8] - a good place to learn Laravel.

Laravel.io[9] - Laravel knowledge base.

Laracasts[10] - Laravel Screencasts by Jeffrey Way.

[6] http://learninglaravel.net/
[7] http://twitter.com/LearningLaravel
[8] http://tutsplus.com
[9] http://laravel.io
[10] http://laracasts.com

Larasnippets[11] - Laravel snippets collected by John Kevin Basco.

Nettus+[12] - Great web development blog with lots of Laravel articles.

Laravel Tricks[13] - Sharing ways of using Laravel.

Laravel Official Doc[14] - Laravel official documentation.

- Books:

Learning Laravel: The Easiest Way by Jack Vo[15] - this book :)

Laravel Testing Decoded by Jeffrey Way[16] - Introduction to TDD, this book teaches you how to test your Laravel applications.

Code Bright by Dayle Rees[17] - A first book about Laravel 4. Code Bright contains many basic things you love to learn.

From Apprentice To Artisan by Taylor Otwell[18] - Written by the creator of Laravel, it's good for advanced developers. Covers dependency injection, interfaces, service providers, SOLID design, and more.

Implementing Laravel by Chris Fidao[19]- This book focuses on an overall approach to coding with Laravel, including code organization along with useful patterns for creating real-world testable and maintainable code.

Getting Stuff Done with Laravel by Chuck Heintzelman[20] - A guide taking you through application design, building console applications, and developing web applications.

---

[11] http://larasnippets.com
[12] http://net.tutsplus.com
[13] http://laravel-tricks.com
[14] http://laravel.com/docs
[15] https://leanpub.com/learninglaravel/
[16] https://leanpub.com/laravel-testing-decoded/
[17] https://leanpub.com/codebright/
[18] https://leanpub.com/laravel/
[19] https://leanpub.com/implementinglaravel/
[20] https://leanpub.com/gettingstuffdonelaravel/

PART 2: BUILDING APPLICATIONS WITH LARAVEL

Chapter 1 - Building Our First Website

Welcome to our first lesson about Laravel. We're going to build a small website in this chapter while we're learning some Laravel basic stuffs.

Getting start with some PHP frameworks could be a painful process with a steep learning curve. However, you will feel that it's very easy to begin with Laravel. This chapter will teach you to create your first web application with Laravel. You will learn how to use Composer, Artisan, Laravel application structure and some Laravel features such as: routing, templating, database operations and many more.

Installing Laravel

In this book, we will install Laravel on our localhost to develop the applications. Please head over to Part 3 to know how to install Laravel for your system, when you come back, we will start to code something.

Let's wait...1...2...3...

Ok, do you know how to install Laravel yet? If you encounter any problem, or you find a better way to install, feel free to contact us at:

www.learninglaravel.net[21]

As you know, we will use Composer or the new Laravel installer to install Laravel. I will create learningLaravel app. So if you've already created it following the instruction in Part 3, you don't need to do again. However, if you want, delete the app and install it one more time so you can remember how to do it. From now on, I will develop the app on Mac OS X. It's very similar on Windows, so don't worry.

What is Composer?

Laravel 4 consists of many components that are put together automatically into the framework through Composer. Using Composer, you can easily specify which other PHP libraries are used in your code, initiate installation and update of those dependencies through the command line. You can find more information about Composer in Part 3. Remember, Part 3 will be our documentation. If you need to find out something, go there. If you can't find the information, please wait a little bit, I will update it soon.

There are 2 methods to install Laravel: Use Composer or new Laravel Installer.

[21] http://learninglaravel.net

Method 1: Composer - To create a new app using Composer. Execute this command on Terminal (or Git Bash/Command Prompt on Windows):

```
cd desktop
composer create-project laravel/laravel learningLaravel --prefer-dist
```

As you see, we go to Desktop by using the cd command, then we let Composer do the magic work. Please wait a little bit. Composer will create a folder called learningLaravel. It's our web application. We will use it to develop our first website.

Method 2: Laravel Installer - To create a new app using the new Laravel Installer, execute this command:

```
cd desktop
laravel new learningLaravel
```

We also go to Desktop by using cd command, then we create our new app. In my opinion, we should use Laravel Installer method, because it's much faster.

After creating the app, you may need to use Artisan to "serve" your application. (in case you don't use WAMP or XAMPP)

What is Artisan?

Laravel comes with a command line interface called Artisan. We can use Artison to execute repetitive tasks. For example we can launch development server, create models, and run database migrations. You can run Artisan command through your system command line (Terminal on Mac, Git Bash on Windows).

You need to navigate to your application folder to run Artisan.

**Note: I put the app on my Desktop, so I navigate to the Desktop, your path may be different. If you install Laravel using XAMPP or WAMP (for Windows), follow the instruction in Part 3 to create learningLaravel folder.*

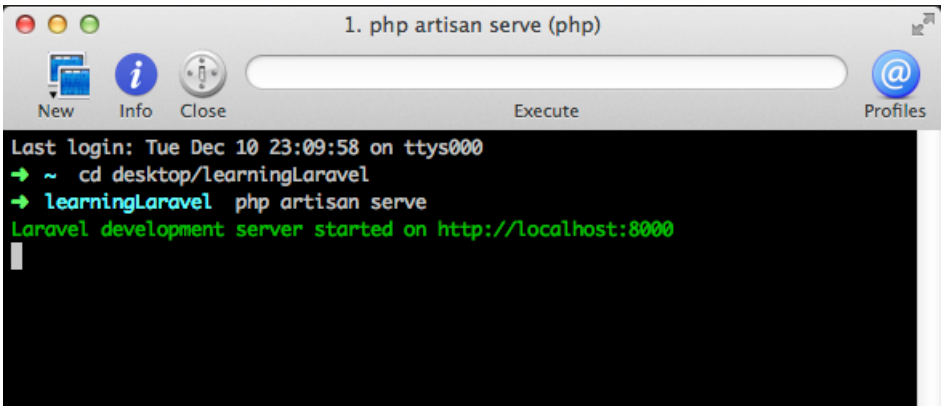
```
cd desktop/learningLaravel
```

Then run the command below:

```
php artisan serve
```

Output:

```
1 Laravel development server started on http://localhost:8000
```



The output through system command line

You can now open your web browser, go to localhost:8000 and see your application there.

Your site learningLaravel is running on localhost:8000

**Note: If you follow this section on Windows, go to learningLaravel.dev instead. (please follow instructions in Part 3)*

If you type:

```
php artisan
```

You should see a list of all commands available trough Artisan CLI. Executing commands through Artisan affects only the application that you navigated to.

Changing The Default Homepage

Ok, so what will we do next? How about changing the index page and print "Hello! Welcome to my first app!"? Let's do it.

Each website has certain endpoints that are reachable by typing a URL in your web browser address bar. For example, when you type google.com, you will go to Google's home page. If you type google.com/mail, you will go to Gmail!

Laravel called it "routes". Usually, we will have the home page (or index page) at a root route:

"/" : root route, application's home page.

"**about**" : about page.

We can go to app/routes.php to modify application's routes. So let's go to our learningLaravel app folder, find routes.php, open it with your text editor.

If you're using Mac, I recommend you to use Sublime Text. It's the best editor for Mac now.

You will see:

```
Route::get('/', function()
{
return View::make('hello');
});
```

We modify it to:

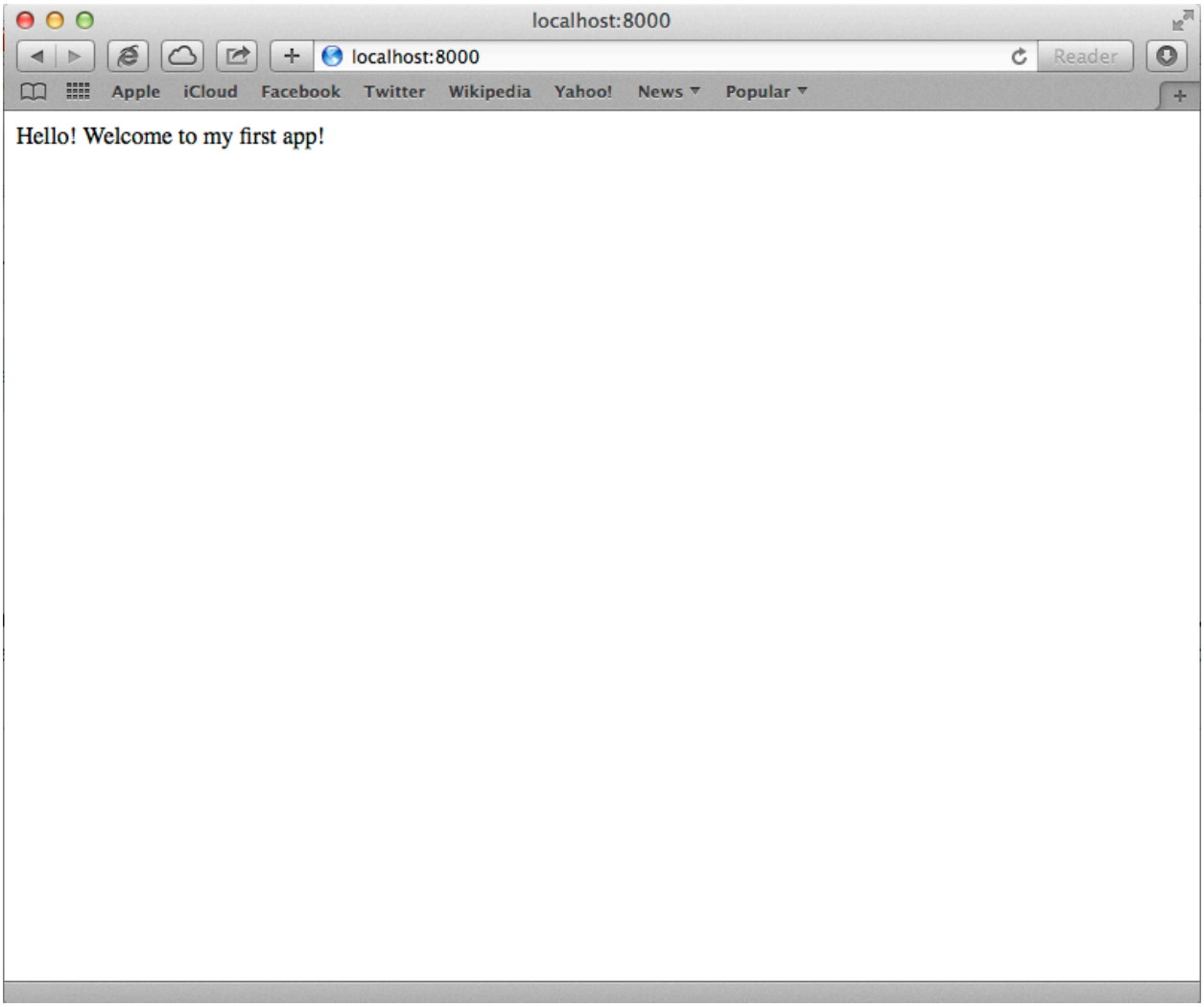
```
Route::get('/', function()
{
return 'Hello! Welcome to my first app!';
});
```

Save the file and we go to our web browser:

http://localhost:8000[22]

Well done! You should see:

[22] http://localhost:8000



A new home page! YAY!

Creating Other Pages

Let's imagine that we're going to build a simple website for our company. The website will consist of three pages:

-**Home page**: there will be a welcome message here and a menu for navigation.

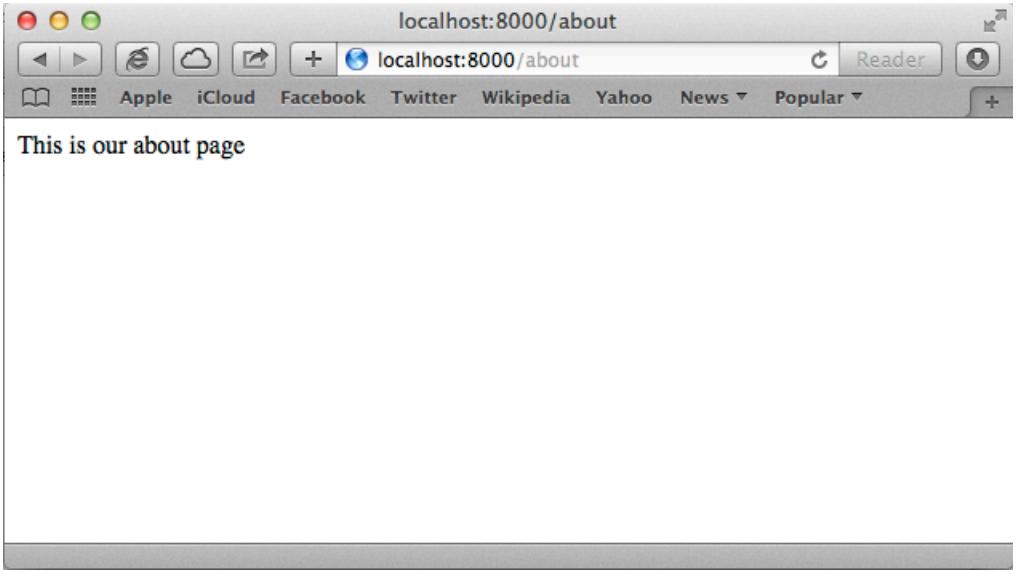
-**About page**: some information about our company.

-**Contact page**: our company contact information. This page contains a contact form. Users can contact you via email.

With the idea in mind, we're going to create other pages. Let's go to app/routes.php again. Insert:

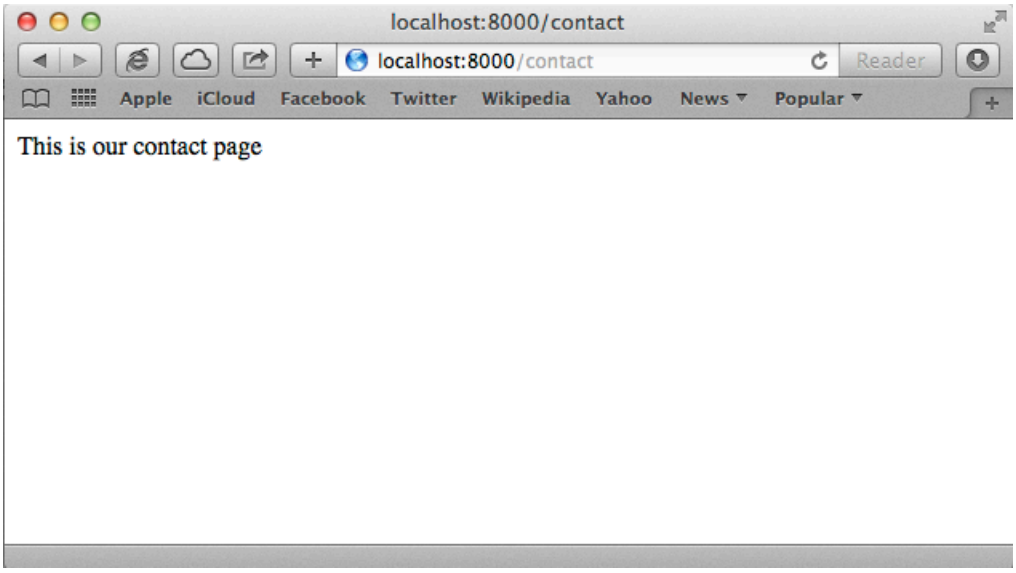
```
Route::get('/', function()
{
        return 'Hello! Welcome to my first app!';
});

Route::get('/about', function()
{
        return 'This is our about page';
});

Route::get('/contact', function()
{
        return 'This is our contact page';
});
```

We just define our application's routes. Go ahead and try the two new routes in the web browser. You should see:



About page

and



Contact page

Creating HTML and Blade Views

At the moment, we only show users the blank screens with a single line of text. It's boring. Let's improve that by creating HTML templates.

Laravel comes with a poweful templating engine called Blade. If you know HTML, you can use Blade easily. Blade looks like plain HTML, but Taylor Otwell has added some special statements into it to make it output PHP data, loops, use different layouts and many more.

Every Blade file ends with **.blade.php**. You can find the templates in **app/views** directory.

The best feature of Blade is an ability to use a layout for all the pages of our web application. For example, you can create a main menu in a master layout, and then use it for all your pages. We will build a layout, which allows us to keep common elements for our pages. Let's make a new file named **layout.blade.php** in our **app/views** directory.

```
1  <!doctype html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Learning Laravel Website </title>
6  </head>
7  <body>
8      <ul>
9          <li><a href="./">Home</a></li>
10         <li><a href="./about">About</a></li>
```

```
        <li><a href="./contact">Contact</a></li>
    </ul>
@yield('content')
</body>
</html>
```

We have the layout for the site! If you know HTML, you will easily understand all. Let's look at the one within the body first, that's the easy one:

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

We use **@yield('content')** to tell Blade to create a section here, we can fill in content later. We also give it the nickname 'content', so we can refer back.

Creating Blade Template For Home Page

Now create a template for our home page by making the new **home.blade.php** file and fill in these codes:

```
@extends('layout')
@section('content')
<h1>Welcome!</h1>
<p>Learning Laravel is a learning center. You can find everything about Laravel h\
ere.</p>
@stop
```

Let's take a look at:

```
@extends('layout')
```

This tells Blade which layout we will be using to render our content. The name that we pass to the function should look like those that we pass to **View::make()**. In this situation we are referring to the '**layout.blade.php**' file within '**app/views**' directory.

Now that we know which layout we are using, it's time to fill in some contents. We can use the **@section** function to inject content into sections. It looks like this:

```
@section('content')
        <h1>Welcome!</h1>
        <p>Learning Laravel is a learning center. You can find everything about Laravel \
here.</p>
@stop
```

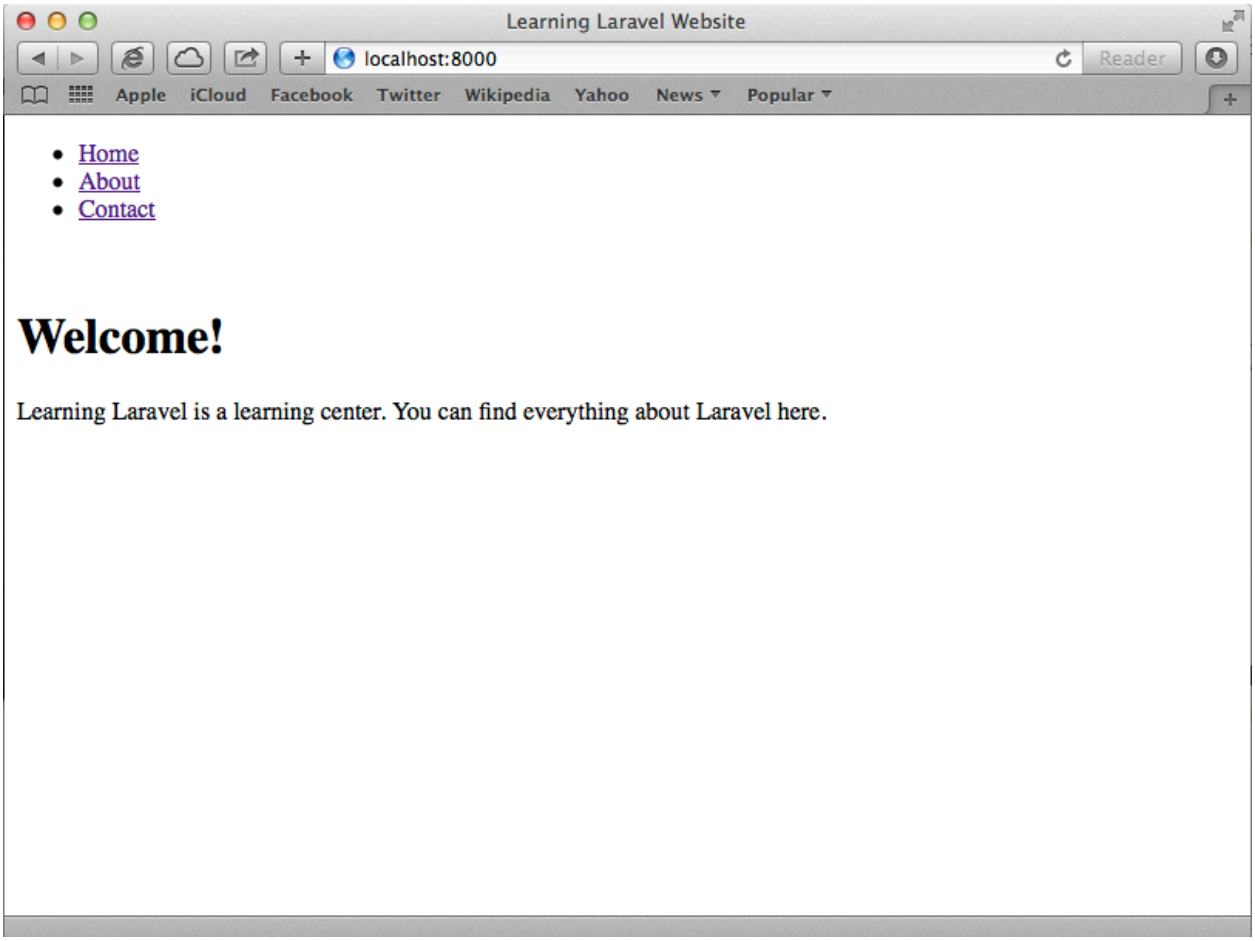
We pass the '**@section**' function the nickname that we gave our section within the parent template. The nickname is **content**. Everything that is contained between '**@section**' and '**@stop**' will be injected into the parent template, where the '**@yield('content')**' is.

Last step, edit **app/routes.php** file. This is how it will look like after replacing the text string output:

```
Route::get('/', function()
{
        return View::make('home');
});
```

We edit our home route and to render the template, we need to add a **View::make()** response to render the child template. When we put **View::make('home')**, it will render **home.blade.php** within our '**app/views**' directory.

Finally, go to our website, you will see:



Our new home page

Creating Blade Template For About Page

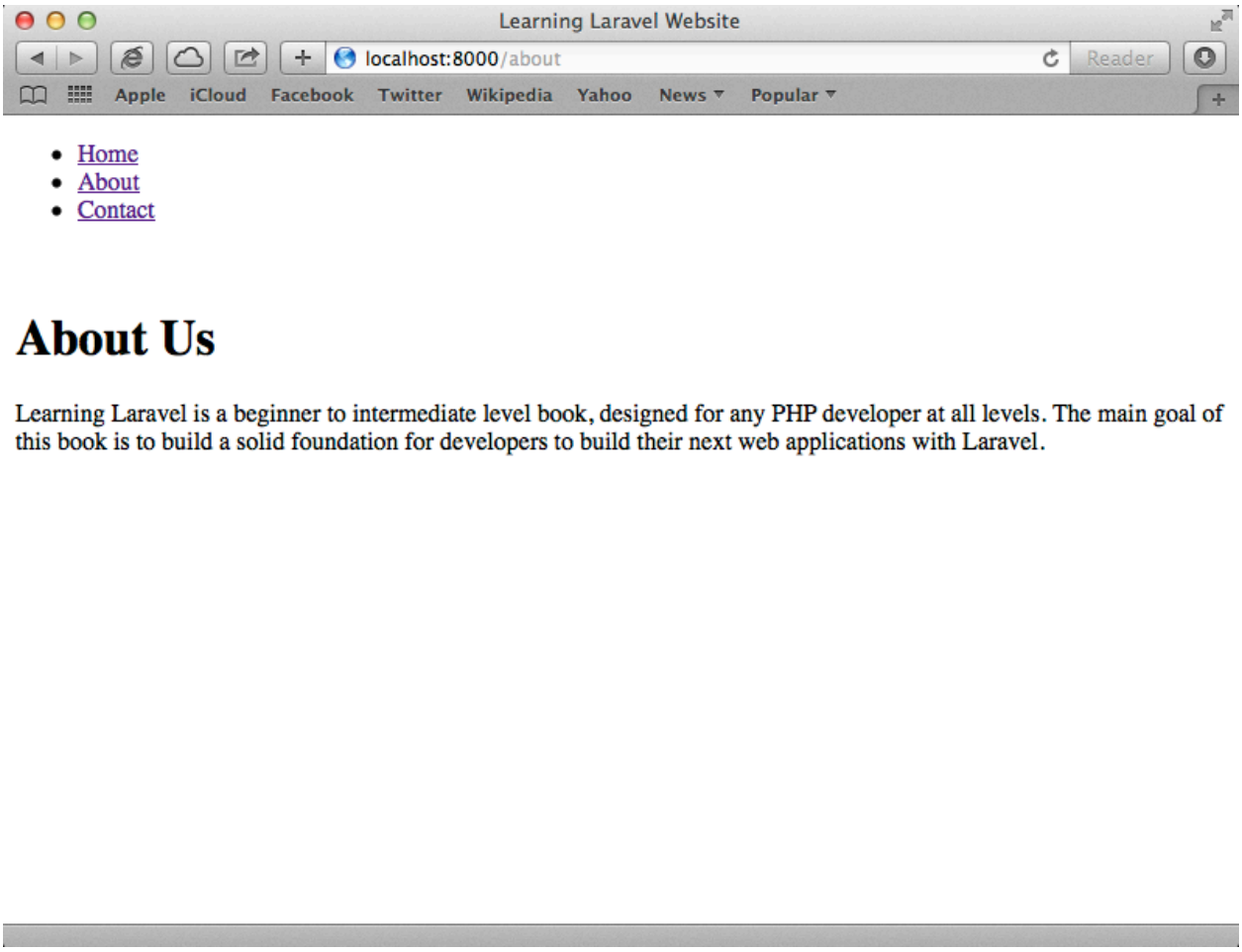
It's time for practice. Now, trying to create about.blade.php yourself! It's very easy:

```
1         @extends('layout')
2         @section('content')
3         <h1>About Us</h1>
4         <p>Learning Laravel is a beginner to intermediate level book, designed for any P\
5 HP developer at all levels. The main goal of this book is to build a solid founda\
6 tion for developers to build their next web applications with Laravel.</p>
7         @stop
```

Because we've created the **layout.blade.php** already. We can just use **@extends** to render it and inject some contents. After that, don't forget to modify the **routes.php** file:

```
Route::get('/about', function()
{
        return View::make('about');
});
```

We just create our new about page using Blade template within a few minutes:



Our new about page

Creating Blade Template For Contact Page

The only page left for us to make is the contact page. We will make a contact form for users to contact us via email. Laravel is amazing. It provides multiple shortcuts for creating the forms. We can create labels, buttons, text inputs and everything we need to make forms. We can still use normal HTML to make forms, but it's easier and cleaner to do using Laravel method. Let's create a template called **contact.blade.php**, then placing some codes inside to make our contact form:

```
@extends('layout')
@section('content')
        <h1>Contact Us.</h1>
        <p>Please contact us by sending a message using the form below:</p>
        {{ Form::open(array('url' => 'contact')) }}
        {{ Form::label('Subject') }}
        {{ Form::text('subject','Enter your subject') }}
        <br />
        {{ Form::label('Message') }}
        {{ Form::textarea('message','Enter your message') }}
        <br />
        {{ Form::submit() }}
        {{ Form::close() }}
@stop
```

With a few lines, we can create our form. To create our opening form tag, we use the **Form::open()** generator method. This method accepts only an array.

```
{{ Form::open(array('url' => 'contact')) }}
```

In the above example we are using the URL index, the value of the route is where we want to link to. You can see that our form is targeting the /contact route and will by default use the POST request verb.

We use the **Form::close()** method to close the form. We can also use **</form>** to close it.

We need labels so that people know what values to enter. Labels can be generated using the **Form::label()** method:

```
{{ Form::label('Subject') }}
```

Text fields can be used to collect string data values. Let's take a look at the generator method for this field type:

```
{{ Form::text('subject','Enter your subject') }}
```

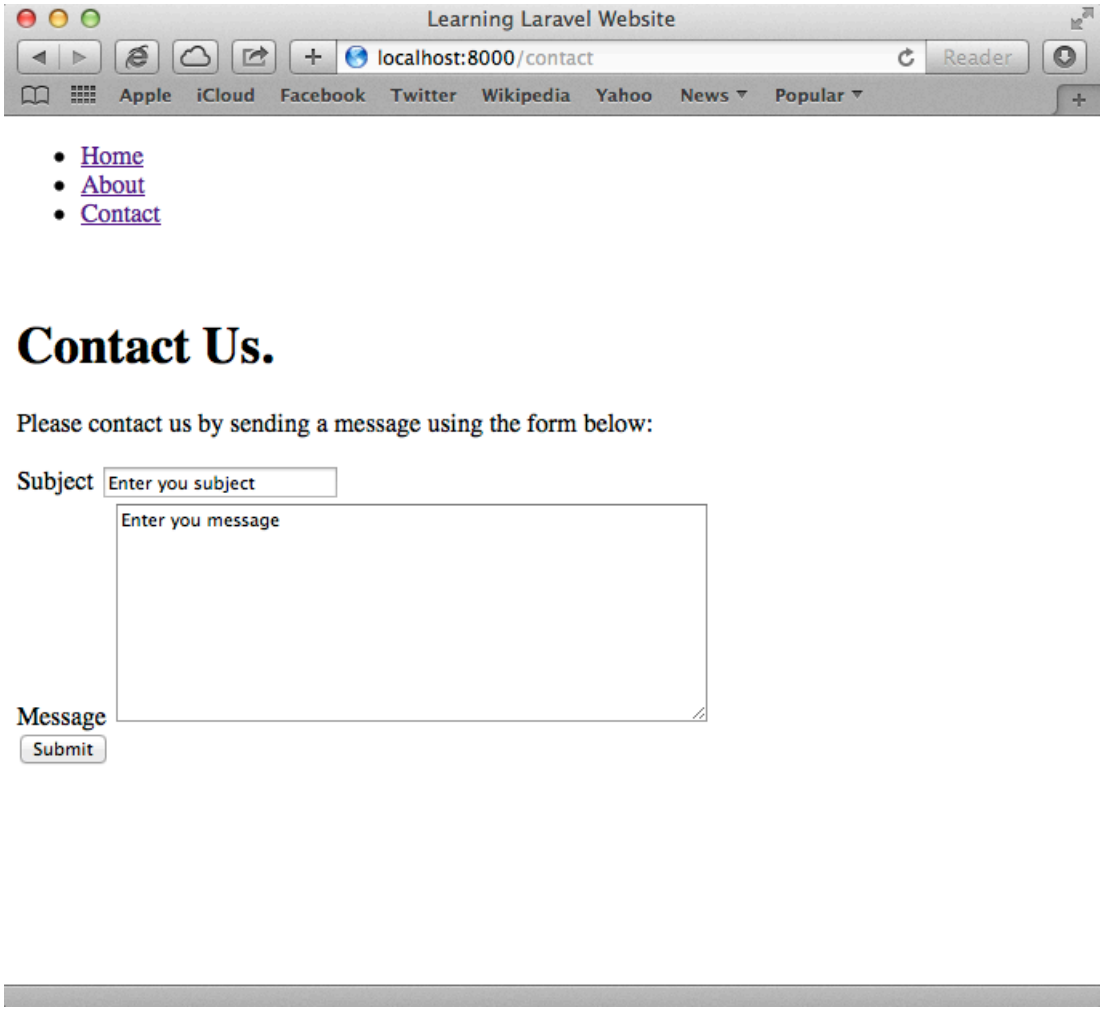
The first parameter for the Form::text() method is the name attribute for the element. The second parameter is optional. It is a default value for the input element.

```
{{ Form::textarea('message','Enter your message') }}
```

The textarea field function is similar to the text field function.The first parameter is the name attribute for the element, and the second parameter is the default value for the input element.

```
{{ Form::submit() }}
```

Finally, the submit button can be generated easily by the code above. Simple, right? If you love to learn more about Laravel forms, you can always go to Part 3 and look for the documentation there. If everything is ok, we will have a contact page look like this:



Our new contact page

Add Validation to Our Contact Page

We need to apply validation to the form to ensure that the data entered by users is valid. If we don't check, the data could be bad, and it could cause severe problems to our application or even a security vulnerability.

Laravel provides us a set of validation rules, and some methods to apply validation easily. Open our **app/routes.php** file. Add these codes:

```
Route::post('contact', function()
{
$data = Input::all();
$rules = array(
        'subject' => 'required',
        'message' => 'required'
);

$validator = Validator::make($data, $rules);


if($validator->fails()) {
        return Redirect::to('contact')->withErrors($validator)->withInput();
}

        return 'Your message has been sent';
});
```

Ok, let's break them apart for easier to understand:

```
Route::post('contact', function()
{
$data = Input::all();
```

We use **Route::post('contact')** to handle submissions from our form, we get all the inputs and assign them to $data variable.

```
$rules = array(
        'subject' => 'required',
        'message' => 'required'
);
```

A set of validation rules takes the form of an associative array (which is $rules). The array key represents the field that is being validated (subject and message). The array value will consist of rules that will be used to validate (required). For example, if we want the users to type something into the subject field, we just put:

```
'subject' => 'required'
```

The "**required**" is a rule, and it can be used to ensure that we will get subject's values. Take a look at this line:

```
$validator = Validator::make($data, $rules);
```

We can use the Validator::make() method to create a new instance of our validator. The first parameter to the make() method ($data) is an array of data that will be validated. The second parameter to the method ($rules) is the set of rules that will be used to validate the data.

After our validator instance has been created, we can use it to do something:

```
if($validator->fails()) {
        return Redirect::to('contact')->withErrors($validator)->withInput();
}
```

In this case, if the validator fails, then we will redirect users to contact page again, after that, storing validation errors in the session, and also remembering the user's input. If the validator is valid, then we return a message:

```
return 'Your message has been sent';
```

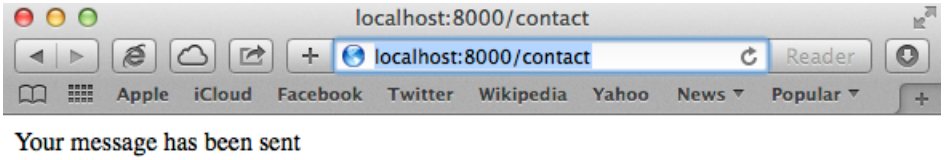
One last thing to do is to display the errors. Open our contact.blade.php template file, find:

```
<p>Please contact us by sending a message using the form below:</p>
```

Then add this line below the paragraph to display errors above our form:

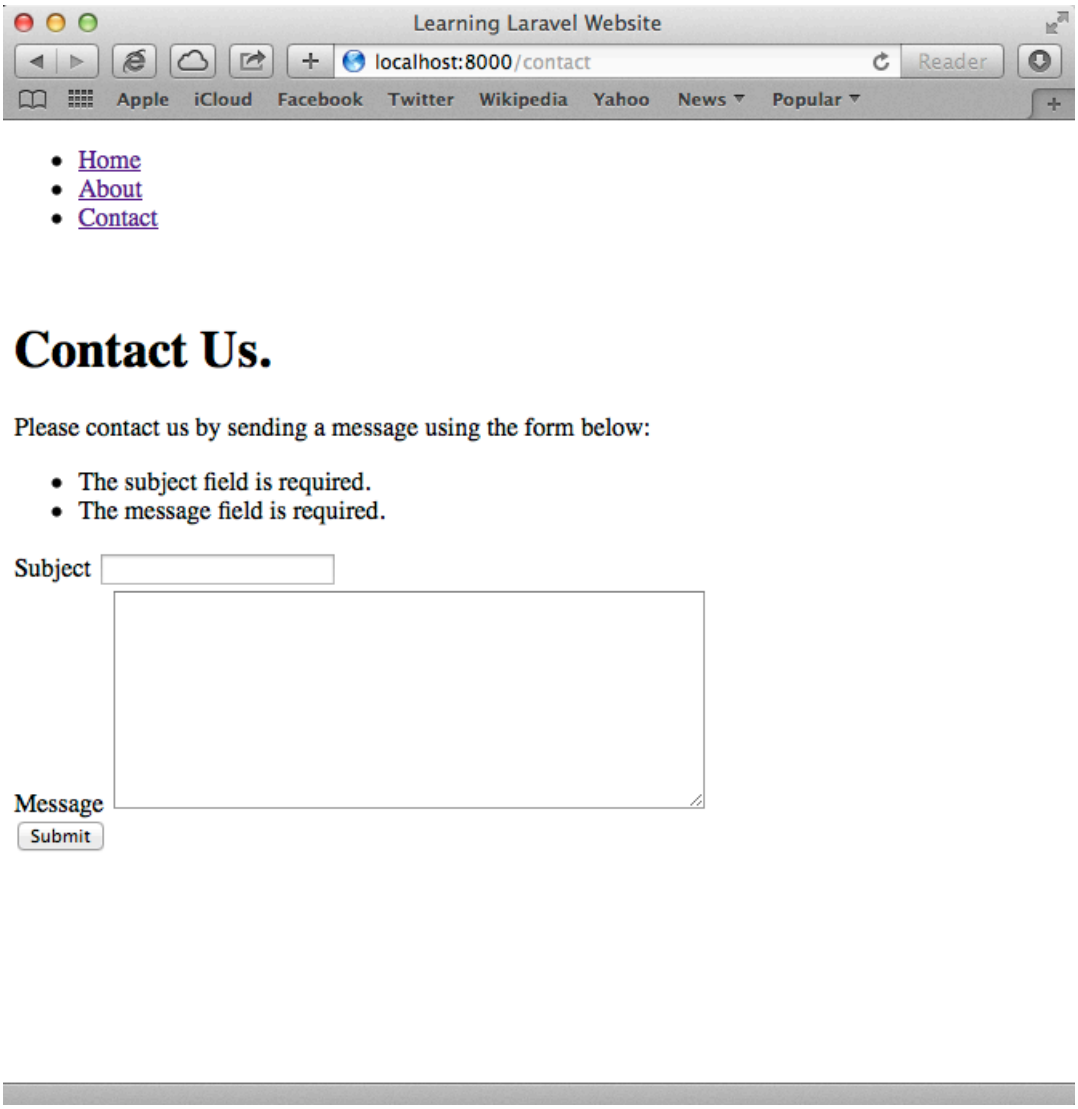
```
{{ HTML::ul($errors->all(), array('class'=>'errors'))}}
```

If you're doing correctly, when you click submit button, you will get a successful message:



Message has been sent

Otherwise, try to remove subject field's content and message field's content, make them blank. You would see the errors:



Errors display on our contact page

Congratulations! But hold on, we can't send emails yet. We will make the form send a real email in the next section!

Sending Emails Using Laravel Mail Method

Sending email with Laravel is pretty simple. You can use a traditional plain PHP method to send emails. However, it's messy and hard to maintain. Laravel has provided an easier way to send emails. Let's do it!

Configure Settings For Sending Emails

First, we need to go to **app/config/mail.php** to configure the settings for sending emails. Laravel provides three different email drivers: smtp (default), PHP mail function and sendmail driver. We will use mail driver, because it's an easiest way to set up. If you don't like to use mail driver, you can use Gmail and other services, too. Find:

```
'driver' => 'smtp',
```

Laravel uses smtp as a default driver, we need to change it to PHP mail funciton:

```
'driver' => 'mail',
```

After that, we need to set the details for our email address and the name where the email will appear to be sent from, find:

```
'from' => array('address' => null, 'name' => null),
```

Change to:

```
        'from' => array('address' => 'support@learninglaravel.net', 'name' => 'Learning \
Laravel'),
```

As you see, we change the address to the Learning Laravel email, and we change the name as well.

Save the file and... done! We can now send emails using PHP built in mail driver!

Sending Email With Gmails

If you have a gmail account, you can use it to send email as well. It's very easy to configure. Open **app/config/mail.php**. Make sure you're using **smtp** driver:

```
'driver' => 'smtp'
```

Lovely, your host should be **smtp.gmail.com**, port is **587** and encryption is **tls**:

```
'host' => 'smtp.gmail.com',
'port' => 587,
'encryption' => 'tls',
```

Don't forget to change:

```
'from' => array('address' => null, 'name' => null),
```

to your email address and your name:

```
'from' => array('address' => 'support@learninglaravel.net', 'name' => 'Learning L\
aravel'),
```

The last step is filling your Gmail username and password:

```
'username' => 'yourname',
'password' => 'yourpassword',
```

Done! I love to use mail driver, but sometimes it's not working. Luckily, Laravel has provided us a very easy way to send emails using Gmail and any smtp service providers. Awesome!

Create HTML Template For Our Emails

Second, we need to create data for our emails. We should use HTML template for our emails. Laravel has created a folder for us. You can view the example of a HTML email template by going to **app/views/emails/auth/reminder.blade.php**. It looks like this:

```
<!DOCTYPE html>
<html lang="en-US">
<head>
        <meta charset="utf-8">
</head>
<body>
        <h2>Password Reset</h2>

        <div>
                To reset your password, complete this form: {{ URL::to('password/reset', array(\
$token)) }}.
        </div>
</body>
```

It's a password reset HTML email template, provided by default by Laravel. We will be sending contact emails, thus we will create a new template. We call it **contactemail.blade.php**, and place it in **app/views/emails** folder. Here is the content:

```
<!DOCTYPE html>
<html lang="en-US">
        <head>
                <meta charset="utf-8">
        </head>
        <body>
                <h2>Contact Form</h2>

                <div>
                        Someone just contacted us:
                </div>
                <div> Subject: {{ $subject }} </div>
                <div> Message: {{ $emailmessage }} </div>
        </body>
</html>
```

As you see, we create a basic HTML template, and then we use **{{ $subject }}** and **{{ $emailmessage }}** to output our email subject and email message.

Sending Our First Emails

When users submit the form, Laravel will check the form input, if it passes validation, the above template will be sent out to our email address. It's time to go back to our **app/routes.php** file and use **Mail::send** function to send emails:

```
if($validator->fails()) {
        return Redirect::to('contact')->withErrors($validator)->withInput();
}

$emailcontent = array (
        'subject' => $data['subject'],
        'emailmessage' => $data['message']
);

Mail::send('emails.contactemail', $emailcontent, function($message)
{
        $message->to('support@learninglaravel.net','Learning Laravel Support')
        ->subject('Contact via Our Contact Form');
});

        return 'Your message has been sent';

});
```

Let's seperate the codes to understand easier:

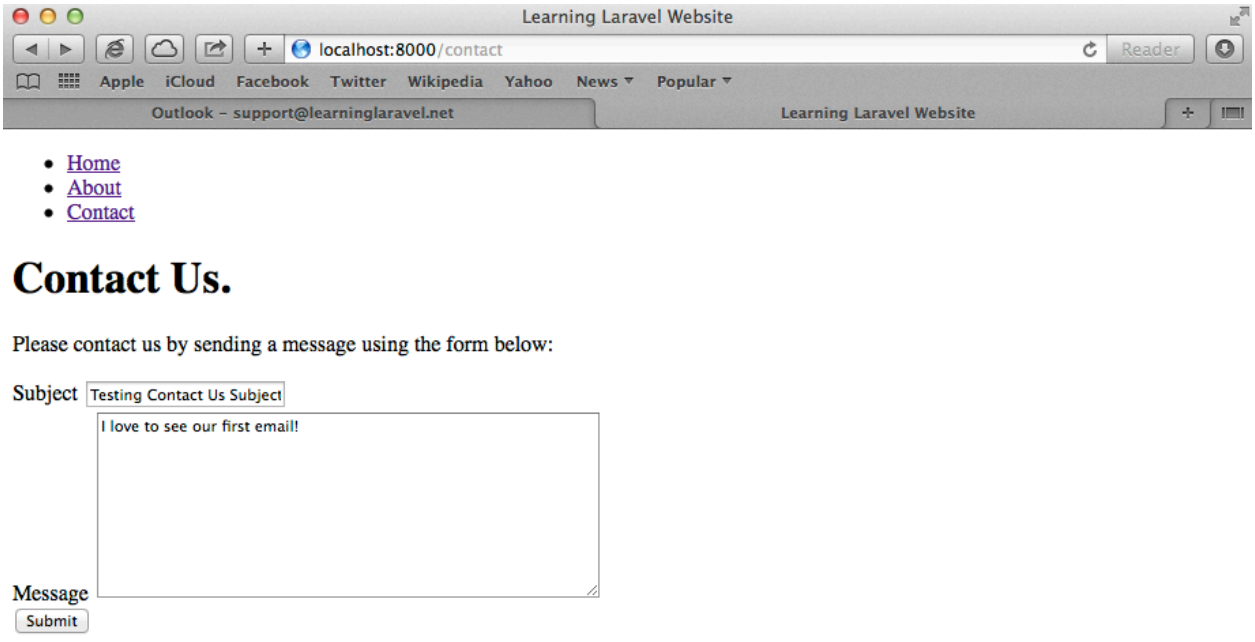
```
$emailcontent = array (
        'subject' => $data['subject'],
        'emailmessage' => $data['message']
);
```

We create an array called $emailcontent, this array contains subject and emailmessage from the form.

```
Mail::send('emails.contactemail', $emailcontent, function($message)
{
        $message->to('support@learninglaravel.net','Learning Laravel Support')
        ->subject('Contact via Our Contact Form');
});
```

After that, we use **Mail::send** Laravel method to send emails. The first parameter is our **contactemail.blade.php** HTML template and the **emails folder** where the template file is located. The second parameter is our email content. (the array we've just created) The third parameter is a function that will send our email. We need to fill in **where** the email will be sent to, the **name** of the receiver and the **subject** of our email!

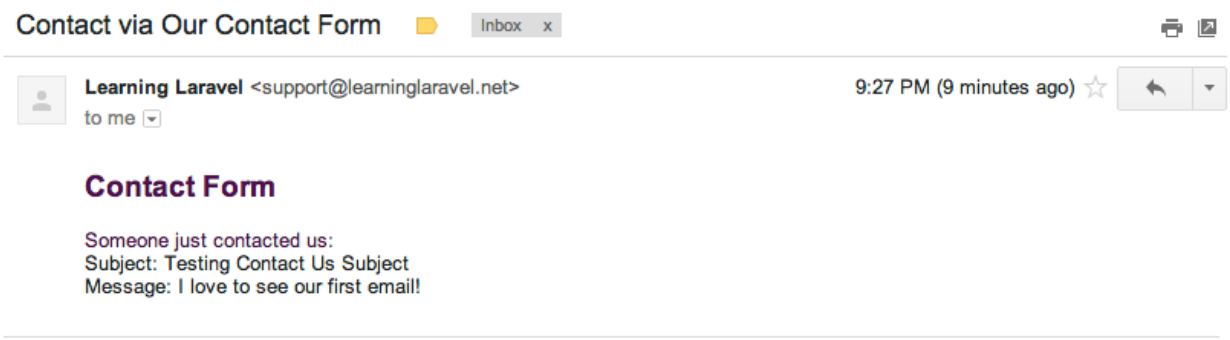
If everything is ok, go to our contact page, fill in the contact form, like this:



Fill the contact form

After that, hit the Submit button! We will get the message "**Your message has been sent**"!

Congrats! Now, let's check our email inbox. We should see:



Our email content

The email should be in our inbox. If you don't receive emails, your email service providers may block and mark emails from localhost domain as spam. Sometimes, they may ban our IP addresses as well, thus you can't recieve emails. Don't worry, you can create another email address at different email service providers and try again, or test it on a real hosting/server, or use different driver (such as SMTP). There are many ways to do.

I use Gmail and I can receive the emails, but I can't get the emails from my localhost when using Outlook.

Cool! At the moment, we have a working website. But it's ugly! Right? I'm a designer as well so I hate something ugly! We're going to use Twitter Bootstrap 3 in the next section to make our website look better!

Integrating Twitter Bootstrap

What is Twitter Bootstrap? Well, it's the most popular front-end framework today. It's a collection of CSS, Jquery and HTML conventions. You can use Twitter Bootstrap to quickly create stylish typography, forms, buttons, tables, grids and many more. The newest Twitter Bootstrap version is v3.0.3 (at this time). You can go to its homepage to find out more information:

Twitter Bootstrap Homepage[23]

Integrating Twitter Bootstrap into Laravel is very easy. You can do it by using just a single line of code. Interesting? Let's do it.

We open our **layout.blade.php** and place this code inside (below the title):

[23] http://getbootstrap.com

```
<title>Learning Laravel Website </title>
<link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootst\
rap.min.css">
```

Done! It's very easy, isn't it?

Integrating Bootstrap differently?

There are many ways to integrate Twitter Bootstrap into Laravel. What I just show you is Bootstrap CDN method. You can use other methods if you like, just visit Bootstrap documentation to learn more.

We've just integrated Bootstrap CSS into our web application. Now we will integrate Bootstrap JavaScripts plugins into our website. The Bootstrap JavaScript plugins require jQuery to work, thus we will include jQuery as well:

```
<title>Learning Laravel Website </title>
<link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootst\
rap.min.css">
<script src="https://code.jquery.com/jquery.js"></script>
<script src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js"></scr\
ipt>
```

Congrats! Now you have fully integrated Twitter Bootstrap into our website.

A little tip about JavaScript

If you want your website loads a bit faster, you can put those JS scripts at the bottom, before the **</body>** tag.

Wondering where I get the code? It's in the Bootstrap Getting Started page:

Twitter Bootstrap Getting Started[24]

Modify Navigation Bar Using Twitter Bootstrap

Nothing has changed yet. We will need to apply some html and css rules to make our website look better. First, we will modify our menu. Open **layout.blade.php** and find:

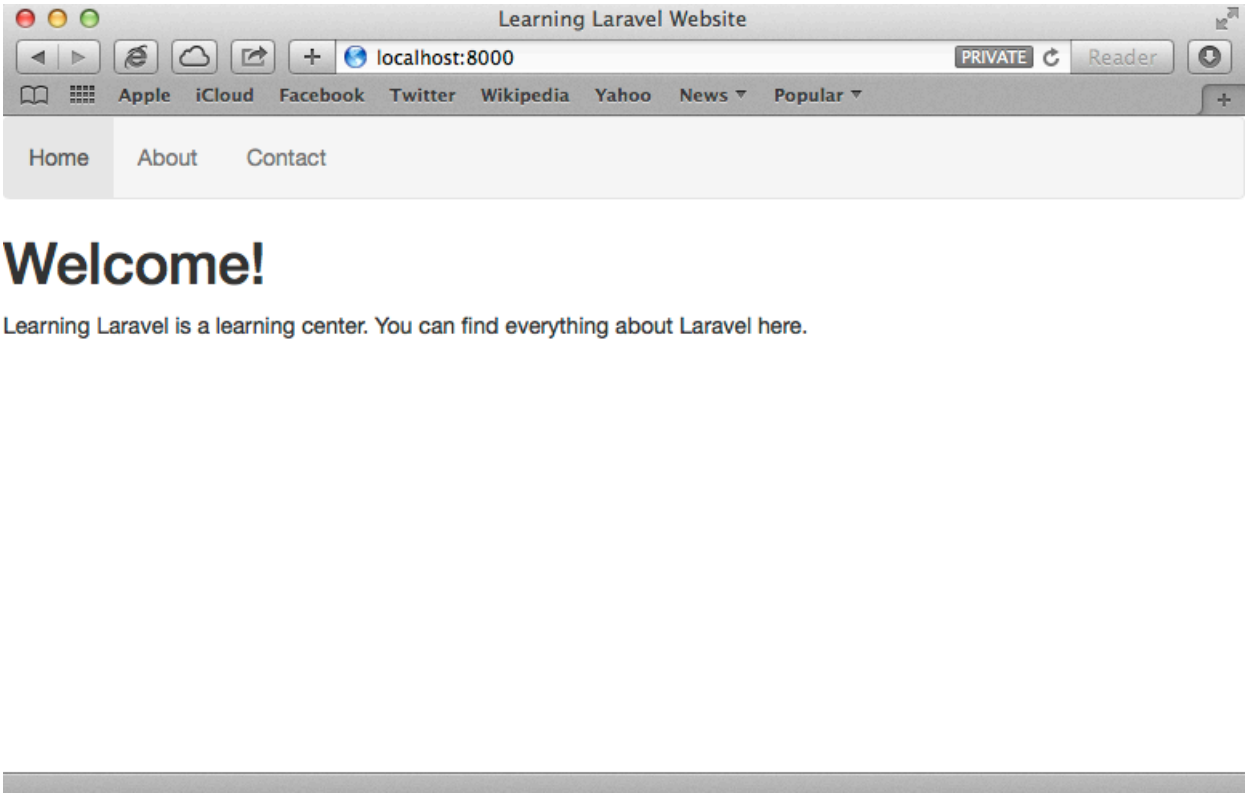
[24] http://getbootstrap.com/getting-started

```
<ul>
    <li><a href="./">Home</a></li>
    <li><a href="./about">About</a></li>
    <li><a href="./contact">Contact</a></li>
</ul>
```

Replace with:

```
<nav class="navbar navbar-default" role="navigation">
  <div class="navbar-header">
     <ul class="nav navbar-nav">
        <li class="active"><a href="./">Home</a></li>
        <li><a href="./about">About</a></li>
        <li><a href="./contact">Contact</a></li>
     </ul>
  </div>
</nav>
```

Save the file, refresh our website and you should see:



Our new menu - a navigation bar

Basically, we just add some css classes into our HTML divs, then we have a nice flat navigation bar!

You can take a look at Bootstrap Components to see some examples:

Twitter Bootstrap Components[25]

Good, if you want to center our page to make it look better. Add these codes:

```
        <div class="container col-md-8 col-md-offset-2">
        <nav class="navbar navbar-default" role="navigation">
                <div class="navbar-header">
                        <ul class="nav navbar-nav">
                              <li class="active"><a href="./">Home</a></li>
                      <li><a href="./about">About</a></li>
                      <li><a href="./contact">Contact</a></li>
                      </ul>
                </div>
        </nav>
        @yield('content')
        </div>
```

We use a div containter to wrap our website content. After that, we use Bootstrap's own offset classes to center the div, so it requires no change in markup and no extra CSS.

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

The key is to set an offset equal to half of the remaining size of the row. For example, a column of size 8 would be centered by adding an offset of 2, that's (12-8)/2.

If you don't understand, that's fine, you need some basic knowledges about CSS. Don't worry and let's move one. You just need to remember that:

If you have a column size of 10, the offset will be 1

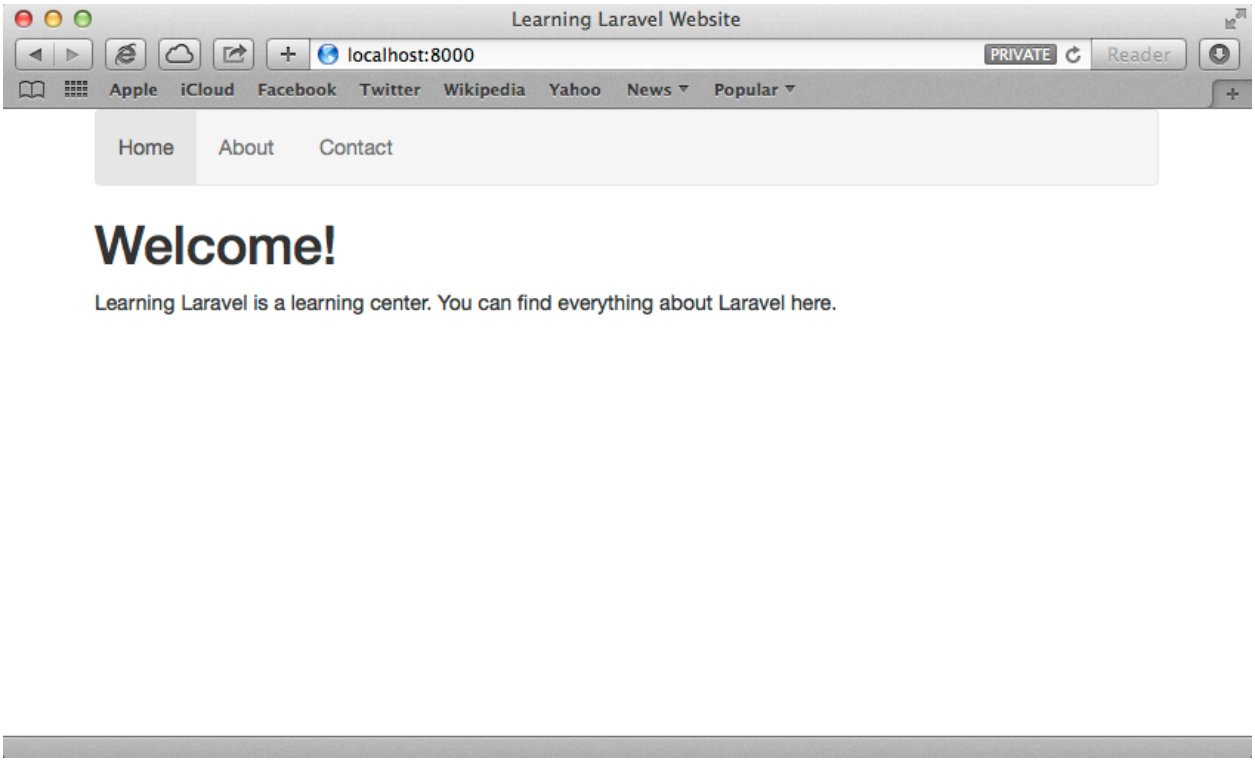
If you have a column size of 8, the offset will be 2

If you have a column size of 6, the offset will be 3

If you have a column size of 4, the offset will be 4

If you have a column size of 2, the offset will be 5

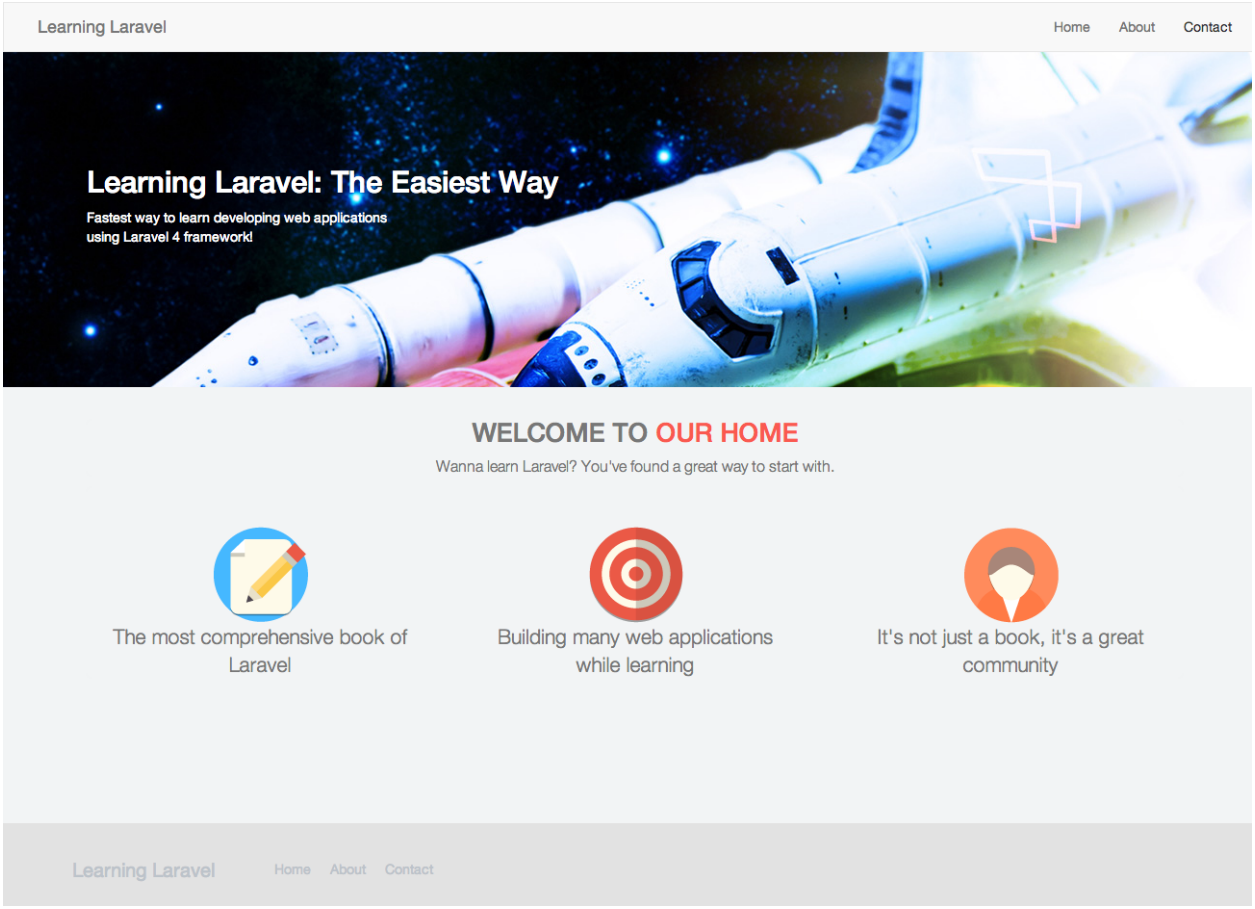
[25] http://getbootstrap.com/components/#navbar



Our new home page!

Change Our Layout Using Bootstrap

At this time, we have known how to integrate Twitter Bootstrap into our Laravel application and how to modify our menu (navigation bar) by using Twitter Bootstrap. We will take one more step further! In this section, we will change the home page again and make it fully responsive. We can use it for our personal home page. Let's take a look at what we will build:



A fully responsive home page!

Is that cool? Let's start!

Don't know CSS + HTML5?

You will need to have basic knowledges about CSS and HTML5 to understand this section easier. But don't worry, if you don't understand, just copy + paste and follow along.

First, we will create a **css folder** in **public folder**. After that, we create a new custom css file in the **public/css folder**, we called it **style.css**. We use this file to apply css rules to our page. Finally, we change the background and the font size of our page by putting these codes into our style.css file:

```
1  body {
2          font-size: 14px;
3          background: #F2F4F5;
4  }
```

As you see, I change the font size to 14px and the background color of our page to #F2F4F5.

We won't see any different yet, we need to link the css file to our page. Laravel has a useful function called asset. We can use it to access our public folder easily. Open and modify our **app/views/layout.blade.php** file. Here is our header:

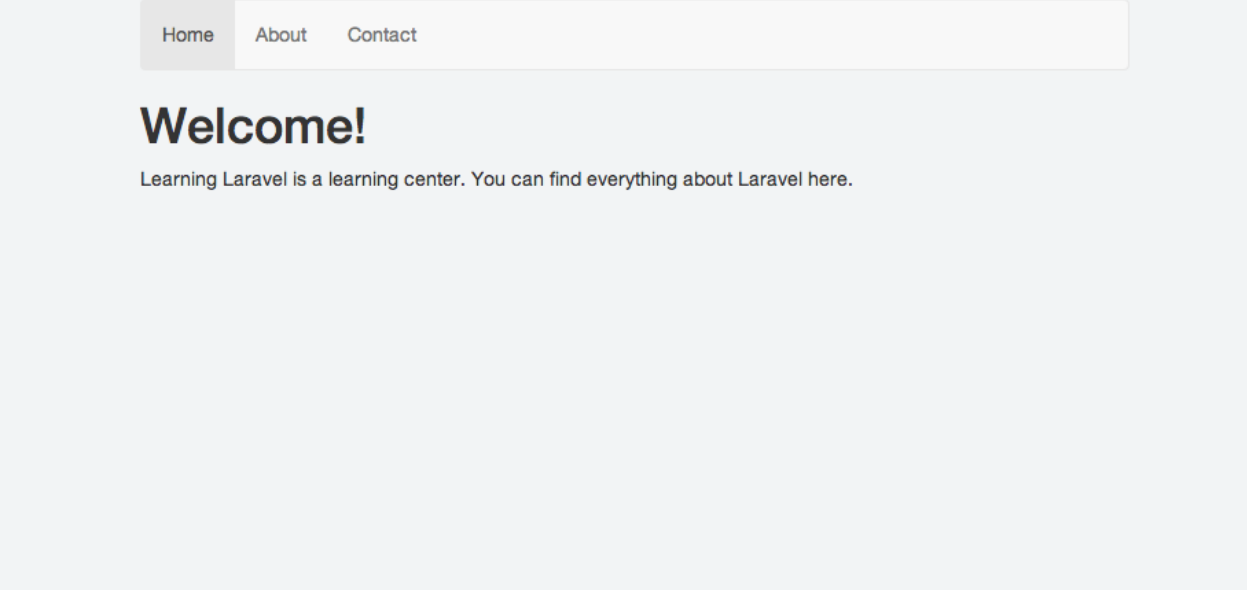
```
<head>
     <meta charset="UTF-8">
    <title>Learning Laravel Website </title>
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bo\
otstrap.min.css">
    <script src="https://code.jquery.com/jquery.js"></script>
    <script src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js"><\
/script>
    <link rel="stylesheet" href="{{ asset('css/style.css') }}">
</head>
```

Take a look at this line:

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

We use **{{ asset('css/style.css') }}** to link the style.css to our page. Pretty simple, right?

Go ahead and refresh our page, you should see the background color and the font size have changed.



Background color has changed!

We're going to change our navigation bar again and make it "mobile friendly". The easiest way to do is going to:

Twitter Bootstrap Components[26]

Copy the code of the Default navbar:

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

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

[26] http://getbootstrap.com/components/#navbar-default

```
    <ul class="nav navbar-nav navbar-right">
      <li><a href="#">Link</a></li>
      <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown">Dropdown <b cl\
ass="caret"></b></a>
        <ul class="dropdown-menu">
          <li><a href="#">Action</a></li>
          <li><a href="#">Another action</a></li>
          <li><a href="#">Something else here</a></li>
          <li class="divider"></li>
          <li><a href="#">Separated link</a></li>
        </ul>
      </li>
    </ul>
  </div><!-- /.navbar-collapse -->
</nav>
```

After that, we paste the codes into our **layout.blade.php** file, replacing the current navigation bar. We also delete some parts that we don't use (such as the dropdown, search form, etc.), and adding some buttons (Home, About, Contact) into our navigation bar. We should have navigation codes like this:

```
<nav class="navbar navbar-default" role="navigation">
  <!-- Brand and toggle get grouped for better mobile display -->
  <div class="navbar-header">
    <button type="button" class="navbar-toggle" data-toggle="collapse" data-targe\
t="#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>

  <!-- Collect the nav links, forms, and other content for toggling -->
  <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">

    <ul class="nav navbar-nav navbar-right">
      <li><a href="/">Home</a></li>
      <li><a href="/about">About</a></li>
      <li><a href="/contact">Contact</a></li>
```

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

Good job! One more step, we replace:

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

with a header div to define our header section:

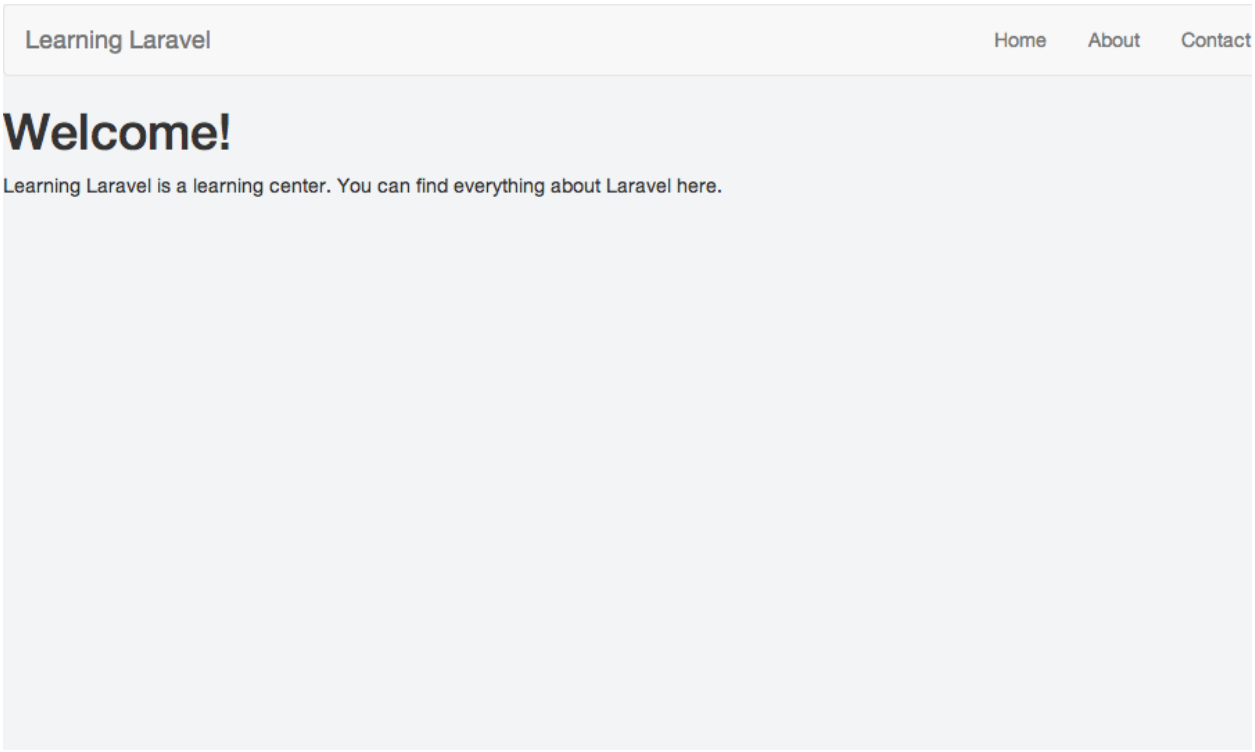
```
<header>
...
</header>
```

If you're doing it correctly, you should have a **layout.blade.php** look like this:

```
<!doctype html>
<html lang="en">
<head>
     <meta charset="UTF-8">
    <title>Learning Laravel Website </title>
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bo\
otstrap.min.css">
    <script src="https://code.jquery.com/jquery.js"></script>
    <script src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js"><\
/script>
    <link rel="stylesheet" href="{{ asset('css/style.css') }}">
</head>
<body>
<header>
<nav class="navbar navbar-default" role="navigation">
  <!-- Brand and toggle get grouped for better mobile display -->
  <div class="navbar-header">
    <button type="button" class="navbar-toggle" data-toggle="collapse" data-targe\
t="#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>
```

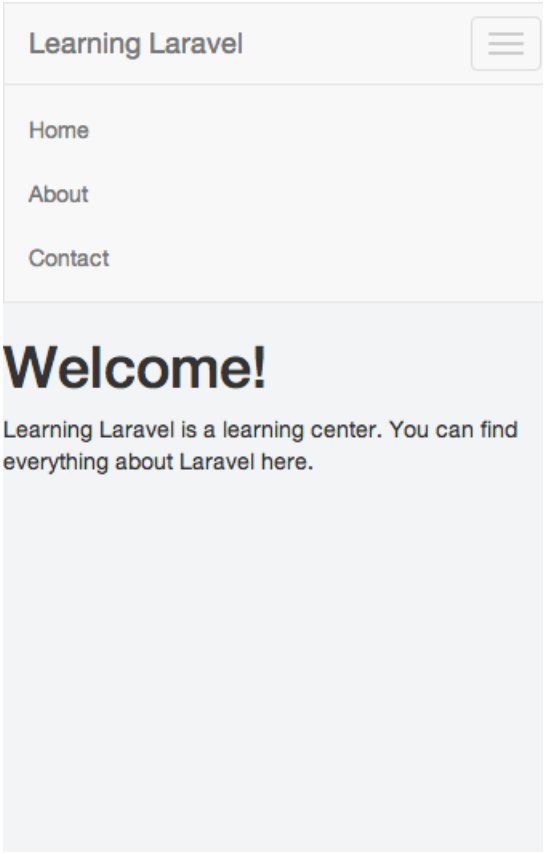
```
24      </button>
25      <a class="navbar-brand" href="/">Learning Laravel</a>
26    </div>
27
28    <!-- Collect the nav links, forms, and other content for toggling -->
29    <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
30
31      <ul class="nav navbar-nav navbar-right">
32        <li><a href="/">Home</a></li>
33        <li><a href="/about">About</a></li>
34        <li><a href="/contact">Contact</a></li>
35      </ul>
36    </div><!-- /.navbar-collapse -->
37  </nav>
38  </header>
39
40  @yield('content')
41
42
43  </body>
44  </html>
```

Save the file and refresh our page to see a new fully responsive navigation bar:



Our new navigation bar!

Try to resize our web browser, you should see a button. This button will be displayed when we access our page using mobile phone or small screen devices. In Twitter Bootstrap 3, it's called **navbar toggle**.



Our small lovely navbar toggle

Change Our Home Page

We have a good layout now, this layout (including the navigation bar) will be used for all other pages. It's time to modify our home page and make it look better.

I have prepared some images for you. You can download them here:

Download Chapter 1-1 file[27]

Download the **zip file** to your computer, unzip it. You will have all the images. Create a folder called **img** in **public** folder. After that, put all the images into **public/img folder**.

Open **home.blade.php** and delete these:

```
1     <h1>Welcome!</h1>
2     <p>Learning Laravel is a learning center. You can find everything about Larav\
3 el here.</p>
```

[27] http://learninglaravel.net/DL/chapter1-1.zip

We will create a section for our main page banner, and the class of the section is called header. We also create a div and class=”background” for it:

```
      <section class="header">
            <div class="background"> </div>
        </section>
```

It’s time to apply some css rules in order to display the banner. Open our **style.css** file, insert:

```
.background {
  position: absolute;
  left: 0;
  top: 0;
  right: 0;
  bottom: 0;
  background: 50% 50% no-repeat;
  -webkit-background-size: cover;
  -moz-background-size: cover;
  -o-background-size: cover;
  background-size: cover;
}

.header .background {
  background-image: url("../img/bg.jpg");
  height: 400px;
  z-index: -1;
}
```

Good. We just use some basic css rules to display the banner. The image that I use for the banner is 400 px height. If you want to replace it with other image, you can change the height. Please note that, the width of the image must be at least 1100px or higher. Why? Because many people are using high resolution display devices nowadays (usually more than 1100px width).

Now refresh our web browser and we should see the banner:

Our website banner

Next step, we will place some texts on top of our website banner:

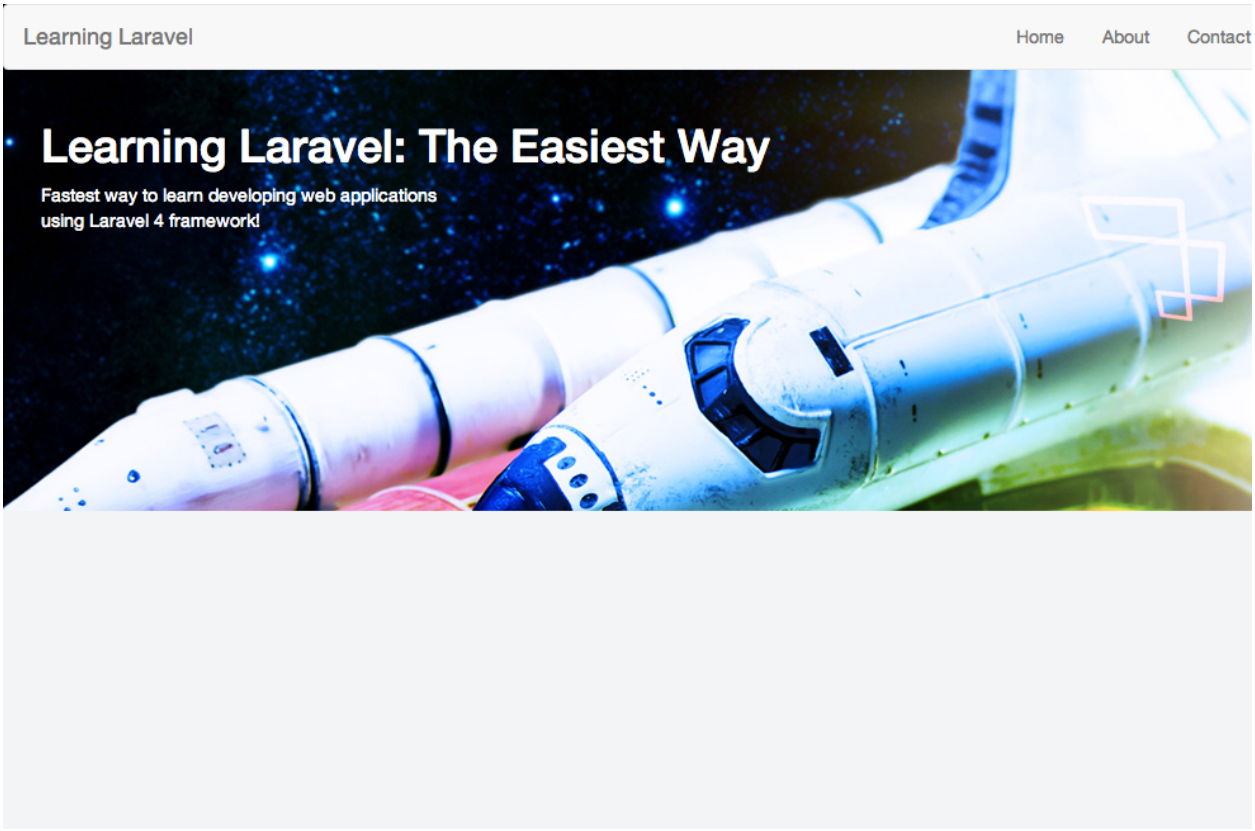
```
<!--app/views/home.blade.php-->

        <section class="header section-padding">
                <div class="background"> </div>
                <div class="container">
                        <div class="header-text">
                                <h1>Learning Laravel: The Easiest Way</h1>
                                <p>
                                Fastest way to learn developing web applications
                                <br /> using Laravel 4 framework!
                                </p>
                        </div>
                </div>
        </section>
```

We put all the texts into divs with class=”container” and class=”header-text”. The title should be bigger, thus we use h1 tag for it.

We also need to change the text color to make it display clearly:

```
<!--public/css/style.css-->

.header-text {
    color: #ffffff;
}
```

Refresh our web browser once again, we should have a good looking website with a nice banner:



Our banner with text

It will be cool if the navbar still stay on the page as we scroll. We will change the navbar to fixed position, and because of that, its width should be 100% as well:

```
<!--public/css/style.css-->

.navbar {
        border-radius: 0;
        position: fixed;
        width: 100%;
        padding: 0 20px;
        z-index: 100;
}
```

We also give our navigation bar (navbar) a bit of padding. Now, back to our web browser and try to resize the windows smaller. When we scroll down, the navbar should stay on top. It looks like it falls down when we scroll.

Next step, we will create another container and add our welcome text there:

```
<!--app/views/home.blade.php-->

<section class="header section-padding">
    <div class="background"> </div>
        <div class="container">
            <div class="header-text">
                <h1>Learning Laravel: The Easiest Way</h1>
                <p>
                                Fastest way to learn developing web applications <br /> using Lara
work!
                </p>
             </div>
         </div>
     </div>
 </section>

 <div class="container">
        <section class="section-padding">
                <div class="jumbotron text-center">
                        <h1><span class="grey">WELCOME TO</span> OUR HOME</h1>
                        <p>
                        Wanna learn Laravel? You've found a great way to start with.
                        </p>
                </div>
        </section>
</div>
```

You may notice that there are some new CSS classes: section-padding, jumbotron, text-center and grey.

```
        <section class="section-padding">
                <div class="jumbotron text-center">
                        <h1><span class="grey">WELCOME TO</span> OUR HOME</h1>
```

We use section-padding to give some padding between each section.

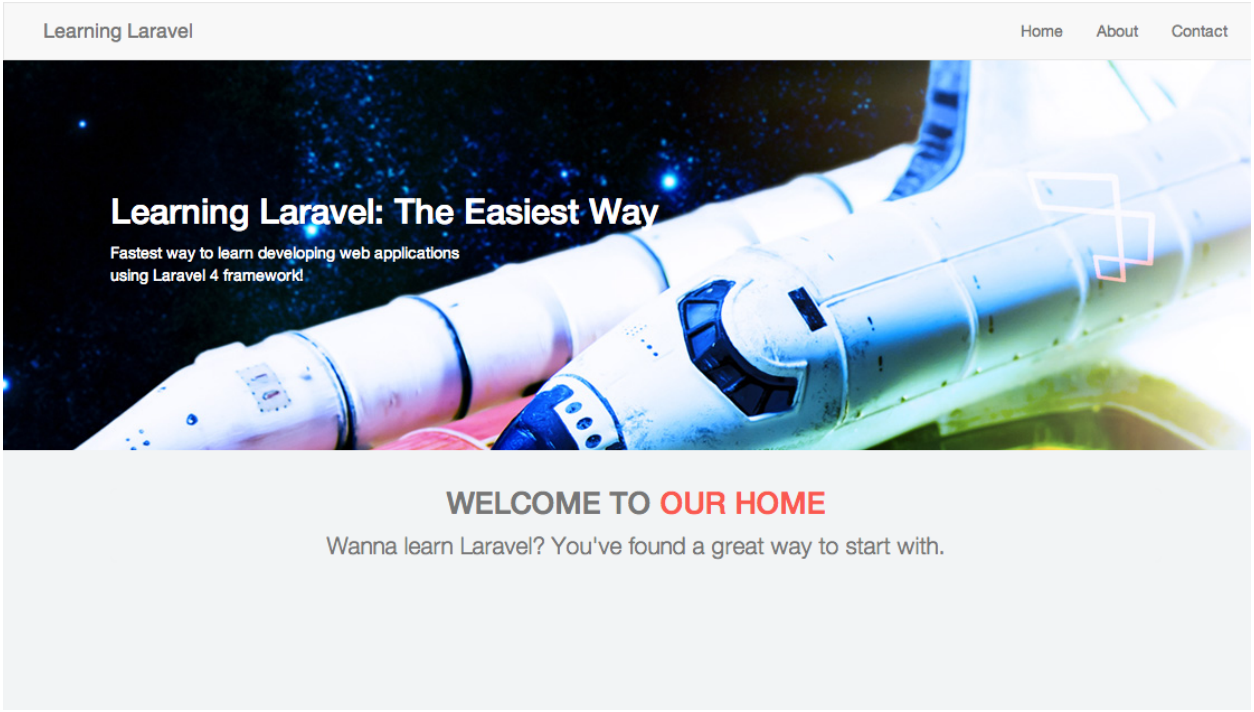
Jumbotron is a large callout. It's a new class in Twitter Bootstrap 3. If you want to know more about Jumbotron, take a look at Twitter Bootstrap 3 documentation.

We use text-center class to center our texts and grey class to change the text color to grey.

Now, adding some CSS:

```
<!--public/css/style.css-->

.jumbotron {
        padding:0 !important;
        margin:0;
        color: #777777;
        background: #F2F4F5;
}

.jumbotron h1 {
        color: #fa5950;
        font-size: 27px;
}

.section-padding {
        padding-top: 150px;
}

.grey {
        color:#777777;
}
```

Great! Refresh our page one more time and you should see nice welcome texts:



Welcome texts!

It's time to put some showcase boxes into our page. Insert these codes below our welcome texts:

```
<div class="jumbotron text-center">
        <div class="row">
                <div class="showcase-box col-md-4">
                        <div class="showcase-item">
                                <img src="{{ asset('img/icon1.png') }}" />
                                        <p>
                                        The most comprehensive book of Laravel
                                        </p>
                        </div>
                </div>
                <div class="showcase-box col-md-4">
                        <div class="showcase-item">
                                <img src="{{ asset('img/icon2.png') }}" />
                                        <p>
                                        Building many web applications while learning
                                        </p>
                        </div>
                </div>
                <div class="showcase-box col-md-4">
                        <div class="showcase-item">
```

```
                                <img src="{{ asset('img/icon3.png') }}" />
                                        <p>
                                        It's not just a book, it's a great community
                                        </p>
                        </div>
                </div>
        </div>
</div>
```

We use jumbotron class again. We also create showcase-box and showcase item class for our showcase boxes. The **col-md-** is a grid class in Bootstrap Grid system. **col-md-4** means that the box has 4 columns.

It should look like this:

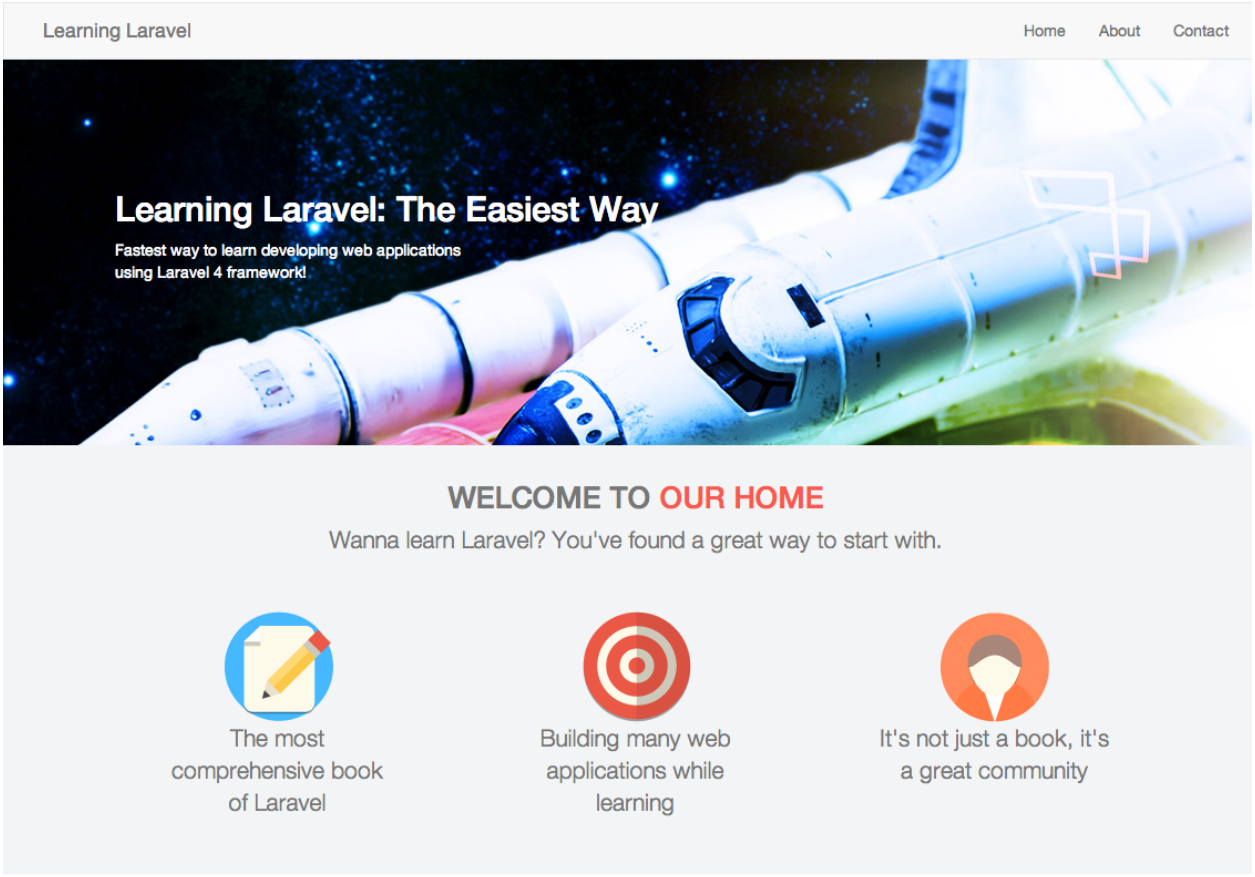
```
<!--app/views/home.blade.php-->

<div class="container">
        <section class="section-padding">
                <div class="jumbotron text-center">
                        <h1><span class="grey">WELCOME TO</span> OUR HOME</h1>
                        <p>
                        Wanna learn Laravel? You've found a great way to start with.
                        </p>
                </div>

        <div class="jumbotron text-center">
                <div class="row">
                        <div class="showcase-box col-md-4">
                                <div class="showcase-item">
                                        <img src="{{ asset('img/icon1.png') }}" />
                                        <p>
                                        The most comprehensive book of Laravel
                                        </p>
                                </div>
                        </div>
                        <div class="showcase-box col-md-4">
                                <div class="showcase-item">
                                        <img src="{{ asset('img/icon2.png') }}" />
                                        <p>
                                        Building many web applications while learning
                                        </p>
                                </div>
```

```
                        </div>
                        <div class="showcase-box col-md-4">
                                <div class="showcase-item">
                                        <img src="{{ asset('img/icon3.png') }}" />
                                        <p>
                                        It's not just a book, it's a great community
                                        </p>
                                </div>
                        </div>
                </div>
        </section>
</div>
```

We also need to add CSS to the showcase boxes:

```
<!--public/css/style.css-->

.showcase-box {
        height: 200px;
        padding: 2px;
}

.showcase-item {
        color: #777777;
        padding: 40px;
        height: 100%;
        width: 100%;
}
```

Nothing special here, we just change the text color, add some padding, define height and width. That's all. Refresh our page again:



Our page. Almost done!

One more step! We're going to build a footer for our page. I call it bottom-menu. Open **layout.blade.php** and add:

```
<!--app/views/layout.blade.php-->

<div class="bottom-menu">
  <div class="container">
    <div class="row">
      <div class="col-md-2 navbar-brand">
         <a href="/">Learning Laravel</a>
      </div>

      <div class="col-md-10">
        <ul class="bottom-links">
          <li><a href="/">Home</a></li>
           <li><a href="/about">About</a></li>
           <li><a href="/contact">Contact</a></li>
        </ul>
```

```
      </div>

    </div>
  </div>
</div>
```

Our **layout.blade.php** should look like this:

```
    <!doctype html>
    <html lang="en">
    <head>
         <meta charset="UTF-8">
        <title>Learning Laravel Website </title>
        <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.3/cs\
s/bootstrap.min.css">
        <script src="https://code.jquery.com/jquery.js"></script>
        <script src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.j\
s"></script>
        <link rel="stylesheet" href="{{ asset('css/style.css') }}">
    </head>
    <body>
        <header>
        <nav class="navbar navbar-default" role="navigation">
        <!-- Brand and toggle get grouped for better mobile display -->
        <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-t\
arget="#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>

        <!-- Collect the nav links, forms, and other content for toggling -->
        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">

            <ul class="nav navbar-nav navbar-right">
              <li><a href="/">Home</a></li>
              <li><a href="/about">About</a></li>
              <li><a href="/contact">Contact</a></li>
```

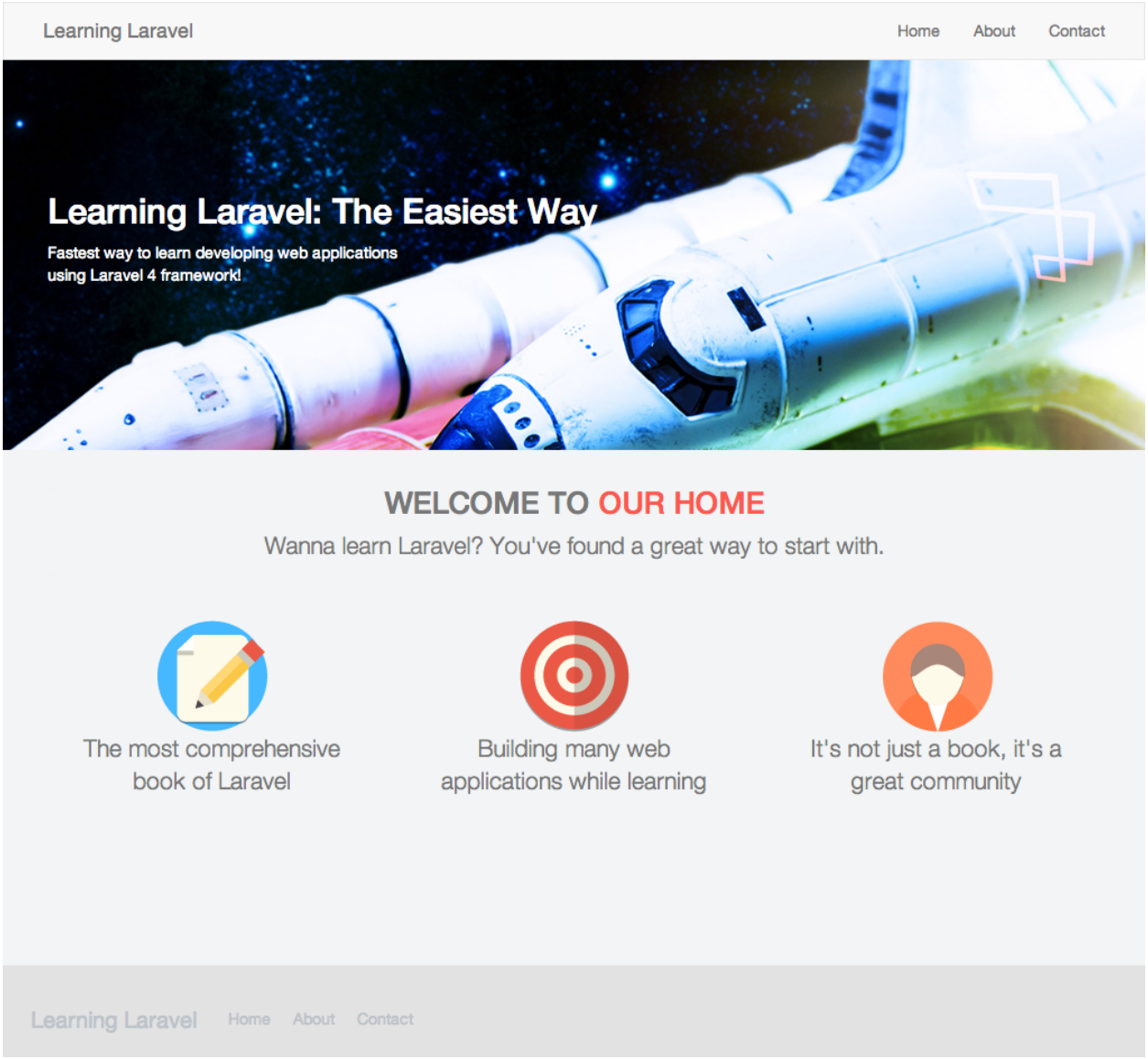
```
            </ul>
          </div><!-- /.navbar-collapse -->
        </nav>
        </header>

    @yield('content')

    <div class="bottom-menu">
      <div class="container">
        <div class="row">
          <div class="col-md-2 navbar-brand">
             <a href="/">Learning Laravel</a>
          </div>

          <div class="col-md-10">
            <ul class="bottom-links">
              <li><a href="/">Home</a></li>
               <li><a href="/about">About</a></li>
               <li><a href="/contact">Contact</a></li>
            </ul>
          </div>

        </div>
      </div>
    </div>

    </body>
    </html>
```

It's time for applying CSS to our bottom-menu:

```
.bottom-menu {
  background: #e2e2e2;
  color: #bcc3ca;
  padding: 39px 0 42px;
  margin: 150px 0 0 0;
}
.bottom-menu .navbar-brand {
  font-size: 20px;
  padding: 0;
}
.bottom-menu .title {
```

```
  font-size: 13px;
  font-weight: 700;
  margin-top: 0;
}
.bottom-menu a {
  color: inherit;
}
.active .bottom-menu a,
.bottom-menu a:hover,
.bottom-menu a:focus {
  color: #e47054;
  text-decoration: none;
}

.bottom-menu .bottom-links {
  font-size: 14px;
  line-height: 1.286;
}

.bottom-links{
  margin: 0;
  padding: 0;
  list-style: none;
}
.bottom-links li {
  display: block;
  float: left;
  margin: 0 20px 0 0;
}
```

We now have a great simple bottom menu!



Bottom menu!

We're almost finished creating our first home page. However, our banner text still doesn't display well on smaller screen devices (such as tablets, iPhone). The text's color is white; thus it's hard to see. To fix that, we can put the following rule to our CSS file:

```
@media (max-width: 767px) {
        .jumbotron h1 {
                font-size:22px;
        }
        .header-text {
        color: #fa5950;
        }
```

Do you notice the **@media** line above? It called **Media Queries** in CSS3. We often use Media Queries to write CSS specifically for certain situations. For example, detecting that the user uses a smaller screen devices and giving them a specific layout. As you can see, if the user's screen is smaller than 768px, we apply different CSS rules to our website. We change font-size of the jumbotron h1 to 22px, and the header-text's color will be changed to #fa5950.

We also apply some CSS rules to our bottom-menu as well to make it looks better on smaller devices:

```
  .bottom-menu .navbar-brand,
  .bottom-menu .bottom-links {
    margin-bottom: 10px;
  }
  .bottom-menu .navbar-brand {
    padding: 15px;
    float: none;
  }
  .bottom-menu .bottom-links li {
    float: none;
    margin-bottom: 2px;
  }
```

Our CSS rules look like this:

```
@media (max-width: 767px) {
        .jumbotron h1 {
                font-size:22px;
        }
        .header-text {
        color: #fa5950;
        }

  .bottom-menu .navbar-brand,
  .bottom-menu .bottom-links {
    margin-bottom: 10px;
```

```
  }
  .bottom-menu .navbar-brand {
    padding: 15px;
    float: none;
  }
  .bottom-menu .bottom-links li {
    float: none;
    margin-bottom: 2px;
  }
}
```

Congrats! We now have a perfect fully responsive home page!

Try to resize our page, we should see that our banner text changes its color!

It's working well on small devices

When our home page is finished, creating other pages is pretty easy. Try to change about us and contact page yourself if you like. It's a good way to practice.

I have changed about us page and contact page, and uploaded the app here:

Download Chapter 1-2 file - our first website[28]

You can download it and take a look to check whether you're doing right or wrong. You can also modify and enhance it to have a personal fully responsive website.

That's it for Chapter 1. The fun part is just about to begin!

Chapter 1 Summary

Good job! We just finish Chapter 1! We now have a fully responsive website, built using Laravel!

During this chapter, you have learned many things:

1. You've known how to install Laravel and create our project using Composer or the new Laravel Installer.
2. You've learned how to use Command Line, Php Artisan.
3. You've looked at a Laravel application, known its structure and how it works.
4. You've learned about Laravel routes. Now you can use **routes** to redirect users to a place that you want.
5. You've learned basic Blade Views. You can be able to create Blade templates for your web applications.
6. Configuring Laravel app to send emails is "a piece of cake" at the moment, right?
7. Creating forms in Blade using Laravel method is pretty easy.
8. You've known how to add some basic validations.
9. You've also learned how to integrate Twitter Bootstrap into our Laravel Application and how to use Twitter Bootstrap as well.
10. Finally, you've created a nice fully responsive website with Twitter Bootstrap and Laravel!

Wondering why we've spent a lot of time and messed around with CSS to create a fully responsive home page? It's because we need a cool template for our next web applications. In this era of technology, a good website should run fine and look beautiful as well.

In the next chapter, we will look into Schema Builder, Query Builder, Eloquent ORM and other important Laravel concepts. Afterwards, we will build a dynamic database driven web application!

[28] http://learninglaravel.net/DL/chapter1-2.zip

Chapter 2 - Building A To-do List Application

Welcome to Chapter 2!

We've learned how to configure our web app to send emails and how to use basic Blade templates, now it's time to take one more step and learn how we can store information in database, display the information and manipulate our data.

What is database?

If you're not coming from programming world, I'm happy to explain a bit about database. Basically, a database is just a collection of data. Instead of saving data to our computer as files (such as 1.doc, 2.txt, etc.), we need a way to easily store and access our data. A database is a type of software that is designed to handle lots of data in a way, which we can find, receive and modify efficiently.

More than that, in this Chapter we will learn more about some Laravel special features such as Blade Template, Schema Builder, Eloquent ORM, Controllers, Composer and Artisan.

As usual, we will learn all of those while building a practical application: A To-do List. Let's get started!

Preparing Our Application

In order to build a To-do list application and learn new concepts, we will use our app in Chapter 1 again. I copy it to my desktop and rename the new app to **todo**. My system is Mac, so I will use these commands to serve it:

```
1  cd desktop/todo
2  php artisan serve --port 8080
```

**Note: If your system is Windows and you use XAMPP or WAMP, the steps to do this could be different. However, it's similar to the installation. so I think you should know how to do it by now.*

You may notice that this line is different:

```
php artisan serve --port 8080
```

Basically, **–port 8080** tells that we will start our Laravel development server on port 8080 (default port is 8000).

Therefore, to access our site, we can now go to:

http://localhost:8080[29]

You can use any port that you like.

Laravel Database Configuration

Laravel supports many database platforms and we can choose any of them to develop our applications:

1. MySQL[30]
2. SQLite[31]
3. PostgreSQL[32]
4. SQL Server[33]

In this book, we will use MySQL, which is one of the most popular and free platforms for development. You can also use other database platforms if you like. Laravel is very clever, it takes care of the SQL syntax for you.

To configure our database, let's go to **app/config/**, open **database.php** file and take a look at:

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

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

[29] http://localhost:8080
[30] http://www.mysql.com
[31] http://www.sqlite.org
[32] http://www.postgresql.org/
[33] http://www.microsoft.com/en-us/sqlserver

Laravel uses The PHP Data Objects (PDO), which is a lightweight, consistent interface for accessing databases in PHP. When we execute a Laravel SQL query, rows are returned in a form of a PHP stdClass object. We will access our data by using something like this:

```
echo $user->name;
echo $user->email;
```

If you like to change the format, you can do it easily by just changing the option here. For example:

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

Now, the codes to access our data has been changed to:

```
echo $user['name'];
echo $user['email'];
```

If you would like to learn more about PDO, as well as its PDO fetch modes, you can go to its official documentation:

PDO Official Documentation[34]

Ok, we would like to use MySQL. How to do it? Very easy! We can do that by editing this line:

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

'default' => 'mysql',
```

As you can see, we specify MySQL as our default connection for all database work. Let's scroll down a bit:

[34] http://www.php.net/manual/en/intro.pdo.php

```
'connections' => array(

        'sqlite' => array(
                'driver'   => 'sqlite',
                'database' => __DIR__.'/../database/production.sqlite',
                'prefix'   => '',
        ),

        'mysql' => array(
                'driver'    => 'mysql',
                'host'      => 'localhost',
                'database'  => 'database',
                'username'  => 'root',
                'password'  => '',
                'charset'   => 'utf8',
                'collation' => 'utf8_unicode_ci',
                'prefix'    => '',
        ),

        'pgsql' => array(
                'driver'   => 'pgsql',
                'host'     => 'localhost',
                'database' => 'database',
                'username' => 'root',
                'password' => '',
                'charset'  => 'utf8',
                'prefix'   => '',
                'schema'   => 'public',
        ),

        'sqlsrv' => array(
                'driver'   => 'sqlsrv',
                'host'     => 'localhost',
                'database' => 'database',
                'username' => 'root',
                'password' => '',
                'prefix'   => '',
        ),

),
```

We see an array called **connections** and some database configurations inside it. It's clearly that we need to fill in our database data to create a connection to our Laravel app. We use MySQL, so for

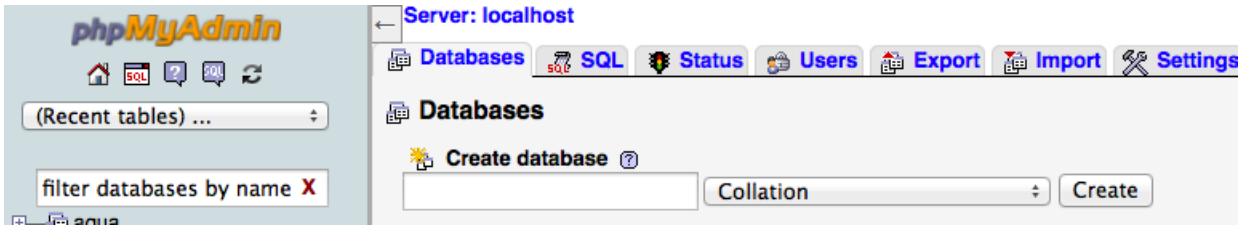
example, we can fill in something like this:

```
        'mysql' => array(
                'driver'    => 'mysql',
                'host'      => 'localhost',
                'database'  => 'todo',
                'username'  => 'root',
                'password'  => 'YourPassword',
                'charset'   => 'utf8',
                'collation' => 'utf8_unicode_ci',
                'prefix'    => '',
        ),
```

How to install MySQL?

When you install WAMP or XAMPP, you have MySQL and phpMyAdmin (a free software tool written in PHP, intended to handle the administration of MySQL over the Web) already. For Mac, you can install MAMP[35] to have MySQL. It's the easiest way to have a database platform for your system. Usually, default username of MySQL is **root** and there is no MySQL password by default. You can leave it blank or find a way to set a password for MySQL.

When we have phpMyAdmin installed. We can easily create a database via **phpMyAdmin Databases** tab.



Create database using phpMyAdmin

We will create a database called **todo**, so fill the name in and click **Create button** to create our first database!

Great! We have finished setting up our database.

Need a phpMyAdmin alternative?

Here is a little tip: There is a great app for Mac, it's called Sequel Pro. You can use it to work with MySQL. Many developers are using it, including me. Unfortunately, it's not available for Windows. Therefore, I use phpMyAdmin for this book.

[35] http://www.mamp.info

Now, it's time to go to next section and create some database tables.

Meet Schema Builder

Database is not simple. It has its own structure, different types, relationships and syntax. In order to follow along this chapter easily, you need a basic understanding of SQL. At least, you should know the concept of a database, how to read, update, modify and delete it. I will try to include a section about database in our Appendices part, but for now, if you don't know anything about database, a good place to learn about database is:

W3Schools SQL[36]

**Note: Don't worry, if you don't want to learn about database now, you can skip it. It's boring, I know. Just read this chapter. If you don't understand something, you have a place called W3Schools to go.*

Before doing anything with our database, we need to create and define its structure. Laravel has a class, called Schema. This class can be used to manipulate database tables. The great thing is, it works with all database platforms; thus we don't need to worry about specific database's syntax.

In this section, we will learn how to use Schema class and write some schema building codes to build our database structure.

Creating Tables

Before creating a table, we need a database first. Luckily, we have created a database in the previous section, it's called **todo**. If you don't create it yet, let's create it now.

We can use **create()** method of Schema class to create table. Open our **app/routes.php** file and write (make sure to replace the old route):

```
1  Route::get('/', function())
2  {
3          Schema::create('tasks', function($table)
4          {
5
6          });
7  });
```

As you can see, the **Schema::create()** method has two parameters:

The first parameter is the name of the table that we want to create. We're creating a To-do list, so we create a table called **tasks**.

[36] http://www.w3schools.com/sql

The **tasks** table will contain many tasks; thus, it should be plural and it should be lowercase as well. It's a naming convention that many developers use. We don't use uppercase for database tables and columns, and spaces should be replaced by _ (underscores).

The second parameter is a function. In PHP, we called it **Closure**. The Closure has one parameter, which is $table. You can use any name that you wish to use.

A table should have an auto incrementing column and this column will also be a primary key of the table. We can create the column by adding:

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->increments('id');
        });
});
```

It's very simple, right? Please note that if we don't create any column when creating a table, we will get an error.

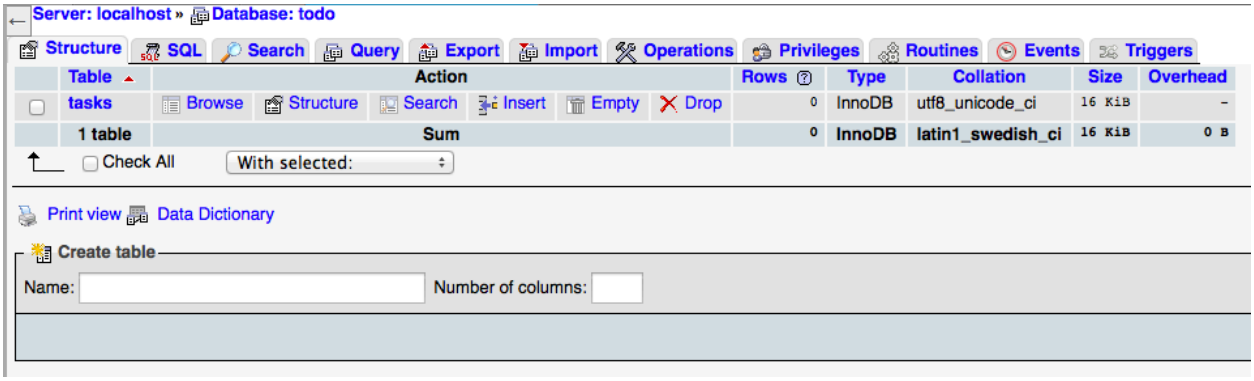
Go to our app's home page, you should see a blank white page.

If you don't see a blank white page, and what you see is a PDOException error page with a big text, like this:

```
PDOException
SQLSTATE[HY000] [2002] No such file or directory
```

It could be a MySQL's path problem. You can fix it by replacing **localhost** with **127.0.0.1** in **app/config/database.php**

Good! To check if we have already created a table yet, let's go to phpMyAdmin. Go to our **todo** database, you should see the **tasks** table, like this:



Our tasks table in phpMyAdmin

Cool! We have created a database table! Now, let's...drop it!

Dropping Tables

Dropping a table is easy. We can use Schema::drop() method:

```
<!--app/views/home.blade.php-->

Route::get('/', function())
{
        Schema::drop('tasks');
});
```

As you see, we just pass our table name (which is tasks) as the first parameter of **Schema::drop()** method to remove the table.

However, we will get an error if a table that we want to drop doesn't exist. To solve this problem, we can use **Schema::dropIfExists()** method:

```
<!--app/views/home.blade.php-->

Route::get('/', function())
{
        Schema::dropIfExists('tasks');
});
```

This method checks if there is a table called tasks. If the tasks table exists, it will drop that table for us.

Great! We now can create and drop tables. It's time to add some columns into our tables.

Adding Columns Into A Table

Adding a column is easy. The only thing you need to remember is column types' name. There are many column types. Luckily, Laravel has provided us many methods to easily add columns. I will list only some column types, that we use frequently:

Increments

In my opinion, the most important method for building database structure is increments() method. This method adds an auto increment primary key to our table.

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->increments('id');
        });
});
```

Text

The text() method is used to store large text. For example: description, blog post, comments, etc.

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->text('description');
        });
});
```

We also have mediumText() and longText() method.

Integer

You can guess it by its name right? Integer() method is used to store integer. For example: size, score, id, etc.

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->integer('scores');
        });
});
```

We also have bigInteger(), mediumInteger(), and smallInteger() methods.

Float

If you want to store floating point number (a number that can contain a fractional part), use float() method:

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->float('scores');
        });
});
```

Decimal

If you like to use decimal values instead of float, you can use decimal method:

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->decimal('amount', 7, 2);
        });
});
```

This method is a bit different. It accepts two optional parameters. The first parameter represents the length of the number, the second parameter represents the digits to its right.

Boolean

Boolean has only 2 value. For example: true or false, 1 or 0, approved or declined, etc. You can create boolean colum by using boolean() method:

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->boolean('approved');
        });
});
```

String

The string() method is used to store a varchar. We usually use string to store email, username, etc.

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->string'email');
        });
});
```

We can also define the length of a string by use the optional second parameter:

```
$table->string('name', 100);
```

Time

If we want to store time, we will use time() method.

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->time'sunset');
        });
});
```

Date

Date() method is used to store...date, of course!

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->dateTime('created_at');
        });
});
```

This method also store time as well.

Binary

The last frequently used one is binary() method. We usually use it to store binary files, such as: uploaded files, data, images, etc.

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->binary('banner');
        });
});
```

Timestamp

In case we need to store time in a binary Timestamp format, we can use timestamp() method:

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->timestamp('updated_at');
        });
});
```

Timestamps

Laravel has a special method, which is **timestamps** (it's not **timestamp**). This method doesn't have any parameter. It creates two column: created_at and updated_at. These column are used to keep track when a row is created or updated.

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->timestamps();
        });
});
```

Other Column Types

You can view other column types at official documentation as always:

Schema Official Docs[37]

Or you can view it in Part 3, which is our Alternative Documentation (will be added later).

Ok, we have learned about column types, it's time to create a complete table for our To-do List app.

[37] http://laravel.com/docs/schema

```
Route::get('/', function())
{
        Schema::create('tasks', function($table)
        {
                $table->increments('id');
                $table->string('title');
                $table->text('body');
                $table->integer('user_id');
                $table->boolean('done');
                $table->timestamps();

        });
});
```

As you see, our tasks column will have many tasks. A task should have an id (which is auto incrementing field), a title, a body text (description), a user_id (which is used to know who create the task), a boolean (to define whether it's completed or not), and a timestamps (to keep track when a task is created or updated).

Introducing Migrations

If you're working in a team, it would be messy if you're creating a database structure without anything to keep track, and "roll back" when you make any mistakes, right?

For example, you're building a database structure, and the other team members are also building another database structure at the same time, or they want to modify your database structure to do something. Things will be complicated. You will need a way to keep your database synchronized.

Laravel brings you a way to do that, it's called Migrations. Basically, Migrations are PHP scripts that you can use to build or modify your database. Migrations keep a record of what you do and help you to have the same database structure with your team.

Let's try to create migrations to understand more clearly.

Creating Migrations

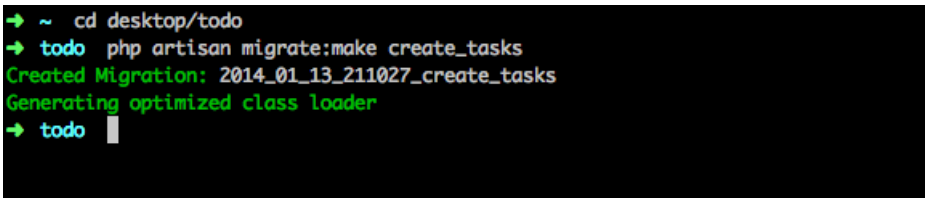
In order to create a migration, we will need to use PHP Artisan. First, we open Terminal (or Command Prompt/Git Bash on Windows). After that, we navigate to our project folder:

```
cd desktop/todo
```

Note: I put my project on desktop, and my app is called todo. Your path and your project name may be different.

We will try to recreate our tasks table, but we'll use migrations this time. When we're in our project folder, executing this command:

```
php artisan migrate:make create_tasks
```



Executing migrate:make command

When we execute Artisan **migrate:make** command, Laravel creates a new migration template with a timestamp for us. The template is located in **app/database/migrations**. **create_tasks** is the name of the template, you can name it whatever you like. For example, you can find in the migrations folder a file look like this:

```
2014_01_13_110107_create_tasks.php
```

*Please note that the timestamp (2014*01*13_110107) could be different. *

Open it with a text editor, we should see something like:

```
<?php

use Illuminate\Database\Migrations\Migration;

class CreateTasks extends Migration {

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

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

```
23          {
24                  //
25          }
26
27  }
```

There are 2 methods, **up()** and **down()**. We will write our schema building code inside those methods. When we need to do something, we write it in up() method. When we need to undo something, we write it in down() method. Easy?

Let's try to create our tasks table by filling the up() method:

```
1   public function up()
2   {
3           Schema::create('tasks', function($table)
4           {
5                   $table->increments('id');
6                   $table->string('title');
7                   $table->text('body');
8                   $table->integer('user_id');
9                   $table->boolean('done');
10                  $table->timestamps();
11          });
12  }
```

We just put the same codes, that we use in previous section, into the up() method.

Up() method is used to create our tasks table, so down() method is used to drop it.

```
1   public function down()
2   {
3           Schema::drop('tasks');
4   }
```

Remember, when we create something in a up() method, always make sure that we remove it in a down method().

If we need to create a migration faster, we can use:

```
1   php artisan migrate:make create_tasks --create --table=tasks
```

By adding **–create –table=tasks**, migrations will automatically create a template with a tasks table for us. The template looks like:

```php
<?php

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

class CreateTasks extends Migration {

        /**
         * Run the migrations.
         *
         * @return void
         */
        public function up()
        {
                Schema::create('tasks', function(Blueprint $table)
                {
                        $table->increments('id');
                        $table->timestamps();
                });
        }

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

}
```

It's very clever, right?

We're not finished yet. We've just created a migration template. We must execute it to create our tasks table.

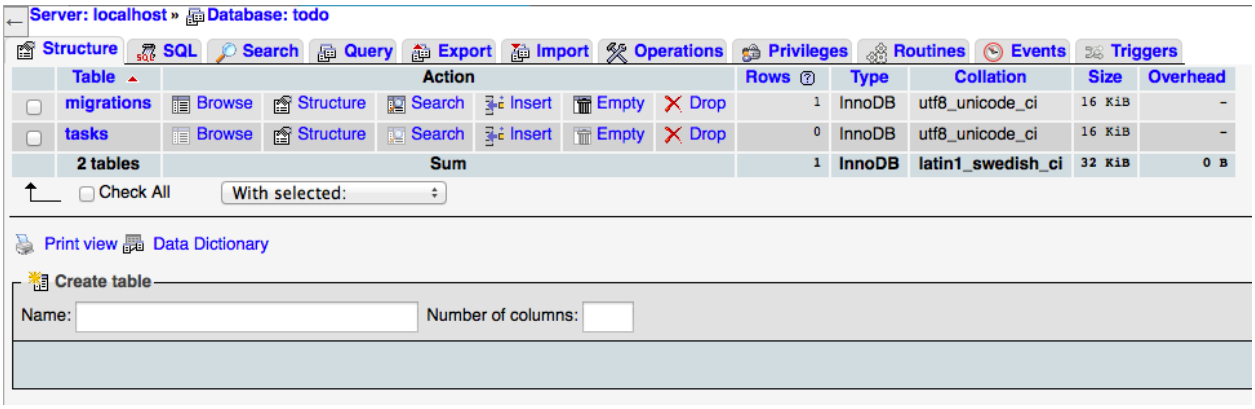
To run our migration, it's very easy:

```
php artisan migrate
```

Note: make sure that you've deleted the tasks table before running migrations.

Great! We've just created our **tasks table** again using migrations! When you use **php artisan migrate**, Laravel will look for all migration templates, and execute all up() methods, if those methods are not executed yet.

If everything is ok, check your database using phpMyAdmin or any database tool you like, we should see:



Our database structure in phpMyAdmin

There are 2 tables: migrations and tasks. Hey, but what is migrations table? We don't create it?

Well, Laravel checks and creates a **migrations** table for us, if the table doesn't exist. Migrations table is used to keep track of all the migrations that have already run for our application. It's a part of migrations system. If you don't like its name, you can change it in **app/config/database.php**:

```
1  'migrations'=>'migrations',
```

Change ⇒'**migrations**' to whatever you like.

Here is a little trick, if we want to see the intended SQL result of a migration command, we can use:

```
1  php artisan migrate --pretend
```

Basically, adding **–pretend** will show us the SQL queries, that would be executed:

```
➜ todo  php artisan migrate --pretend
CreateTasks: create table `tasks` (`id` int unsigned not null auto_increment primary key, `title` varchar
(255) not null, `body` text not null, `user_id` int not null, `done` tinyint(1) not null, `created_at` ti
mestamp default 0 not null, `updated_at` timestamp default 0 not null) default character set utf8 collate
 utf8_unicode_ci
➜ todo
```

Using –pretend

Rolling Back Migrations

Ok, we've executed the up() method and created our tasks table. If we want to drop the table, we can use **migrate:rollback** command:

```
php artisan migrate:rollback
```

Cool! We've just dropped our table by running a migration command. The command executed our down() method.

A little note, **migrate:rollback** command rolls back only the migrations that were ran that last time we used migrate command. If we have many migrations, and we need to roll back all of them, we use **migrate:reset** command:

```
php artisan migrate:reset
```

At this time, you can be able to create a database structure using migrations. You can try to practice by creating some tables and then roll them back.

The ability to create tables and drop/rollback them by using only a single line of code is really nice, right?

In the next section, we will learn one of the most important features of Laravel 4: **Eloquent ORM**.

Hi Eloquent ORM!

In the past, working with databases is a pain. When we build big applications, there are many SQL statements and PHP codes. They made the applications look messy and hard to maintain. Fortunately, Laravel has introduced Eloquent ORM to allow developers use Active Record pattern. Active Record pattern is a technique of wrapping database into objects. By using this technique, developers can present data stored in a database table as class, and row as an object. Each database table has a corresponding "Model" which is used to interact with that table.

It helps to make our codes look cleaner and readable. We can use Eloquent ORM to create, edit, manipulate, query and delete entries easily.

More than that, Eloquent ORM has built in relationships, which allow us to not only manipulate one table in the database, but also we can be able to manipulate all related data.

When Laravel 4.1 is introduced, it also includes one of the most requested features from the community, which is Polymorphic ManyToMany-relationships. This way we can attach a polymorphic relationship to multiple models without writing a lot of code. It makes Eloquent even more pleasant to use.

Eloquent ORM is getting better day by day. By using it, we don't even have to write a single line of SQL. Let's start to learn it!

If you know OOP (Object Oriented Programming), you should know that we will have objects within our Laravel application. Right?

Hey, what is OOP?

In case you don't know about OOP, I'm happy to explain it. Object Oriented Programming is a programming paradigm that represents concepts as objects and associated procedures known as methods. OOP is organized around objects and data rather than actions and logic. Examples of objects can be: a person (name, address, email, etc), your computer (mouse, keyboard, monitor), and so forth. If you still don't get it, well, let's move on and do something to understand the concept easier.

It's all about objects, so we need to identify objects within our Laravel applications first. We're creating a To-do list application; thus, our object will be task.

Creating New Objects With Eloquent ORM

In order to create a task, we need to create a new migation to build our tasks table. We've done it in previous section. If you don't create it yet, go ahead and create it:

```
1  // app/database/migrations/2014_01_13_110107_create_tasks.php
2
3  <?php
4
5  use Illuminate\Database\Schema\Blueprint;
6  use Illuminate\Database\Migrations\Migration;
7
8  class CreateTasks extends Migration {
9
10         /**
11          * Run the migrations.
12          *
13          * @return void
14          */
15         public function up()
16         {
17                 Schema::create('tasks', function($table)
18                 {
19                         $table->increments('id');
20                         $table->string('title');
21                         $table->text('body');
22                         $table->integer('user_id');
23                         $table->boolean('done');
24                         $table->timestamps();
25                 });
```

```
26          }
27
28          /**
29           * Reverse the migrations.
30           *
31           * @return void
32           */
33          public function down()
34          {
35                  Schema::drop('tasks');
36          }
37
38  }
```

Something is not quite right here? You may think that we create **tasks** table, but our object is **task**?

Well, Eloquent ORM is very smart, it will automatically look for the plural form of the model name as the table (in this case: tasks). You can also change it, but actually, we don't need to do that.

Important notes:

1. Eloquent ORM requires a model to have an auto incremental column (for example: id). Therefore, make sure that we always have it in our models.
2. Eloquent always tries to populate updated_at and created_at columns of our table with the current time. Those two columns are created by using **timestamps()** method. Therefore, make sure to include timestamps() when building our schema as well.

Now, we need to create a new Eloquent model to represent our tasks. It's very easy, we create a new file, called **Task.php**, in **app/models**:

```
1  <?php
2
3  // app/models/Task.php
4
5  class Task extends Eloquent
6  {
7
8  }
```

Done! We've just created a new Eloquent model. Amazing?

When we have an Eloquent model that can be used to represent our tasks, it's time to add some data to it. I mean, we're going to create a new task by replacing our homepage route:

```
// app/routes.php

Route::get('/', function()
{
        $task = new Task;
        $task->title = 'Eating breakfast';
        $task->body = 'Remember to buy bread, egg and milk.';
        $task->save();
});
```

Take a look at it! We don't even have to write any SQL query! So cool, right?

Let me explain it a bit:

```
$task = new Task
```

First, we create a new instance of our Task model. When we use new Task, Laravel understands that we want to use tasks table.

```
$task->title = 'Eating breakfast';
$task->body = 'Remember to buy bread, egg and milk.';
```

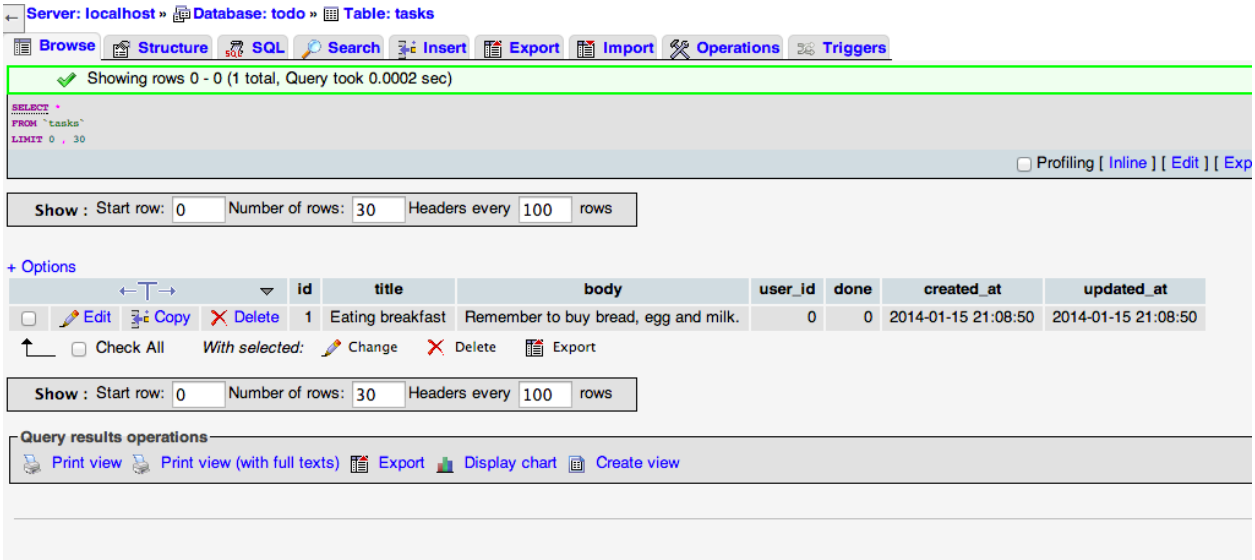
After that, we assign a title and a body (description) to our task.

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

Finally, we use **save()** method to save it. When we call save(), Eloquent ORM inserts data into our table!

Go to our app's homepage, we should see a blank white page again.

After that, we can check our database by using phpMyAdmin (click Browse at the tasks table):



Our first task!

Great! We have created new rows without using any SQL query!

Eloquent is smart and it's very flexible, too. If we want to use a different table name (instead of tasks, for example), we can do that by adding this line to our **Task.php** file:

```
1  <?php
2
3  // app/models/Task.php
4
5  class Task extends Eloquent
6  {
7          public $table = 'other_tasks';
8  }
```

We use a public $table attribute, and give it whatever table name that we like.

Disable or Enable timestamps() Method

If we want to use existing database table or we don't want to have created_at and updated_at columns, we can easily disable this functionality by adding a new public attribute to our model:

```php
<?php

// app/models/Task.php

class Task extends Eloquent
{
        public $timestamps = false;
}
```

Basically, Eloquent class has a boolean value, which can be used to turn on or off the timestamp functionality. By default, it's true, if we want to turn the functionality off, we set it to false.

Even though we can turn the timestamps() method off, but it's recommended that we should always have it to keep track of all database changes.

In case we want to add timestamps() method to a database table, we can create a migration and use:

```php
$table->timestamps();
```

For example, if our tasks table doesn't have the two columns (created_at and updated_at), we can add these columns by using **Schema::table**:

```php
public function up()
{
        Schema::table('tasks', function($table)
        {
                $table->timestamps();
        });
}
```

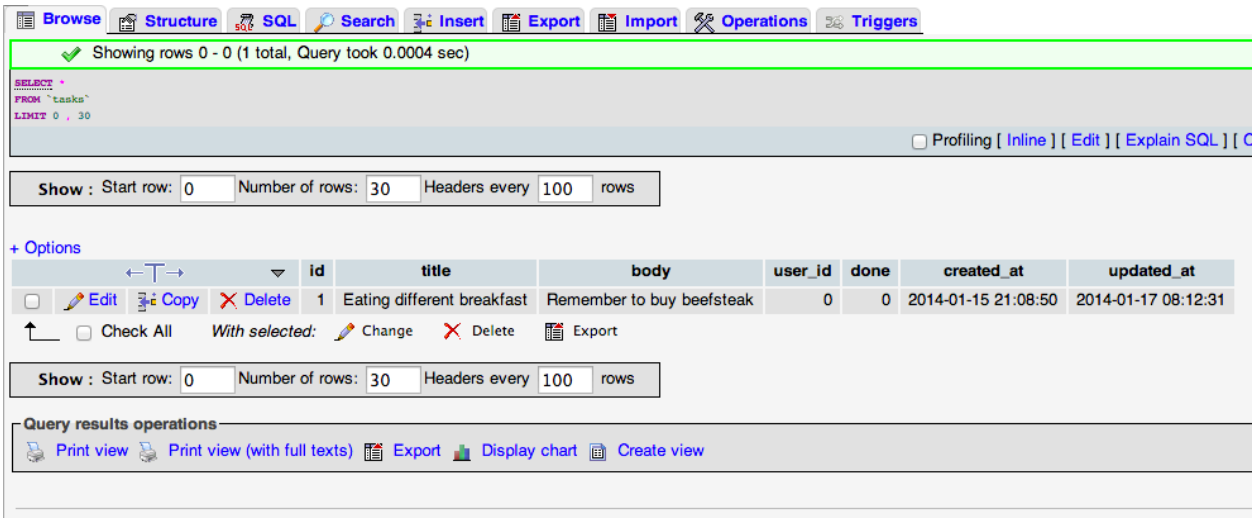
Updating Eloquent Models

To update an Eloquent model, we can use **find()** method to find an existing database row, then we can change it accordingly. For example, the last row, that we've just created in the previous section, has 1 as its id value. We can use find() method to find the instance by giving it the known id:

```
// app/routes.php

Route::get('/', function()
{
        $task = Task::find(1);
        $task->title = 'Eating different breakfast';
        $task->body = 'Remember to buy beefsteak';
        $task->save();
});
```

Visiting our home page again, we should see a blank white page.

To make sure that our task has been updated, we can check it via phpMyAdmin:



Our modified task

Cool! We've just updated our task. Take a look at updated_at column, we should see that it's updated automatically, too!

Deleting Eloquent Models

To delete an Eloquent model, we have three methods to do:

Using delete() method

We can use the **find()** method again to find the model's instance that we want to delete:

```
// app/routes.php

Route::get('/', function()
{
        $task = Task::find(1);
});
```

After that, we use **delete()** method to delete it:

```
// app/routes.php

Route::get('/', function()
{
        $task = Task::find(1);
        $task->delete();
});
```

Using destroy() method

Alternatively, we can use **destroy()** method to find and delete our Eloquent model's instances:

```
// app/routes.php

Route::get('/', function()
{
        Task::destroy(1);
});
```

The destroy() method accepts the id of the instances as its parameter.

We can also delete multiple instances by putting multiple ids:

```
// app/routes.php

Route::get('/', function()
{
        Task::destroy(1,2,3,4,5,6,7);
});
```

The good thing is, destroy method() also accepts an array:

```
// app/routes.php

Route::get('/', function()
{
        Task::destroy(array(1,2,3,4,5,6,7));
});
```

Using softDelete

Sometimes, we will need to soft delete a model, it means we don't actually remove it from our database. Instead, a **delete_at** timestamp is set. For example, we can use this softDelete functionality to keep track of what have been deleted.

***Note**: You can simply watch this section or you can skip it. It's not important for now. I just want you to know about it*

This softDelete functionality is not enabled by default. We need to enable it:

```
<?php

// app/models/Task.php

class Task extend Eloquent {
        protected $softDelete = true;
}
```

After that, we should create a migration, and add detete_at column into our table by using **softDeletes()** method:

```
public function up()
{
        Schema::table('tasks', function($table)
        {
                $table->softDeletes();
        });
}
```

Good job! When we want to update the **delete_at** timestamp, we only need to call **delete()** method and the delete_at will be set to the current timestamp.

Please note that, when we enable softDelete, the model is not actually deleted, it's still in our database. If we really need to remove the model, we can use forceDelete() method, for example:

```
// app/routes.php

Route::get('/', function()
{
        $task = Task::find(1);
        $task->forceDelete();
});
```

Usually, we don't use this softDelete functionality. However, it's still good to know it. For more information, you can look at the official documentation.

Reading and Displaying Eloquent Models

There are many ways, many methods to "read" Eloquent models. We will learn all of them in this book. However, for now, we just need to know some of the most commonly used methods.

Using find() Method

The easiest way to read and display an Eloquent model is using **find()** method.

```
// app/routes.php

Route::get('/', function()
{
        $task = Task::find(1);
        return $task->title;
});
```

As you see, we use **find()** method to find an instance of our Task. After that, we can display the task's title.

Let's refresh our home page again and check if the title has been displayed succesfully. We should see:

```
Eating different breakfast
```

Well done!

You now can be able to create, read, update and delete (CRUD) Eloquent models! It's time to use what we just have learned, and build our To-do list!

Understanding Controllers

In the last chapter and previous sections, we know how to link routes to Closures. Basically, using Closures is a fast way to write our applications. However, Closures are only good for examples and small applications. If we want to build big applications, we need something that can group our route logic into a class and take advantage of more advanced frameworks features. Simply put, instead of using only a **routes.php** file for our application route logic, we will use a class to house the logic. That class is called **Controller**.

In the Controller class, we will put many public methods (similar to Closures). Those methods are known as **actions** (or **Controller actions**).

Let's create a Controller and change our application's route.php to understand it clearly!

Creating Our App Route Logic Using A Controller

Basically, our To-do list application will have four pages:

1. Home: Show all the tasks.
2. Create: Create new tasks.
3. Edit: Edit existing tasks.
4. Delete: Delete the tasks.

As you can see, our application has only a few pages now; thus, we need only one Controller.

First, we're going to create a Controller for our application. Controllers are stored in **app/controllers** folder by default. Therefore, we go there and create a new file, which is **TasksController.php**.

If we create a home page route using Closures, we will have something like this:

```
1  // app/routes.php
2
3  Route::get('/', function()
4  {
5          return View::make('home');
6  });
```

Nevertheless, if we create a home page route using Controller, it's a bit different:

```php
<?php
// app/controllers/TasksController.php

class TasksController extends BaseController
{
        public function home()
        {
                return View::make('home');
        }
}
```

It's our first controller! Simple, right? Both functions are very similar. The difference is, the Controller has a name, but the Closure doesn't have. Actually, we can use Controllers to do everything that we can do with Closures. So, when do we use Closures or Controllers?

1. If we want to do things fast and our project is small, use Closures.
2. If we want to have clean, neat, tidy codes and our project is big, use Controllers.

Take a look at the codes above, you can see that we put all the activities, that are related to a task, to the **TasksController**. In fact, we can name the Controller whatever you like, such as: Tasks, TaskControl, etc. Laravel automatically detects what we try to do when our controller extends **Controller** or **BaseController**. However, it's a naming convention that many developers use, so we should follow it.

Now, it's time to add more pages:

```php
<?php
// app/controllers/TasksController.php

class TasksController extends BaseController
{
        public function home()
        {
                return View::make('home');
        }

        public function create()
        {
                return View::make('create');
        }

        public function edit()
```

```
17          {
18                  return View::make('edit');
19          }
20
21          public function delete()
22          {
23                  return View::make('delete');
24          }
25  }
```

We have created other pages by implementing some new Controller actions for our applications. Those pages will be used to display all the tasks and forms (to create, edit, and delete tasks).

Last step, we just have a view for our home page, we don't have views for other pages yet! Let's create them:

app/views/create.blade.php

app/views/edit.blade.php

app/views/delete.blade.php

You can leave those views empty for now, we will go back and update them soon.

Linking A URI to A Controller

After we've created a Controller, you should notice that there is no URI in our Controller actions. At the moment, Laravel doesn't know how to direct its routes to our Controller yet.

We need to link a URI to our Controller to make it work. In order to link, we use Route::get() method again. Let's open the **routes.php** file:

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

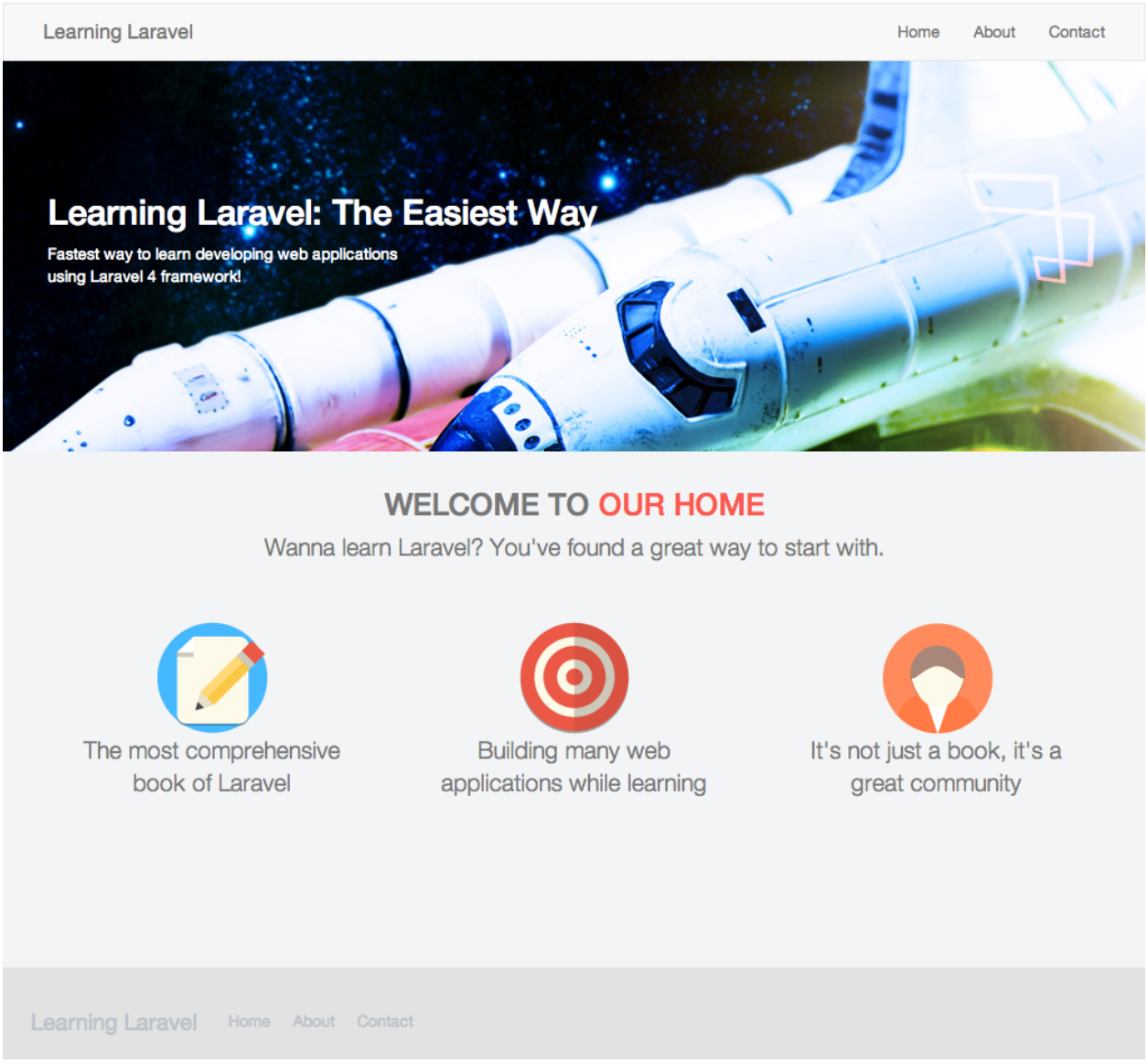
It's similar to the Closures method, but the second parameter is different. There is a string, with an @ (at) in the middle. **TasksController** is our class name, and **home** is the action that we want to route.

Now, try to create other routes yourself. You can delete other unrelated routes if you like.

When you're finished, we should have our **routes.php** file look like this:

```
Route::get('/','TasksController@home');
Route::get('/create','TasksController@create');
Route::get('/edit','TasksController@edit');
Route::get('/delete','TasksController@delete');
```

Good job! We should visit our home page again to make sure that our new Controller work:



If you see this, our new Controller works! Yay!

Display All Tasks

We have tasks in our database, and we should create a page to display them. How about changing the home page and display all the tasks? Let's do it!

We go to **TasksController.php** file and change our home action to:

```
1  public function home()
2  {
3          $tasks = Task::all();
4          return View::make('home', compact('tasks'));
5  }
```

We use **Task::all()** to get all tasks in our database. After that we use compact('tasks') to create an array, containing our tasks' variables and their values. And then we use View::make('home') to give $tasks variable to the home view.

compact() is a PHP array function, if you don't know about it, you can view it here:

Learn more about compact[38]

Alternatively, we can also use:

```
1  return View::make('home')->with('tasks', $tasks);
```

or

```
1  return View::make('home',['tasks'=> $tasks]);
```

Basically, those three methods are the same. You can use whatever you like to give a variable to the views.

Open the **home.blade.php**, modify its contents to:

[38] http://us2.php.net/manual/en/function.compact.php

```
@extends('layout')
@section('content')
                <section class="header section-padding">
                        <div class="background"> </div>
                        <div class="container">
                                <div class="header-text">
                                        <h1>Learning Laravel: The Easiest Way</h1>
                                        <p>
                                        This is our To-do list! <br />
                                        Built using Laravel 4 framework!
                                        </p>
                                </div>
                        </div>
                </section>

        <div class="container">
                        <section class="section-padding">
                                <div class="jumbotron text-center">
                                        //We will put our tasks here
                                </div>
                        </section>
                </div>
@stop
```

It should be easy to understand. We've just removed our home page contents and changed the header text.

I will put our tasks into a box, and use table tag to display them. In Twitter Boostrap 3, there is a component called **Panel**. We will use it to display our tasks:

```
<div class="panel panel-default">
      <div class="panel-heading">
              <h1>
                      <span class="grey">Our</span> To-do List
             </h1>
      </div>

      @if ($tasks->isEmpty())
        <p> Currently, there is no task!</p>
        @else
       <table class="table">
         <thead>
           <tr>
```

```
14              <th>#</th>
15              <th>Title</th>
16              <th>Body</th>
17              <th>Finish</th>
18            </tr>
19          </thead>
20          <tbody>
21                  @foreach($tasks as $task)
22                                  <tr>
23                                          <td>{{ $task->id }} </td>
24                                          <td>{{ $task->title }}</td>
25                                          <td>{{ $task->body}}</td>
26                                          <td>{{ $task->done ? 'Yes' : 'No'}}</td>
27                                  </tr>
28                          @endforeach
29          </tbody>
30        </table>
31    @endif
32 </div>
```

Let's take a look closer at them:

```
1 <div class="panel panel-default">
2
3 </div
```

First, we wrap our tasks into a panel.

```
1 <div class="panel-heading">
2         <h1>
3                 <span class="grey">Our</span> To-do List
4         </h1>
5 </div>
```

We use panel-heading to display our task header.

```
1 @if ($tasks->isEmpty())
2                 <p> Currently, there is no task!</p>
3 @endif
```

After that, we check if our $tasks variable is empty or not. If it's empty, we ouput a text to let our users know that there is no task.

Do you remember that we've used home action to pass the $tasks variable to this view?

```
@else
   <table class="table">
     <thead>
       <tr>
         <th>#</th>
         <th>Title</th>
         <th>Body</th>
         <th>Finish</th>
       </tr>
     </thead>
     <tbody>
            @foreach($tasks as $task)
                       <tr>
                              <td>{{ $task->id }} </td>
                              <td>{{ $task->title }}</td>
                              <td>{{ $task->body}}</td>
                              <td>{{ $task->done ? 'Yes' : 'No'}}</td>
                       </tr>
               @endforeach
     </tbody>
   </table>
```

Finally, if the $tasks is not empty, we create a table and use foreach() to loop over it and display our tasks.

If you don't know about foreach(), you can also view it here to see how it works:

Learn abour foreach()[39]

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

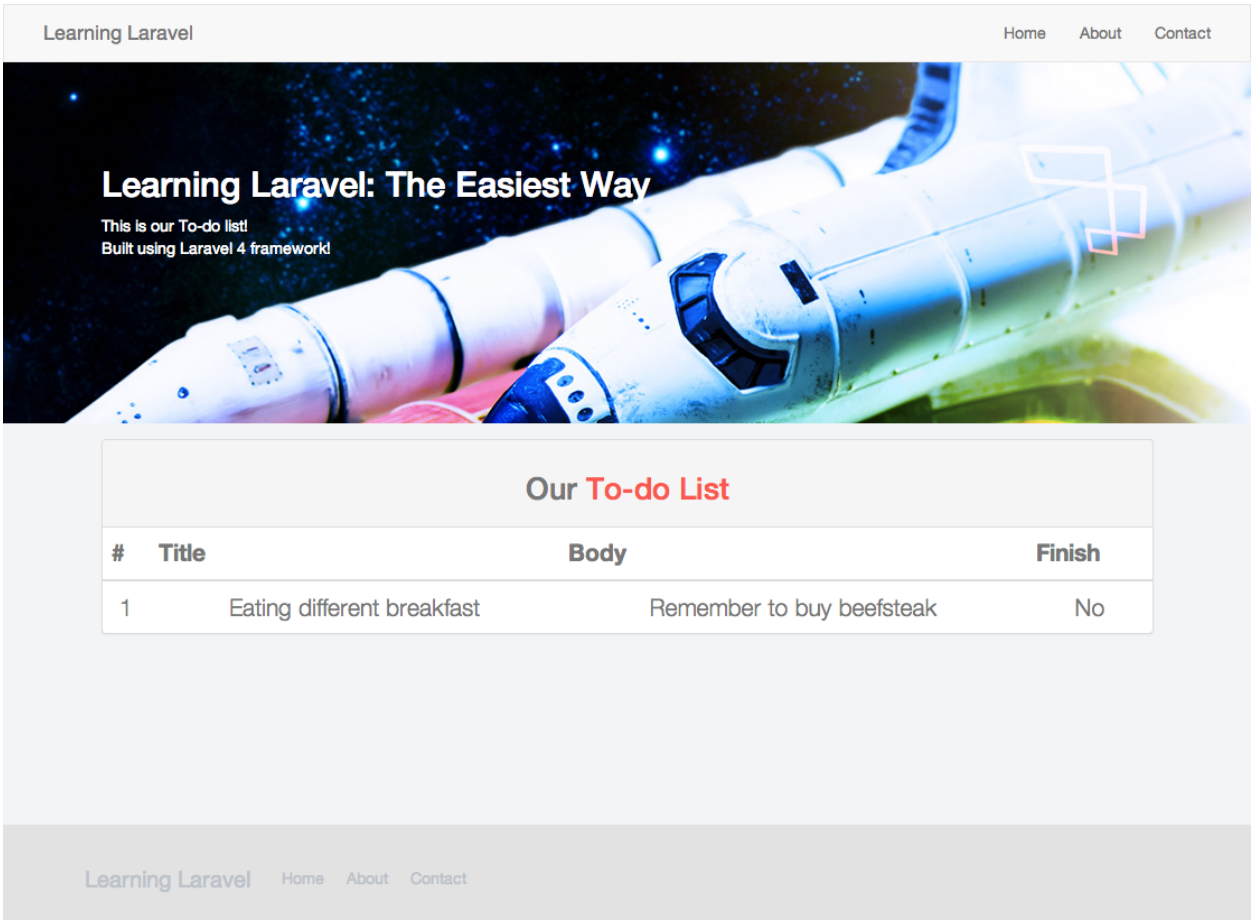
```
@extends('layout')
@section('content')
        <section class="header section-padding">
               <div class="background"> </div>
               <div class="container">
                       <div class="header-text">
                              <h1>Learning Laravel: The Easiest Way</h1>
                              <p>
                              This is our To-do list! <br />
                              Built using Laravel 4 framework!
                              </p>
```

[39] http://www.php.net/manual/en/control-structures.foreach.php

```
12                          </div>
13                  </div>
14          </section>
15
16      <div class="container">
17                  <section class="section-padding">
18                          <div class="jumbotron text-center">
19
20                                  <div class="panel panel-default">
21                          <div class="panel-heading">
22                          <h1>
23                                  <span class="grey">Our</span> To-do List
24                          </h1>
25                          </div>
26
27                          @if ($tasks->isEmpty())
28                                  <p> Currently, there is no task!</p>
29                                  @else
30                          <table class="table">
31                            <thead>
32                              <tr>
33                                <th>#</th>
34                                <th>Title</th>
35                                <th>Body</th>
36                                <th>Finish</th>
37                              </tr>
38                            </thead>
39                                   <tbody>
40                                          @foreach($tasks as $task)
41                                                  <tr>
42                                                  <td>{{ $task->id }} </td>
43                                                  <td>{{ $task->title }}</td>
44                                                  <td>{{ $task->body}}</td>
45                                                  <td>{{ $task->done ? 'Yes' : 'No'}}</td>
46                                                  </tr>
47                                                  @endforeach
48                            </tbody>
49                          </table>
50                          @endif
51                               </div>
52
53                          </div>
```

```
54                  </section>
55          </div>
56  @stop
```

If everything's ok, refresh our home page to see our tasks:



Our tasks

Create The Tasks

We definitely need a page to create our tasks. Let's create it.

First, we go to **layout.blade.php** to change our navigation bar:

```
<ul class="nav navbar-nav navbar-right">
  <li><a href="/">Home</a></li>
  <li><a href="/create">Create</a></li>
  <li><a href="/contact">Contact</a></li>
```

I change the about page link to create link, you can add a new link if you want.

Alternatively, we can also code like this:

```
<ul class="nav navbar-nav navbar-right">
 <li><a href="/">Home</a></li>
 <li><a href="{{ action('TasksController@create') }}">Create</a></li>
 <li><a href="/contact">Contact</a></li>
 </ul>
```

This time, we use **action()** function to automatically generate an URL for us. The parameter of the **action()** function is a controller and its action, with an @ in the middle.

Now, when we click on the **Create** link, we should be redirected to the **create page**.

The create page is blank, so let's add a form to it:

```
@extends('layout')
@section('content')
                <section class="header section-padding">
                        <div class="background"> </div>
                        <div class="container">
                                <div class="header-text">
                                        <h1>Create</h1>
                                        <p>
                                        Create tasks page
                                        </p>
                                </div>
                        </div>
                </section>

        <div class="container">
                        <section class="section-padding">
                                <div class="jumbotron text-center">
                                <h1>Create A Task</h1>

                                {{ Form::open(['url'=> '/create']) }}
                                <div>
```

```
                                        {{ Form::label('title', 'Title:') }}
                                        {{ Form::text('title') }}
                                    </div>

                                    <div>
                                        {{ Form::label('body', 'Body:') }}
                                        {{ Form::textarea('body') }}
                                    </div>
                                    <div>
                                        {{ Form::submit('Create Task') }}
                                    </div>
                                    {{ Form::close() }}

                                    </div>
                                </section>
                            </div>
@stop
```

You've learned how to create a form using Blade template in Chapter 1; thus, it should be easy for you, right?

```
{{ Form::open(['url'=> '/create']) }}
<div>
{{ Form::label('title', 'Title:') }}
{{ Form::text('title') }}
</div>

<div>
{{ Form::label('body', 'Body:') }}
{{ Form::textarea('body') }}
</div>
<div>
{{ Form::submit('Create Task') }}
</div>
{{ Form::close() }}
```

As always, we use Form::open() and Form::close() to create the form. The form has a title input field, a textarea (for the body text), and a submit button.

When the users submit the form, we want to define where the form is going to POST. Usually, it is where we put the form. In this case, it's create page, so we put /create URL there.

Switch back to our **routes.php** file, insert:

```
Route::post('/create', 'TasksController@saveCreate');
```

The above line means, listen for when we POST to the create page, and then call **saveCreate** action in the **TasksController** to handle the form.

You should notice that we use **Route::post** here, instead of **Route::get**.

Let's now edit the **TasksController.php** file. We don't have the **saveCreate()** action yet, so we're going to build it:

```
public function saveCreate()
{
        $input = Input::all();


        $task = new Task;
        $task->title = $input['title'];
        $task->body = $input['body'];
        $task->save();

        return Redirect::action('TasksController@home');
}
```

We use **Input::all()** to get all the input from the form. And then we assign **input['title']** and **input['body']** to our tasks' title and body. After that, we save the task to our database.

Here is a little tip, we can also code like this:

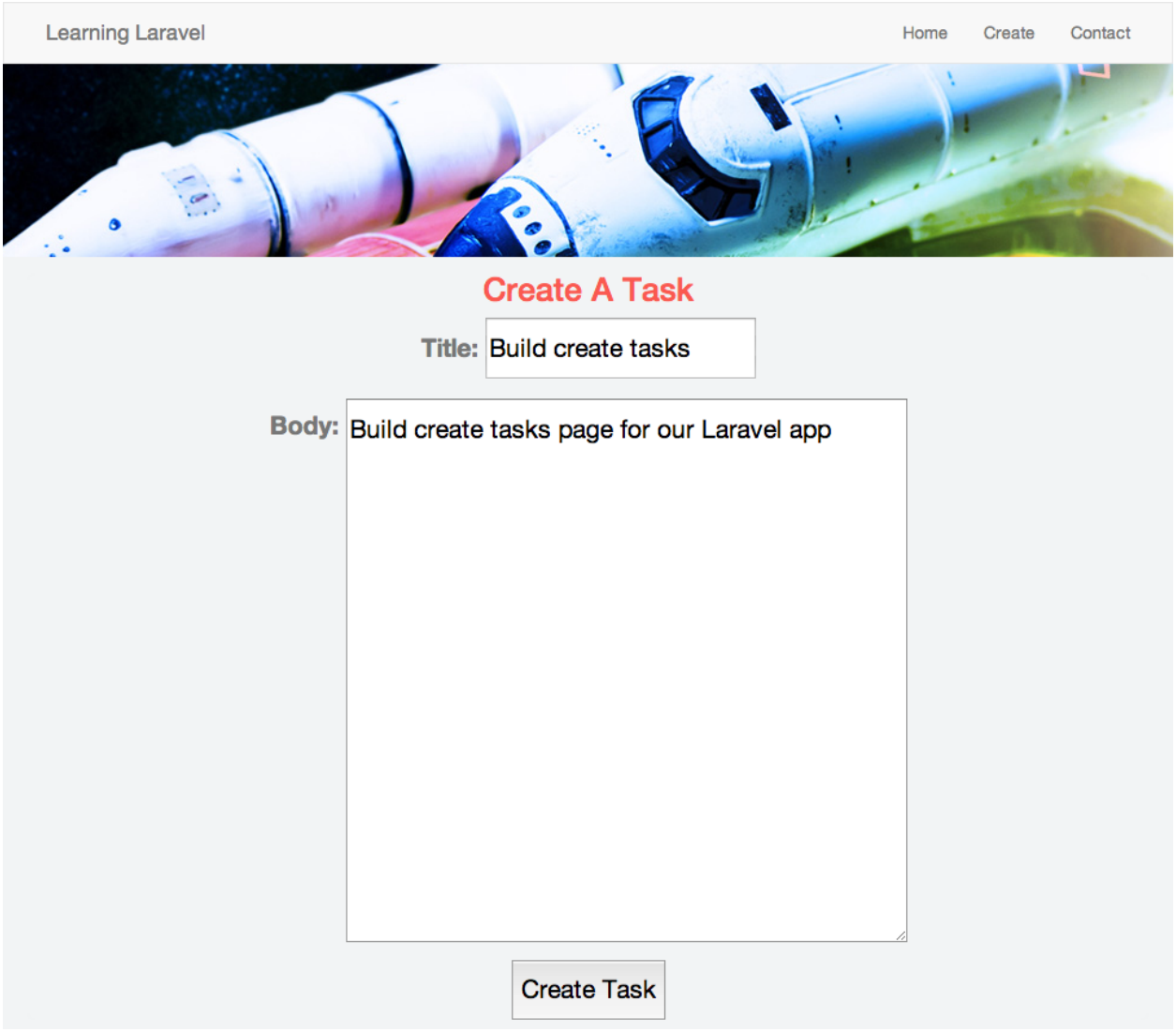
```
        $task = new Task;
        $task->title = Input::get('title');
        $task->body = Input::get('body');
        $task->save();
```

Instead of using **Input::all()** function, we can use **Input::get()** to get the input title and input body. Both methods work the same. You can choose to use whichever method you prefer.

Finally, after creating a task, we redirect users to our home page by using:

```
return Redirect::action('TasksController@home');
```

Go to the create page, we should see our form. Let's try to create a task:



Try to create a task

Hit **Create Task** button, we will see a new task! Congrats!

Adding Bootstrap Classes to The Form

Hey, our form looks ugly! How about adding some css classes to it and make it look nicer?

We use Bootstrap 3, so let's take a look at some Bootstrap form examples:

Bootstrap 3[40]

[40] http://getbootstrap.com/css/#forms

```
<form role="form">
  <div class="form-group">
    <label for="exampleInputEmail1">Email address</label>
    <input type="email" class="form-control" id="exampleInputEmail1"
    placeholder="Enter email">
  </div>
  <div class="form-group">
    <label for="exampleInputPassword1">Password</label>
    <input type="password" class="form-control" id="exampleInputPassword1"
    placeholder="Password">
  </div>
  <div class="form-group">
    <label for="exampleInputFile">File input</label>
    <input type="file" id="exampleInputFile">
    <p class="help-block">Example block-level help text here.</p>
  </div>
  <div class="checkbox">
    <label>
      <input type="checkbox"> Check me out
    </label>
  </div>
  <button type="submit" class="btn btn-default">Submit</button>
</form>
```

We can see that the form has **form** class, each div has **form-group** class and each input field has a class of **form-control**. How do we add it into our form using Laravel methods?

Let's add a class to our form first, find:

```
{{ Form::open(['url'=> '/create']) }}
```

Adding a class to our form:

```
{{ Form::open(['url'=> '/create', 'class' => 'form']) }}
```

Finding all **divs** in our form, and adding **form-group** class to them:

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

For the input fields, we need to add **form-control** class to them:

```
{{ Form::text('title', null, ['class'=>'form-control']) }}

{{ Form::textarea('body', null, ['class'=>'form-control']) }}
```

You may notice that there is a **null** in our code. It represents for the default value. If we don't want to have any default value, we put **null** there.

Finally, let's add btn class to our submit button:

```
{{ Form::submit('Create Task', ['class'=>'btn btn-primary']) }}
```

Here is the entire code:

```
@extends('layout')
@section('content')
        <section class="header section-padding">
        <div class="background"> </div>
        <div class="container">
                <div class="header-text">
                        <h1>Create</h1>
                        <p>
                        Create tasks page
                        </p>
                </div>
        </div>
        </section>

    <div class="container">
        <section class="section-padding">
                <div class="jumbotron text-center">
                <h1>Create A Task</h1>

                {{ Form::open(['url'=> '/create', 'class' => 'form']) }}
                <div>
                        {{ Form::label('title', 'Title:') }}
                        {{ Form::text('title', null, ['class'=>'form-control']) }}
                </div>

                <div>
                        {{ Form::label('body', 'Body:') }}
                        {{ Form::textarea('body', null, ['class'=>'form-control']) }}
                </div>
```

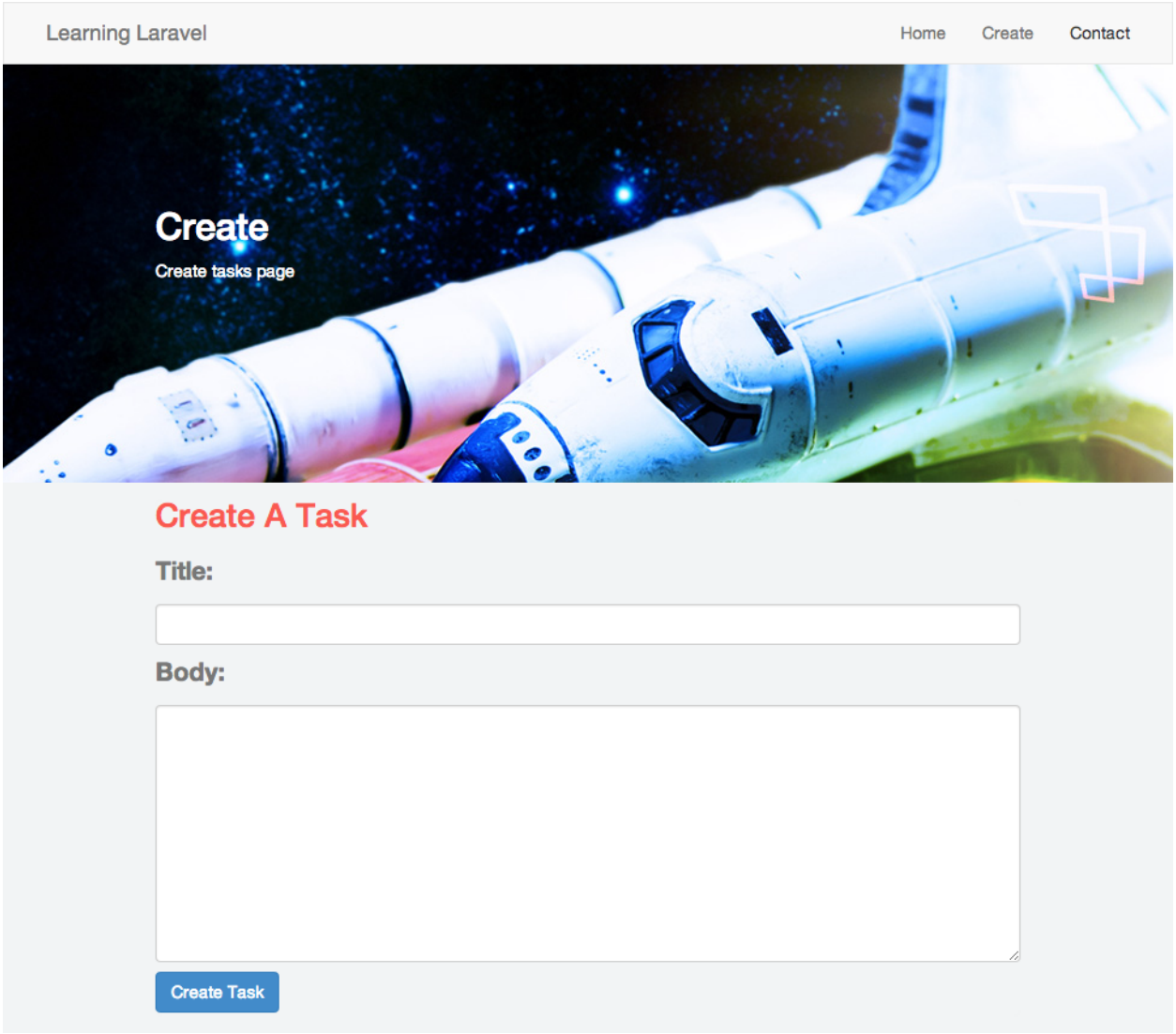
```
30                  <div class="form-group">
31                          {{ Form::submit('Create Task', ['class'=>'btn btn-primary']) }}
32                  </div>
33                  {{ Form::close() }}
34
35                  </div>
36          </section>
37          </div>
38  @stop
```

If we do everything correctly, refresh our create page, we should see a nice form.

However, by default, the texts are aligned center. I like to make them left-align. Open **public/css/style.css**, and add:

```
1  .text-center {
2      text-align: left;
3  }
```

Well done! Refresh our create page one more time:



Our new responsive create tasks form!

Edit Tasks

By now, we can create and view our tasks, the hardest part is to build an edit tasks form.

First, we need to create a link to the edit tasks page. Open **home.blade.php** and find:

```
1  <td>{{ $task->done ? 'Yes' : 'No'}}</td>
```

Insert below this line:

```
<td>
<a href="{{ action('TasksController@edit', $task->id) }}"
class="btn btn-info">Edit</a>

<a href="{{ action('TasksController@delete', $task->id) }}"
class="btn btn-info">Delete</a>
</td>
```

We've just created two buttons: Edit and Delete tasks button. You should notice that there is a second parameter ($task->id). That's the id of the task which we want to delete or edit.

For example, if we want to edit task 1, the url will look like:

```
http://localhost/edit/1
```

Good, now open **routes.php** file and modify our routes:

Add this line at the top:

```
Route::model('task','Task');
```

And change edit route to:

```
Route::get('/edit/{task}','TasksController@edit');
```

There is a new **Route::model** method! It's the new neat feature that was added to Laravel 4. The feature is called **Route Model Binding**. It gives us a convenient way to inject model instances into our routes.

Simply put, we let Laravel know that any route parameter named task (edit/{task}) will be bound to Eloquent Task model. Laravel then automatically detects the model id and pass the right model instance to the Controllers.

For example, if we access edit/1, Laravel will look for a task with an id of 1 and then pass it to **TasksController's edit action**.

Now, let's change the edit action. Open **TasksController.php** and edit:

```
public function edit(Task $task)
{
        return View::make('edit', compact('task'));
}
```

After getting the model instance, we use View::make and the compact() method again to create the view data array and pass it to our edit form.

It's time to build the edit page and edit tasks form, let's open **edit.blade.php**:

```
@extends('layout')
@section('content')
<section class="header section-padding">
<div class="background"> </div>
<div class="container">
        <div class="header-text">
                <h1>Edit</h1>
                <p>
                Edit tasks page
                </p>
        </div>
</div>
</section>

<div class="container">
<section class="section-padding">
<div class="jumbotron text-center">
<h1>Edit Task {{ $task->id }}</h1>

{{ Form::open(['url'=> '/edit', 'class'=>'form']) }}
{{ Form::hidden('id', $task->id)}}

<div class="form-group">
        {{ Form::label('title', 'Title:') }}
        {{ Form::text('title', $task->title, ['class' => 'form-control']) }}
</div>

<div class="form-group">
        {{ Form::label('body', 'Body:') }}
        {{ Form::textarea('body', $task->body, ['class' => 'form-control']) }}
</div>
<div class="form-group">
        {{ Form::label('done', 'Done:') }}
        {{ Form::checkbox('done', 1, $task->done) }}
</div>
<div class="form-group">
        {{ Form::submit('Save Task', ['class' => 'btn btn-primary']) }}
</div>
{{ Form::close() }}

</div>
</section>
```

```
</div>
@stop
```

This page is similar to the create page. We also use Form::open and Form::close to create the form. We also define where the form is going to POST. It will be the edit page, so we put /edit there.

To edit a task, we need to give Laravel the **task id**. We can do that by creating a hidden input type:

```
{{ Form::hidden('id', $task->id)}}
```

Alternately, we can also use this:

```
<input type="hidden" name="id" value="{{ $task->id }}">
```

Both methods are the same, but the Laravel method is much shorter, right?

```
<div class="form-group">
        {{ Form::label('title', 'Title:') }}
        {{ Form::text('title', $task->title, ['class' => 'form-control']) }}
</div>

<div class="form-group">
        {{ Form::label('body', 'Body:') }}
        {{ Form::textarea('body', $task->body, ['class' => 'form-control']) }}
</div>
```

Next, we create task title and body form. We want to display a default value for the users to easily change, so we put **$task->title** and **$task->body** in the second parameter, instead of **null**.

```
<div class="form-group">
        {{ Form::label('done', 'Done:') }}
        {{ Form::checkbox('done', 1, $task->done) }}
</div>
```

This will create a checkbox for us to mark if the task is finished or not. Let's take a look at how we can generate a checkbox form element using Laravel. The first parameter is the field **name** attribute, the second parameter is the field **value**, the third parameter is the optional default value.

We put **$task->done** there, so if the task is done, it will be checked when we access our form; and if the task is not done yet, it will be unchecked.

Two steps left, we need to define our edit route to make it listen for our form when we POST, and call an approriate action:

```
Route::post('/edit', 'TasksController@doEdit');
```

As you see, we will handle the form by using the **doEdit** action, let's open **TasksController.php** again and create it:

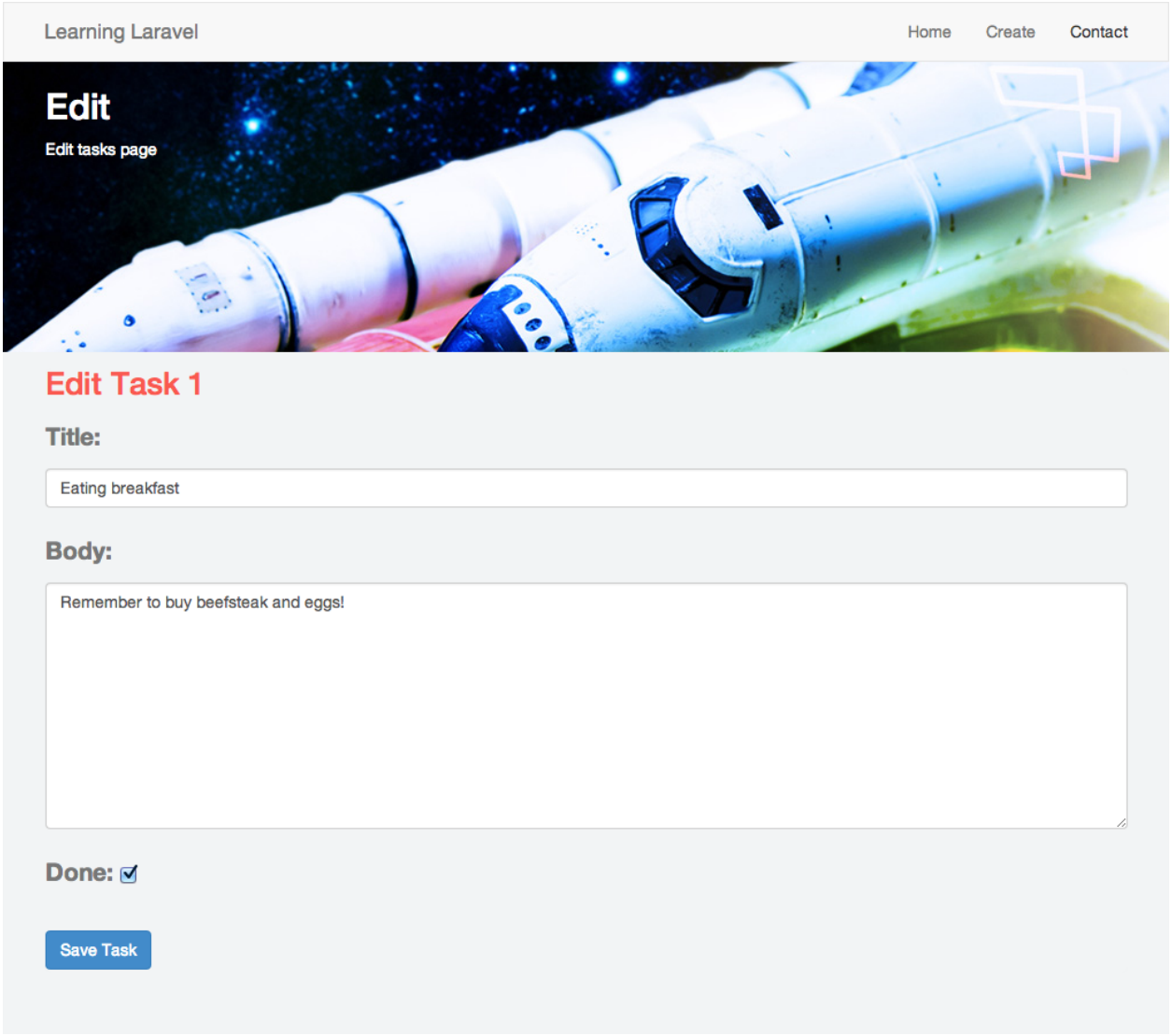
```
public function doEdit()
{
        $task = Task::findOrFail(Input::get('id'));
        $task->title = Input::get('title');
        $task->body = Input::get('body');
        $task->done = Input::get('done');
        $task->save();

        return Redirect::action('TasksController@home');
}
```

We have known how to update our tasks using Eloquent ORM, so this is very easy.

Instead of using **find()** method, we use **findOrFail()** method. It's similar to the find() method, but it will throw an 404 error event, if Laravel can't find any object with the input id.

The rest of the form is fairly standard. We save the form to our database and redirect users to the home page.

Excellent! We should have a nice edit task page:



Our edit task page

You can try to create the delete page now. It's very simple.

Delete Tasks

The way to build delete tasks page is pretty similar to how we build the edit page.

Do you remember that we already have the delete button in our **home.blade.php**?

```
<td>
<a href="{{ action('TasksController@edit', $task->id) }}"
class="btn btn-info">Edit</a>

<a href="{{ action('TasksController@delete', $task->id) }}"
class="btn btn-info">Delete</a>
</td>
```

Let's make it work!

As usual, we create the delete route first:

```
Route::get('/delete/{task}','TasksController@delete');
```

Then we create the delete action:

```
// app/controllers/TasksController.php

public function delete(Task $task)
{
        return View::make('delete', compact('task'));
}
```

After that, we build the delete form:

```
//app/views/delete.blade.php

@extends('layout')
@section('content')
<section class="header section-padding">
<div class="background"> </div>
<div class="container">
        <div class="header-text">
        <h1>Delete</h1>
        <p>
        Delete tasks page
        </p>
        </div>
</div>
</section>

<div class="container">
```

```
<section class="section-padding">
<div class="jumbotron text-center">
        <h1>Do you want to delete Task {{ $task->id }}? </h1>

        {{ Form::open(['url'=> '/delete', 'class'=>'form']) }}
        {{ Form::hidden('id', $task->id)}}

        <div class="form-group">
        {{ Form::submit('Delete Task', ['class' => 'btn btn-primary']) }}
        <a href="{{ action('TasksController@home') }}"
        class="btn btn-danger"> No </a>
        </div>

        {{ Form::close() }}

</div>
</section>
</div>
@stop
```

We also use **Form::open** and **Form::close** to create our delete form. The form is going to POST at the delete page, so the url will be /delete.

In case we change our mind and we don't want to delete the task anymore, we can put a button there to cancel:

```
        <a href="{{ action('TasksController@home') }}"
        class="btn btn-danger"> No </a>
```

Finally, we create our delete POST route to handle our form:

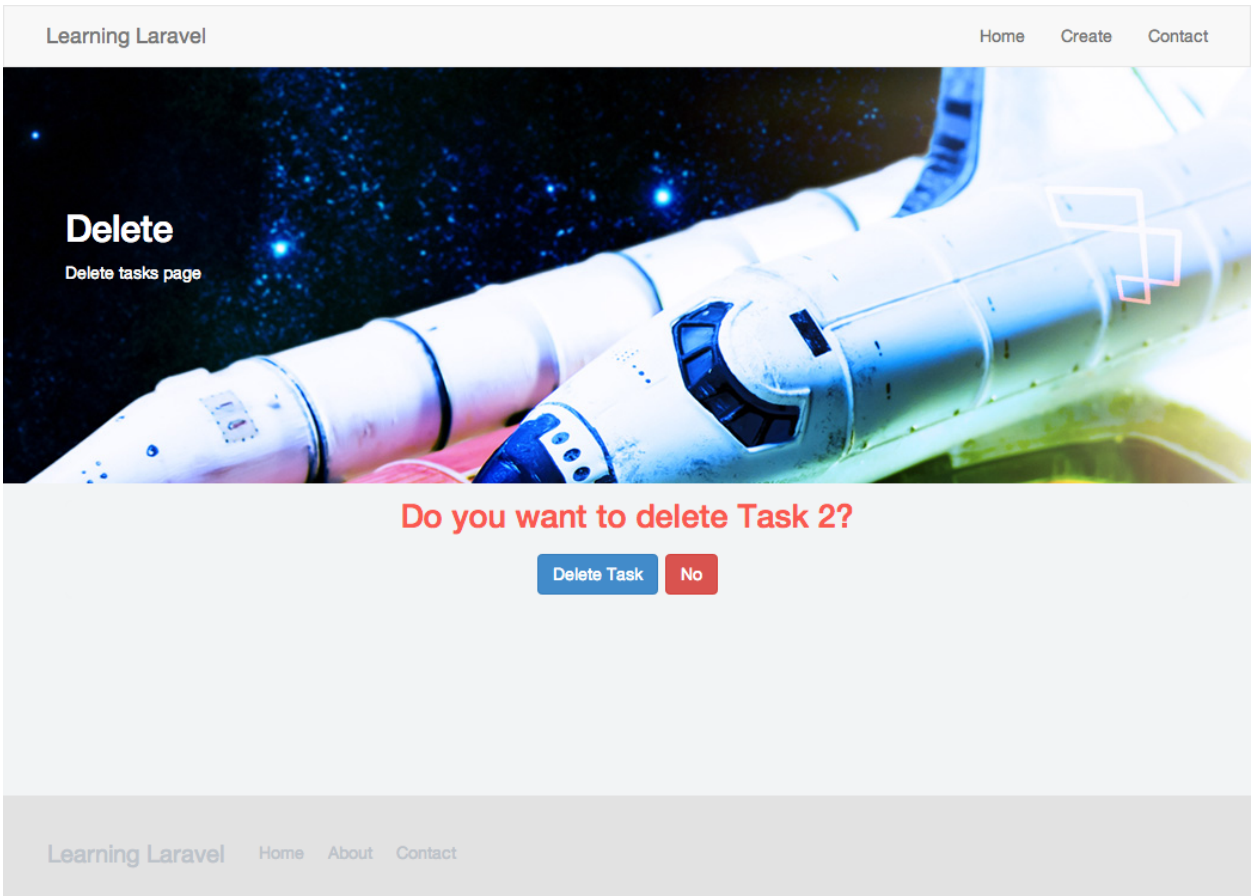
```
Route::post('/delete', 'TasksController@doDelete');
```

And our **doDelete** action:

```
public function doDelete()
{
        $task = Task::findOrFail(Input::get('id'));
        $task->delete();

        return Redirect::action('TasksController@home');
}
```

It should be easy to understand, right? We find the task using the input id. Consequently, we delete it using Eloquent ORM and return to the home page.

Well done! Let's try to delete a task, we should see a nice confirmation page:



The delete page

It's time to test our application. Why don't we go around the app, create some tasks, edit them and try to delete some of them?

I also put the source code here:

Chapter 2-1 Files[41]

You can download and view the source code if you have any problems. The code format in the book is not really looks good. Therefore, I think it's better for you to view the code in other text editors.

In the next sections, we will learn more about Validation, Dynamic Routes, Authentication and how to create an "active" navigation bar!

Load A Single Task Using Dynamic URLs

If you have used a blog or a CMS (e.g., Wordpress), you would notice that it usually has dynamic URLs.

Dynamic URLs look like this: www.yoursite.com/?=123

If you access a blog post via those dynamic URLs, the CMS will find the right posts with the provided id and display them automatically.

The good news is, we can easily do that using Laravel. More than that, we can also create any URL formats that we like for better SEO (Search Engine Optimization).

Let's make simple dynamic URLs to display our tasks!

To create a dynamic URL, we need to define its pattern. Open **routes.php** and insert:

```
1 Route::get('task/{id}', 'TasksController@show');
```

We usually create a dynamic URLs using route parameters.

Route parameters can be used to insert placeholders to our route definition. A {id} placeholder will map anything that is provided after the **task/** URL and then it will pass to the application's logic handler (such as Controller, Closures).

Sounds familiar?

We have used a {task} placeholder to edit and delete our tasks. Do you remember?

When we use task/{id}, we expect that everything after /task will be the id of our task. We then call **show** action in **TasksController.php** to handle it.

Currently, we don't have the show action yet, let's create it:

[41] http://learninglaravel.net/DL/chapter2-1.zip

```
// app/controllers/TasksController.php

public function show($id)
{
        $task = Task::find($id);

        return $task;
}
```

After getting the id from our routes, we use **Task::find** to find the right task and display it.

Now, if we visit our app using localhost:8000/task/1, we should see the task 1:

```
{"id":1,"title":"Eating breakfast","body":"Remember to buy beefsteak and eggs!","\
user_id":0,"done":1,"created_at":"2014-01-16 09:08:50","updated_at":"2014-01-26 0\
1:46:49"}
```

We can also add a validator to our routes. At the present, if we're trying to access a id that is in wrong format (e.g, task/aaa), our routes still handle it and display a blank white page. We can tell our routes to response only if our task id is an integer, otherwise it will throw an error exception:

```
Route::get('task/{id}', 'TasksController@show')->where('id', '\d+');
```

\d+ is a regular expression syntax. If you don't know about it yet, you can find a good tutorial about it here:

Regular Expression[42]

Great, you can test by accessing any {id} that is not an integer, there should be an error!

How about creating a blade template to display our single task? It's similar to how we show all the tasks, we need to pass the task to our view first using the show action:

```
public function show($id)
{
        $task = Task::find($id);

        return View::make('task', compact('task'));
}
```

If you look a little closer at the action, we find the task using the id, and then use **View::make** and **compact()** function again to pass the task to the view. Nevertheless, this time, we have only one task, so we use **task**, instead of **tasks**.

The last thing to do is create the **task.blade.php** template:

[42] http://webcheatsheet.com/php/regular_expressions.php

```
// app/views/task.blade.php

@extends('layout')
@section('content')
<section class="header section-padding">
<div class="background"> </div>
<div class="container">
        <div class="header-text">
                <h1>Learning Laravel: The Easiest Way</h1>
                <p>
                Showing a single task <br/> using route parameter!
                </p>
        </div>
</div>
</section>

<div class="container">
<section class="section-padding">
        <div class="jumbotron text-center">

        <div class="panel panel-default">
    <div class="panel-heading">
    <h1><span class="grey">Task </span> {{ $task->id }}
    </h1>
    </div>
    <table class="table">
      <thead>
        <tr>
          <th>#</th>
          <th>Title</th>
          <th>Body</th>
          <th>Finish</th>
          <th>Control</th>
        </tr>
      </thead>
      <tbody>
                <tr>
                <td>{{ $task->id }} </td>
                <td>{{ $task->title }}</td>
                <td>{{ $task->body}}</td>
                <td>{{ $task->done ? 'Yes' : 'No'}}</td>
                <td>
```

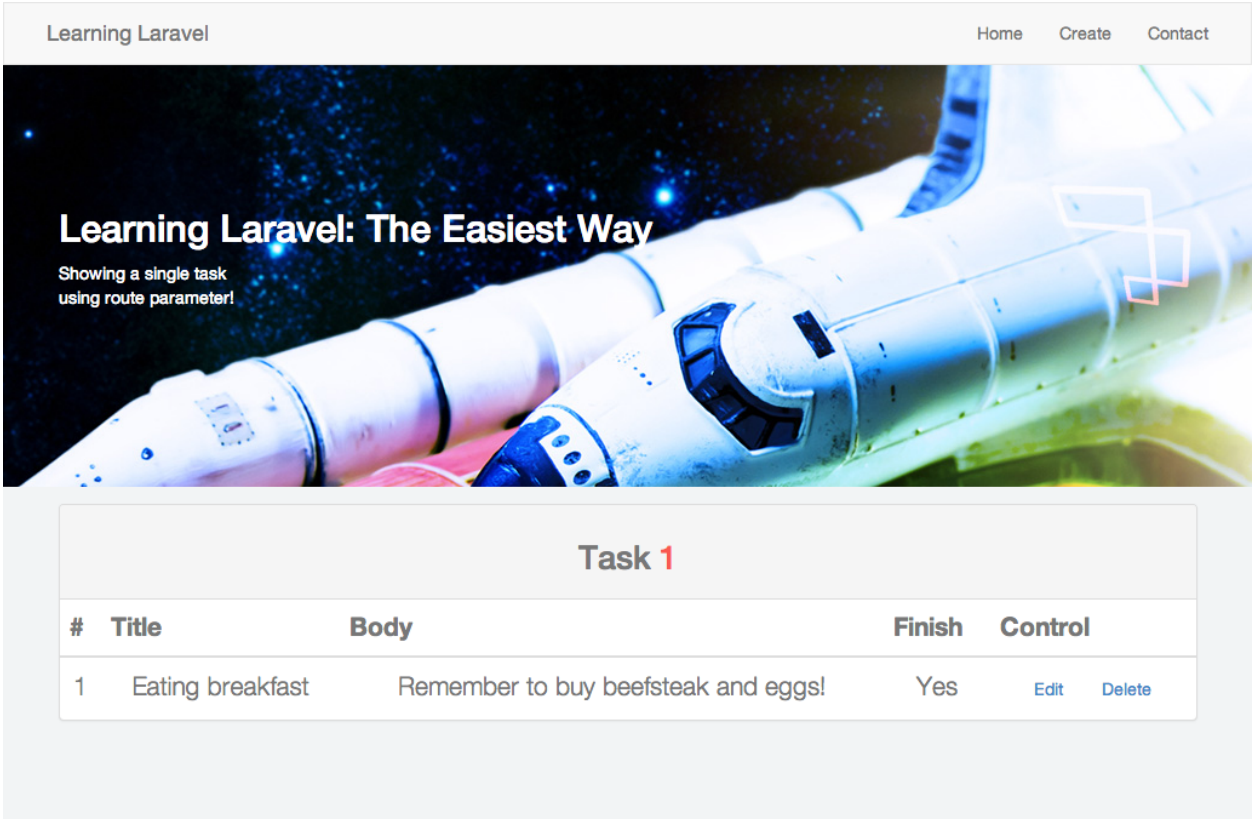
```
43                          <a href="{{ action('TasksController@edit',
44                          $task->id) }}" class="btn">Edit</a>
45                          <a href="{{ action('TasksController@delete',
46                          $task->id) }}" class="btn">Delete</a>
47                  </td>
48                  </tr>
49          </tbody>
50      </table>
51              </div>
52
53          </div>
54  </section>
55  </div>
56  @stop
```

You can copy the **home.blade.php** and change a few things to make the **task.blade.php**, because they're almost the same. We have only one task, so we should remove the if statement and the foreach loop.

Nice! Let's visit our **task/1**!

Note: e.g., localost:8000/task/1. Your path could be different, depends on your system

We have a cool single task page:



Showing task 1!

To access a single task easier, we should make links to our tasks in the home page. Let's change the task's title to make it clickable and link to its task. Open **home.blade.php**, find:

```
<td>{{ $task->title}}</td>
```

Replace with:

```
td>
<a href="{{ action('TasksController@show', $task->id) }}">
{{ $task->title }}</a>
</td>
```

We use **action()** function to automatically generate links for us. Of course, we can also use other methods to display the links, such as: task/{{ $task->id }}.

You can now be able to generate dynamic URLs and display a single task! At this time, you can use what we have learned so far and create a dynamic website! Congratulations!

Add Validation to The Forms

In the previous chapter, we have learned how to add basic validation to the forms, it's time to step one more step to learn all the validation rules, know how to throw error messages, and create our custom rules.

Adding Validation Rules to Controllers Action

The way that we add validation to the Controllers action is similar to what we've done with Closures. Open **TasksController.php** and edit the **saveCreate** action to:

```
1  public function saveCreate()
2  {
3          $data = Input::all();
4
5          $rules = array(
6                  'title'=> 'required',
7                  'body'=> 'required'
8          );
9
10         $validator = Validator::make($data, $rules);
11
12         if ($validator->passes()) {
13                 $task = new Task;
14                 $task->title = Input::get('title');
15                 $task->body = Input::get('body');
16                 $task->save();
17
18                 return Redirect::action('TasksController@home');
19         }
20
21         return Redirect::action('TasksController@create');
22 }
```

In the above code, we create a set of validation rules and put them all in an array, which is called **rules**. The rules array consist of many rules that will be used to validated.

```
        $rules = array(
                'title'=> 'required',
                'body'=> 'required'
        );
```

After that, we use **Validator::make()** method to create a new instance of the validator. The first parameter is the data that we get from the input. The second parameter is the rules that we use to validate the data.

When we've got the validator instance, we can test the result using **passes()** method. If the method returns true, it means that all the data meet the validation requirements. We're going to create a new task:

```
        if ($validator->passes()) {
                $task = new Task;
                $task->title = Input::get('title');
                $task->body = Input::get('body');
                $task->save();

                return Redirect::action('TasksController@home');
        }
```

Otherwise, we redirect users to the create tasks page again:

```
return Redirect::action('TasksController@create');
```

Great, that's what we expected. We can try to create a new task. If we leave one of the fields blank, we can't create a new task.

As you can see, we've just used the **required** rule to validate our form. Laravel is flexible, it provides us many validation rules, we will learn all of them in the next section!

Validation Rules

There are many validation rules in Laravel. You can check for all the rules in the official Docs:

Official Laravel Docs - Validation[43]

Taylor Otwell might add more or change some validation rules in the future. Therefore, make sure to check the link back if you need to use validation rules.

Here are the list of all available validation rules (from Laravel Official Docs, February 2014):

[43] http://laravel.com/docs/validation

Accepted

Active URL

After (Date)

Alpha

Alpha Dash

Alpha Numeric

Before (Date)

Between

Confirmed

Date

Date Format

Different

Digits

Digits Between

E-Mail

Exists (Database)

Image (File)

In

Integer

IP Address

Max

MIME Types

Min

Not In

Numeric

Regular Expression

Required

Required If

Required With

Required With All

Required Without

Required Without All

Same

Size

Unique (Database)

URL

We will be learning all of them and how to use them. You don't need to remember them all; just remember the ones that you think you will use for your project frequently.

Accepted

If you want to let the users accept or agree something, you can use this rule. For example, you can use it for your terms of services, your special site rules, etc. The **accepted** rule will pass if the input value is: **yes**, **on** or **1**.

Usage:

```
$rules = array(
        'term'=> 'accepted'
);
```

Active URL and URL

This will use **checkdnsrr()** PHP function to ensure that the value is a valid URL. It also check the DNS to make sure that the provided URL is within the active DNS records.

Usage:

```
$rules = array(
        'url'=> 'active_url'
);
```

If you don't want to check DNS records, you can use the **url** rule.

Usage:

```
$rules = array(
        'url'=> 'url'
);
```

After (Date)

This rule accepts time as its parameter. The rule compares the date between the field and the parameter. It will pass if the field's date occurs after the parameter's date.

Usage:

```
$rules = array(
        'date'=> 'after:03/02/14'
);
```

Before (Date)

This rule is the opposite of the **after** rule. It also accepts time as its parameter. The rule compares the date between the field and the parameter. It will pass if the field's date occurs before the parameter's date.

Usage:

```
$rules = array(
        'date'=> 'before:03/02/14'
);
```

Alpha, Alpha Dash and Alpha Numeric

The **alpha** rule is used to ensure that the provided field consists entirely alphabetic characters.

The **alpha dash** rule is used to ensure that the provided field consists entirely alphabetic characters and - (dashes) or _ (underscores).

The **alpha numeric** rule is used to ensure that the provided field consists entirely alphabetic characters and numeric characters.

Usage:

```
$rules = array(
        'username'=> 'alpha'
);
```

Between

This rule is used to validate a size between two parameters. The parameters can be strings (compare the length of string), numerics and files (compare the size of file).

Usage:

```
$rules = array(
        'age'=> 'between:13,18'
);
```

Confirmed

This rule is used to ensure that another field (appended with _confirmation) matches the current field. This rule is usually used for validating the password field. For example, the rule will pass if a password field matches the password_confirmation field.

Usage:

```
$rules = array(
        'password'=> 'confirm'
);
```

Date

To ensure that a date is valid, we can use this rule.

Usage:

```
$rules = array(
        'birthdate'=> 'date'
);
```

Date Format

To ensure that the provided date matches the parameter's format. We can use date_format rule. To know more information about how to use a date format, you can visit:

Date Format[44]

Usage:

```
$rules = array(
        'birthdate'=> 'date_format:m/d/y'
);
```

Different and Same

To ensure that the value of the givien field is different than the rule parameter's field, we can use different rule.

Usage:

```
$rules = array(
        'title'=> 'different:subtitle'
);
```

The same rule is the direct opposite of the different rule. We can use it to ensure that the current field's value is the same as another field.

Usage:

[44] http://www.php.net/manual/en/datetime.formats.date.php

```
$rules = array(
        'last_name'=> 'same:sur_name'
);
```

Digits and Digits Between

The digit rule can be used to ensure that, the given field contains numeric values, and the length of the value must be the same with the provided parameter.

Usage:

```
$rules = array(
        'tax_id'=> 'digits:5'
);
```

The digits_between rule can be used to ensure that, the given field contains numeric values, and the length of the value must be between the provided min max parameter.

Usage:

```
$rules = array(
        'age'=> 'digits_between:1,3'
);
```

E-Mail

This rule is used to ensure that the given value is a valid email address.

Usage:

```
$rules = array(
        'email'=> 'email'
);
```

Exists (Database)

This rule can be used to check if the value being validated exists in a database table. The rule is very useful for registration forms, subscribe forms or similar forms.

Usage:

1- Check if the user exists in the users table:

```
$rules = array(
        'user'=> 'exists:users'
);
```

2- We can add more conditions by providing additional parameters (it works just like where clauses):

```
$rules = array(
        'user'=> 'exists:users, role, moderator'
);
```

3- We can also check for a NULL database value:

```
$rules = array(
        'user'=> 'exists:users, role, moderator, deleted_at, NULL'
);
```

Image (File)

This rule can be used to ensure that the uploaded file is an image.

Usage:

```
$rules = array(
        'avatar'=> 'image'
);
```

In and Not In

The **in** rule can be used to ensure that the value being validate is included in the given list of values.

Usage:

```
$rules = array(
        'color'=> 'in:black,blue,red'
);
```

The not_in rule is the opposite of the in rule.

Usage:

```
$rules = array(
        'color'=> 'not_in:black,blue,red'
);
```

Integer

This rule can be used to ensure that the value of the the given field is an integer.

Usage:

```
$rules = array(
        'number'=> 'integer'
);
```

IP

This **ip** rule can be used to ensure that the value of the the given field is an IP address.

Usage:

```
$rules = array(
        'ip_address'=> 'ip'
);
```

Max and Min

The **max** rule is used to ensure that the size of the field being validated is less than a maximum provided parameter. The parameters can be strings (compare the length of string), numerics and files (compare the size of file).

Usage:

```
$rules = array(
        'id'=> 'max:3'
);
```

The **min** rule is the opposite of the max rule. The rule is used to ensure that the size of the field being validated is greater than a minimum provided parameter. The parameters can be strings (compare the length of string), numerics and files (compare the size of file).

```
$rules = array(
        'id'=> 'min:3'
);
```

MIME Types

The **mimes** rule is used to ensure that the mime type of the uploaded file matches one of the provided parameters.

Usage:

```
$rules = array(
        'avatar'=> 'mimes:jpg,png,gif'
);
```

Numeric

This **numeric** rule is used to check if the given filed contains a numeric value.

Usage:

```
$rules = array(
        'student_id'=> 'numeric'
);
```

Regular Expression

The **regex** rule can be used to provide a custom regular expression to validate the field.

Usage:

```
$rules = array(
        'username'=> 'regex:[a-z]'
);
```

Required

The **required** rule is very familiar, right? We have used it many times. This rule can be used to ensure that the field being validated must present in the input data.

Usage:

```
$rules = array(
        'username'=> 'required'
);
```

Required If

The **required_if** rule can be used to ensure that, the current field is required only if the first parameter (represents a field) of the rule matches the second parameter's value.

Usage:

```
$rules = array(
        'special_discount'=> 'required_if:username,jack'
);
```

Required With and Required Without

The **required_with** rule can be used to ensure that, the current field is required only if any other fields are also present.

Usage:

```
$rules = array(
        'age'=> 'required_with:weight,height'
);
```

The **required_without** rule is the opposite of the required_with rule, the current field is required only if any of other fields are not present.

Usage:

```
$rules = array(
        'age'=> 'required_without:weight,height'
);
```

Required All and Required Without All

The **required_with_all** rule can be used to ensure that, the current field is required only if all other fields are also present.

Usage:

```
$rules = array(
        'age'=> 'required_with_all:weight,height'
);
```

The **required_without_all** rule is the opposite of the required_with_all rule, the current field is required only if all other fields are not present.

Usage:

```
$rules = array(
        'age'=> 'required_without_all:weight,height'
);
```

Size

You can guess it by the name. The **size** rule can be used to ensure that the current field's size matches the size of the parameter. For strings, the size is the number of characters. For numeric, the size corresponds to a given integer value. For files, the parameter refers to the file size in kilobytes.

Usage:

```
$rules = array(
        'shoe_size'=> 'size:8'
);
```

Unique (Database)

To ensure that the field being validated is unique and not present in a defined database table, we use the **unique** rule. For example, We often use this to check for username in the registration form.

Usage:

```
$rules = array(
        'username'=> 'unique:users'
);
```

We can also add optional parameters to ignore a number of IDs:

```
$rules = array(
        'username'=> 'unique:users, email_address, 1, 7, 12'
);
```

Or we can even add additional where clauses to the query. For example, if we want to "unique check" only rows with student_id of 1, we can use:

```
$rules = array(
        'username'=> 'unique:users, email_address, NULL, id, student_id, 7'
);
```

Phew... Many rules, right?

All of these rules are very useful! That's why I love Laravel. In the next section, we will learn how to display error messages.

Chapter 3 - Building A Product Management System (TODO)

Coming Soon

I'm writing, please wait. If you have any ideas or want to learn something, don't hesitate to send me a message. If this book have helped you in anyway, then I would really appreciate if you would share the URL to the book with your friends. It's at www.learninglaravel.net[45] :D

[45] http://learninglaravel.net

PART 3: AN ALTERNATIVE LARAVEL DOCUMENTATION

This part will be an alternative documentation. You will learn how to install Laravel on different systems, read this part as a cheat sheet, and know everything about Laravel framework here.

A Guide to Install Laravel 4

I will show you how to install Laravel 4 on Mac and Windows from scratch. It means that you will know how to install PHP, Mysql, Mcrypt, Composer and everything to get your app running!

Please note that there are many ways to install PHP, Composer, Mcrypt, etc. I will show you the basic ones first, and then I'll update other methods later. If you love to do it in a different ways, go ahead and do it! You will learn a lot by doing it yourself.

What We Need to Install Laravel 4?

Laravel 4 requires the following things to run:

1. PHP >= 5.3.7 (version 5.3.7 or newer)
2. MCrypt PHP Extension
3. Composer (optional, but you will need it to build Laravel 4 applications)
4. Openssl should be enabled in php.ini.

Installing Laravel on Mac OS X

1- Installing Xcode and activate Xcode Command Line Tools

In order to install Laravel on Mac, you will need to download Xcode. You can download it at the link below for free:

Apple Xcode[46]

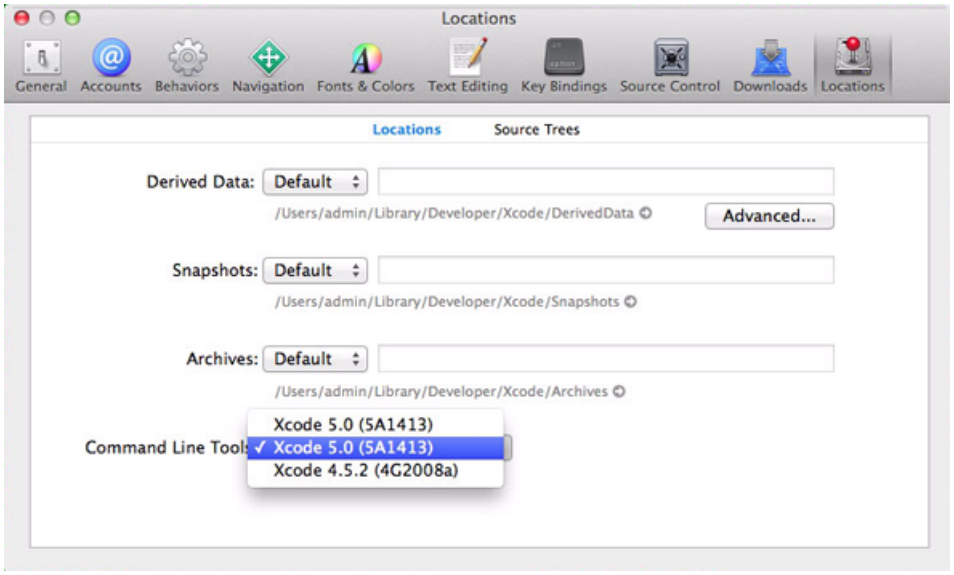
or

Apple Xcode on Apple app store[47]

After that, we go to Xcode -> Preference -> Locations button -> Select '**Command Line Tools**' -> Make sure to choose '**Xcode 5.0**' -> Download and install the **Command Line Tool** from Apple Developer website.

[46] https://developer.apple.com/xcode

[47] http://itunes.apple.com/us/app/xcode/id497799835?ls=1&mt=12



Style Vintage Theme

A tip for older Mac OS X

You should upgrade to Mavericks 10.9. If for some reasons you don't want to upgrade, and you don't see the interface above, you can still install Command Line Tools by going to Xcode -> Preference -> Downloads -> Install '**Command Line Tools**'

2- Installing PHP 5.4

We need to install a right PHP version to run Laravel. To do that, we have to use Terminal app, or some other terminal emulator applications.

What is Terminal?

Terminal is terminal emulator provides an environment for Unix shells, which allows the user to interact with the operating system through the command line interface. On Mac, you can find it in Applications -> Utilities -> Terminal.

Open Terminal and type in the below code to know which PHP version you're using:

```
1 php -v
```

Cool, you will see something like this:

```
PHP 5.4.17 (cli) (built: Sep 18 2013 14:31:13)
Copyright (c) 1997-2013 The PHP Group
```

It means that you currently have PHP 5.4.17 on your machine. If you have PHP version 5.4 or newer, then skip this section.

Using OS X Mavericks?

Mac OS X 10.9 ships with PHP 5.4.17 out of the box. So you can skip this part if you're using Max OS X 10.9. However, just run php -v to double check that you're using a correct PHP version.

If you see that you have an older PHP version or you don't have PHP, then let's install it. Type this line or copy and paste in into the Terminal:

```
curl -s http://php-osx.liip.ch/install.sh | bash -s 5.4
```

Wait for while, **type y and hit Enter** if it asks you, then you paste the following line:

```
export PATH=/usr/local/php5/bin:$PATH
```

Done! if we check using php -v again, it will show that you have PHP 5.4.xx.

3- Installing Mcrypt PHP Extension

We're going to install Mcrypt PHP Extension now.

What is Mcrypt?

Mcrypt is a file encryption method using secure techniques to exchange data.

Open Terminal, change directory (cd) to the home account and make a directory that you will work in, call it mcrypt. You can do these things by typing into Terminal (line by line):

```
cd ~
mkdir mcrypt
cd mcrypt
```

This will make a folder called mcrypt in my home account folder, which is Users/~YourUsername. For example, I can find the mycrypt folder on my Mac at: Mac/Users/JV.

Get libmcrypt 2.5.8 from Sourceforge:

libmcrypt 2.5.8 from Sourceforge[48]

Get the php code in a tar.gz or .bz2 format at:

Choose and download PHP code[49]

Make sure to download a correct PHP version (same with your OS).

In case you don't remember, you can check your PHP version using the command line:

```
1 php -v
```

Move both of these files that you downloaded into your working directory (mcrypt) and go back to Terminal:

```
1 cd ~/mcrypt
```

Expand both files via the command line (or just double click them in the Finder and skip this part):

```
1 tar -zxvf libmcrypt-2.5.8.tar.gz
2 tar -zxvf php-5.4.17.tar.gz
```

Remove the compressed archives:

```
1 rm *.gz
```

Change directory into libmcrypt:

```
1 cd libmcrypt-2.5.8
```

Libmcrypt needs to be configured, enter:

```
1 ./configure
2 make
3 sudo make install
```

When you type sudo, usually it will ask for a password, enter your system password and move on.

You now have libmcrypt configured and libraries now installed, it's time to make the mcrypt extension. Enter:

[48] http://sourceforge.net/projects/mcrypt/files/Libmcrypt/2.5.8/libmcrypt-2.5.8.tar.gz/download

[49] http://php.net/releases/index.php

```
cd ../php-5.4.17/ext/mcrypt/
/usr/bin/phpize
```

The output should be:

```
Configuring for:
PHP Api Version:         20100412
Zend Module Api No:      20100525
Zend Extension Api No:   220100525
```

Autoconf Errors

If you see "Cannot find autoconf. Please check your autoconf installation..." error occurs after you try the following compile of mcrypt, then autoconf is not installed. Install it using the following guide, otherwise you can skip it.

Go to Terminal, type:

```
cd ~/mcrypt
curl -O http://ftp.gnu.org/gnu/autoconf/autoconf-latest.tar.gz
tar xvfz autoconf-latest.tar.gz
cd autoconf-2.69/
./configure
make
sudo make install
```

After installing Autoconf, you need to go back to the folder mcrypt/php-5.4.17/ext/mcrypt, enter:

```
cd ~/mcrypt/php-5.4.17/ext/mcrypt
/usr/bin/phpize
```

You should see the output like this, without "Cannot find autoconf..." error:

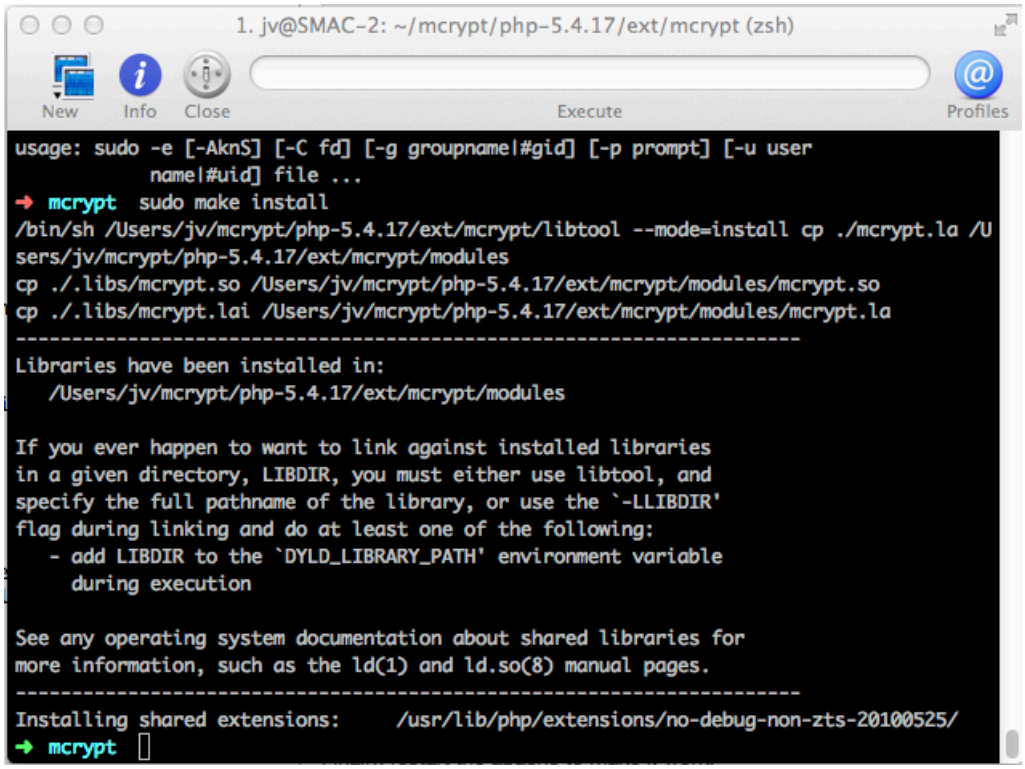
```
Configuring for:
PHP Api Version:         20100412
Zend Module Api No:      20100525
Zend Extension Api No:   220100525
```

Good, make sure you're still in the mcrypt folder, now enter:

```
./configure
make
sudo make install
```

Done! You shoud see:

```
Installing shared extensions:/usr/lib/php/extensions/no-debug-non-zts-20100525
```



The output

Last step, You need to enable mcrypt.so PHP extension. Open /etc/php.ini and add the line below at the end of the file:

```
extension=mcrypt.so
```

If there is no php.ini file, then you need to make one from php.ini.default in the same location using Terminal:

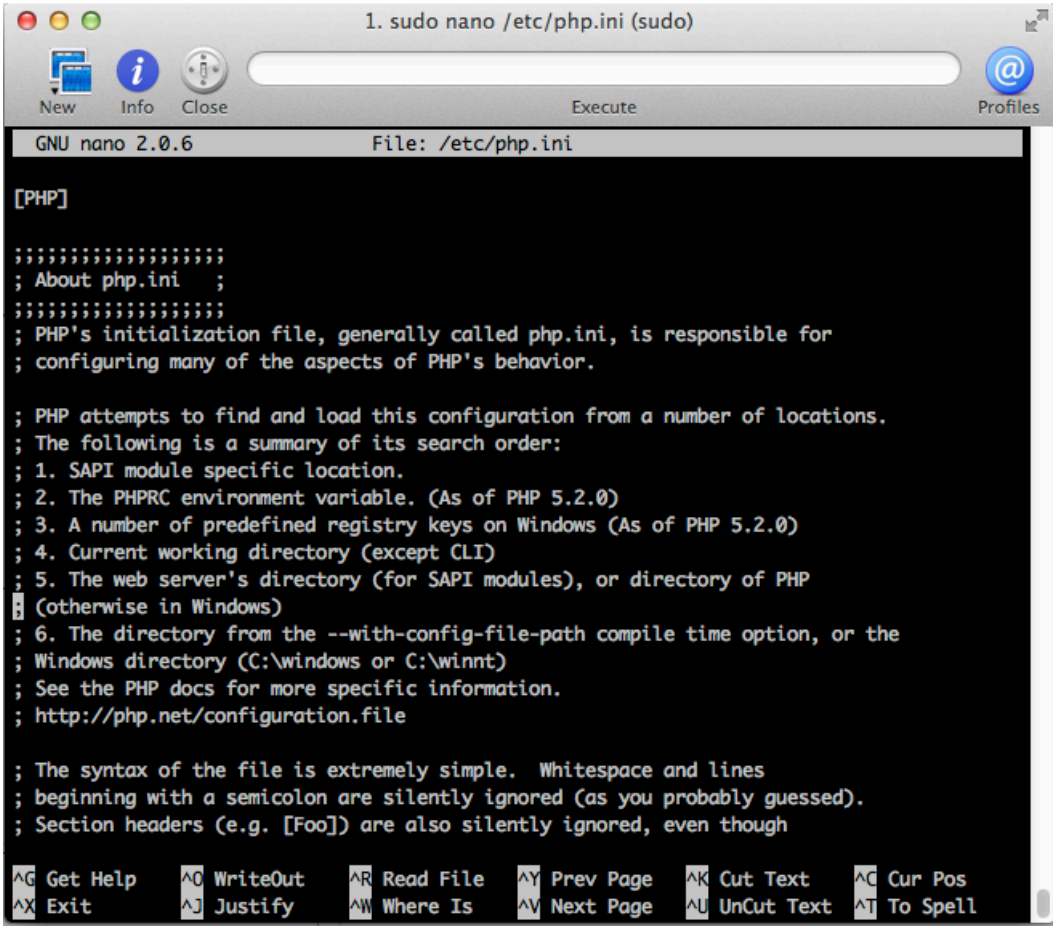
```
sudo cp /etc/php.ini.default /etc/php.ini
sudo chmod u+w  /etc/php.ini
```

Then open the php.ini file by typing:

```
sudo nano /etc/php.ini
```

or

```
sudo vi /etc/php.ini
```



View php.ini in nano

Make sure that you have the line below at the end of your php.ini:

```
extension=mcrypt.so
```

A note about editing php.ini

You can use a normal text editor to edit php.ini, but try to use nano or vi. It's very easy to learn.

Finally, restarting the Apache service to make it work:

```
sudo apachectl restart
```

Congrats! You have installed Mcrypt PHP Extension!

4- Installing Composer

Open Terminal and then execute this command:

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

If you see error: The detect_unicode setting must be disabled. Use this command instead:

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

This generates composer.phar (a PHP executable). Then run it:

```
php composer.phar
```

Final step, execute this command to easily access Composer everywhere on your system:

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

Well done! Now you can install Laravel!

5- Installing Laravel

When you have Composer configured, you can easily install and start using Laravel. We will install our application on desktop, so go there by using this command:

```
cd desktop
```

Cool, now you're at your desktop, you just need execute one command in the command line to create a Laravel application,replacing **learningLaravel** with the name of your project:

```
composer create-project laravel/laravel learningLaravel --prefer-dist
```

After executing this command, Composer will download all components of Laravel, and put them together into your application folder called learningLaravel. You will see something like this when it finishes:

```
Writing lock file
Generating autoload files
Generating optimized class loader
Application key [qAAhzWNNhYnzAsVc0NzV19ocWnZICiBa] set successfully.
```

You can start to develop application right now!

6- Start your first app

When you have your app, you need to "start it" using Artisan - a Laravel's command line interface. It's easy, first, navigate to your app folder:

```
cd learningLaravel
```

and then execute:

```
php artisan serve
```

Congrats! You will see this output:

```
 Laravel development server started on http://localhost:8000
```

Open your web browser, and go to http://localhost:8000, enjoy your first site!

Your site is running on localhost:8000

Having some errors?

Don't hesitate to send me a message! I'll be with you to solve the problem! There is an easier method to set things up by using MAMP or XAMPP. But you should try to install it manually like the above method, this way you can learn more.

Installing Laravel on Windows 7 + Windows 8

Install Laravel on Windows is a piece of cake! You can install it easily using XAMPP or WAMP. So what is XAMPP or WAMP? Well, they're a distribution that includes an Apache 2 web server, integrated with the latest builds of MySQL, PHP and Perl. You can install them in just a few clicks.

I will show you how to install Laravel using XAMPP (using MAMP is very similar).

1- Installing XAMPP

Now, go to the website below, download XAMPP and install it:

XAMPP official website[50]

You should choose installer, and install it just like installing a normal application. There is a documentation about how to install it here:

How to install XAMPP[51]

The cool thing is, Mcrypt comes by default with XAMPP or WAMP, so you don't have to mess around with it :D

There is also WAMP and XAMPP for Mac OS!

The great thing is, WAMP and XAMPP is also available for Mac OS. On Mac, WAMP is called MAMP! Google it and use that method if you like.

2- Enable OpenSSL

Go over to php.ini (located in C:\xampp\php) and open it with a text editor. Find:

```
1 ;extension=php_openssl.dll
```

And remove the ";", then save the file. So it should look like this:

[50] http://www.apachefriends.org/en/xampp-windows.html
[51] http://www.apachefriends.org/en/xampp-windows.html#522

```
extension=php_openssl.dll
```

Good, now restart the Apache.

3- Installing Composer

Now go to Composer site, download and install Composer-Setup.exe:

Composer official site[52]

4- Make sure that you have httpd-vhosts.conf

We use httpd-vhosts.conf to create our virtual host. So let's go to C:\xampp\apache\conf, and then open httpd.conf file with your text editor. Search for:

```
# Virtual hosts
Includes "conf/extra/httpd-vhosts.conf"
```

It should look like that, if you see something like:

```
#Includes "conf/extra/httpd-vhosts.conf"
```

Let's remove the # sign.

5- Installing Git Bash

To execute commands, you can use any tool, but I recommend you Git Bash. A very popular program that is under active maintenance. It creates a prompt on Windows, which is similar to a UNIX-like system. To install it, go to:

Git Bash[53]

Download and install Git-1.8.4-preview20130916.exe (you can download a newer version, at the time of writing, the current version is 1.8.4). When it asks something, you should accept the defaults.

5- Installing Laravel

Now you can install Laravel. First, go to C:/xampp/htdocs, create a folder called **learninglaravel** (or any name that you like, just make sure to replace it at the codes below)

Good, now click the Windows or Start icon -> in the Programs list, open the Git folder -> open Git Bash.

Nice, you can type some commands there to install Laravel. First, we go to the folder that we just created, type this into Git Bash:

[52] http://getcomposer.org/doc/00-intro.md#installation-windows

[53] http://code.google.com/p/msysgit/downloads/list?q=label:Featured

```
cd C:/xampp/htdocs
```

and then use ls command to see what's inside, type:

```
 ls
```

You should see your folder there in the output, like this:

```
FF learningLaravel
```

Cool! Now go to that folder by using cd command:

```
cd learningLaravel
```

Finally, you can install Laravel into the folder by typing:

```
composer create-project laravel/laravel --prefer-dist
```

Composer will be going to download and install Laravel into that folder. After it's done, you will see something like this:

```
Writing lock file
Generating autoload files
Generating optimized class loader
Application key [qAAhzWNNhYnzAsVc0NzV19ocWnZICiBa] set successfully.
```

Good job, you have just installed Laravel. You can open your web browser and go to your site at this address:

http://localhost/learningLaravel/public[54]

If there are some errors, don't worry, we will fix it soon.

Install Laravel in an easier way?

Actually, you can install and create the learningLaravel folder at a time by using this command: "composer create-project laravel/laravel learningLaravel –prefer-dist". If you understand what I say, well done. If you don't, don't worry, you just need more time to get familiar with it. You can also install Laravel using the new Laravel Installer method if you like, there is a tutorial about it in this book.

6- Edit httpd-vhosts.conf to access Laravel

Now we have to edit the **httpd-vhosts.conf** file to access our site. Go to "C:\xampp\apache\conf\extra", open and edit the file. Copy and paste these lines into the end of the file:

[54] http://localhost/learningLaravel/public

```
<VirtualHost *80>
        DocumentRoot "C:/xampp/htdocs/learningLaravel/public"
        ServerName learningLaravel.dev
</VirtualHost>
```

What we're doing here is putting the path to our application folder (**learningLaravel**) into DocumentRoot. And set the ServerName to learningLaravel.dev. So we just only need to type **learninglaravel.dev** into our web browser to access it!

Now you should restart the Apache using XAMPP Control Panel. Opening up the Xampp control panel, clicking 'Stop' (next to 'Apache'), waiting for it to stop, then clicking 'Start'.

7- Last step! Edit the hosts file

Go to **C:/Windows/System32/Drivers/etc** and edit the **hosts** file. The **hosts** file require Administrator permission. So you need to open your notepad as Administrator (Ctrl -> Right Click -> Open As Administrator), and then open the hosts file as Administrator.

Ok, put "127.0.0.1 learninglaravel.dev" into the hosts file, below this line:

```
127.0.0.1   localhost
```

You should have something look like this:

```
127.0.0.1   localhost
127.0.0.1   learninglaravel.dev
```

Save the file and... Congrats! You can now access your site using this link:

http://learninglaravel.dev[55]

You should see this screen:

[55] http://learninglaravel.dev

Your site is running on localhost:8000

8- Extra step

If you don't see the screen above, instead it shows many project files like normal localhost does. Follow these steps below to fix it:

First, you need to make sure that **rewrite_module** and **vhost_alias_module modules** are enabled. The following lines should be uncommented in **xampp/apache/conf/httpd.conf** and in **xampp/apache/conf/original/httpd.conf**, make sure to remove the # sign:

```
1  LoadModule rewrite_module modules/mod_rewrite.so
2
3  LoadModule vhost_alias_module modules/mod_vhost_alias.so
4
5  Include conf/extra/httpd-vhosts.conf
```

Update your **httpd-vhosts.conf** with the following:

```
<VirtualHost *:80>
    DocumentRoot "c:/xampp/htdocs/laravel/public"
    ServerName learningLaravel.dev
    ServerAlias www.learningLaravel.dev
</VirtualHost>
```

A New Faster Way to Install Laravel 4

Recently, Taylor Otwell has just released a new method to install Laravel 4 easier and faster! You can now install Laravel using Laravel Installer.

This tutorial is for Mac Os X

This method is tested, and it's working on Mac OS. Laravel Installer is still new, so in case you can't use it, don't worry, please wait for a few days. You can still install Laravel using Composer as always. If you're using Windows, go ahead and try to install Laravel 4 using this method, it's pretty similar.

A little note: just remind that you must have PHP and Composer installed on your system to install Laravel.

First, you need to download the Laravel installer PHAR archive at the link below:

Download Laravel Installer[56]

Put it in your working folder, or your desktop. I put it on my desktop. If you're doing the same, then navigate to your desktop:

```
cd desktop
```

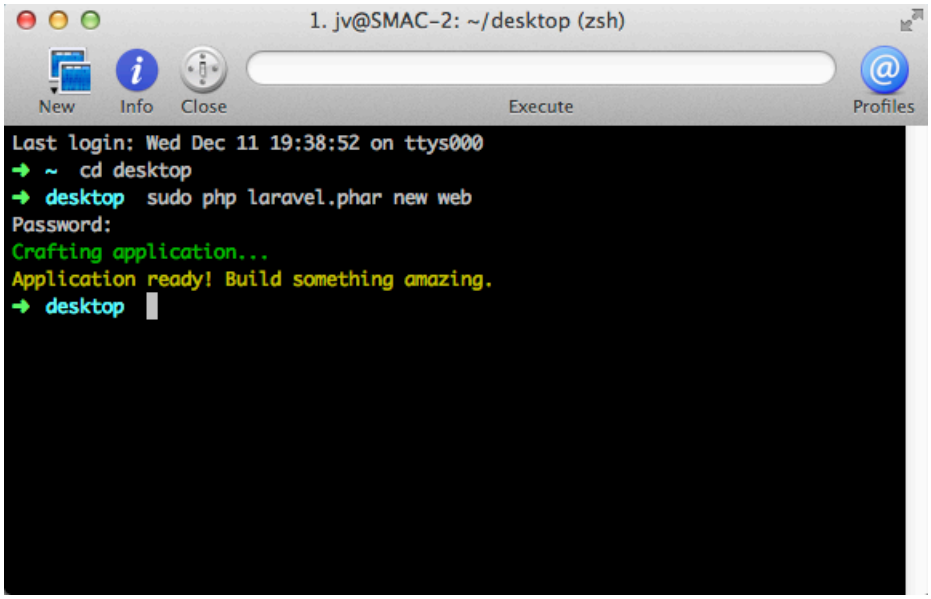
Good, you're in the desktop. Now just type this single line to install Laravel in no time:

```
sudo php laravel.phar new web
```

If it asks for password, enter your system password.

Great! You just create a directory named web containing a fresh Laravel installation with all dependencies installed.

[56] http://laravel.com/laravel.phar



Install Laravel using the new Laravel Installer

This method is much faster than using Composer!

For convenience, you can rename the laravel.phar file to laravel and move it to /usr/local/bin. After that, you can create new Laravel apps everywhere on your system. To move laravel.phar to /usr/local/bin, execute this command:

```
1 sudo mv laravel.phar /usr/local/bin/laravel
```

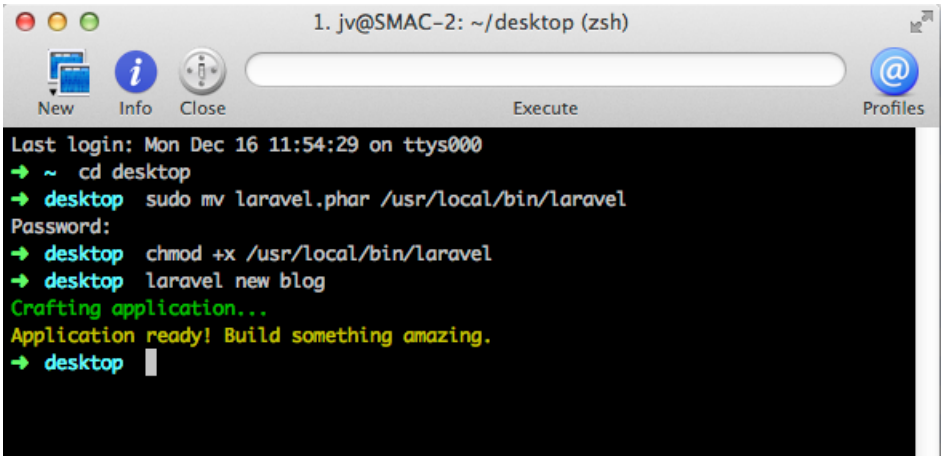
As you see, I use mv command to move laravel.phar on my desktop to /usr/local/bin and rename the file to laravel. Ok, the last trick is, you need to set permissions for the file, then we can use it:

```
1 chmod +x /usr/local/bin/laravel
```

Well done! Now, everytime you want to create a new Laravel app. You just navigate to a place (for example: desktop), and execute this command:

```
1 laravel new web
```

So cool! Right? Here is the output:



Install Laravel using new method

I'm writing this chapter!

I'm writing, please wait. If you have any ideas or want to learn something, don't hesitate to send me a message. If this book have helped you in anyway, then I would really appreciate if you would share the URL to the book with your friends. It's at www.leanpub.com/learninglaravel[57] or www.learninglaravel.net[58] :D

[57] http://leanpub.com/learninglaravel
[58] http://learninglaravel.net

PART 4: LARAVEL CHEAT SHEET

Want a Laravel Cheat Sheet? Here it is! You can use this chapter as your cheat sheet! More information and descriptions will be updated later!

This cheat sheet is based on Jesse Obrien cheat sheet and Laravel Docs. You can view the Jesse's cheat sheet here:

http://cheats.jesse-obrien.ca/[59]

Artisan

```
php artisan --help OR -h
php artisan --quiet OR -q
php artisan --version OR -V
php artisan --no-interaction OR -n
php artisan --ansi
php artisan --no-ansi
php artisan --env

php artisan changes
php artisan clear-compiled
php artisan down
php artisan dump-autoload
php artisan env
php artisan help
php artisan list
php artisan migrate
php artisan optimize
php artisan routes
php artisan serve
php artisan tinker
php artisan up
php artisan workbench
```

[59] http://cheats.jesse-obrien.ca/

```
23
24 php artisan asset:publish [--bench["vendor/package"]] [--path["..."]] [package]
25 php artisan auth:reminders
26 php artisan cache:clear
27 php artisan command:make name [--command["..."]] [--path["..."]] [--namespace[=\
28 "..."]]
29 php artisan config:publish
30 php artisan controller:make [--bench="vendor/package"]
31 php artisan db:seed [--class["..."]] [--database["..."]]
32 php artisan key:generate
33 php artisan migrate [--bench="vendor/package"] [--database["..."]] [--path["...\
34 "]] [--package["..."]] [--pretend] [--seed]
35 php artisan migrate:install [--database["..."]]
36 php artisan migrate:make name [--bench="vendor/package"] [--create] [--package["\
37 ..."]] [--path["..."]] [--table["..."]]
38 php artisan migrate:refresh [--database["..."]] [--seed]
39 php artisan migrate:reset [--database["..."]] [--pretend]
40 php artisan migrate:rollback [--database["..."]] [--pretend]
41 php artisan queue:listen [--queue["..."]] [--delay["..."]] [--memory["..."]] [\
42 --timeout["..."]] [connection]
43 php artisan queue:subscribe [--type["..."]] queue url
44 php artisan queue:work [--queue["..."]] [--delay["..."]] [--memory["..."]] [--\
45 sleep] [connection]
46 php artisan session:table
47 php artisan view:publish [--path["..."]] package
```

Composer

```
1 composer create-project laravel/laravel folder_name
2 composer install
3 composer update
4 composer dump-autoload [--optimize]
5 composer self-update
```

Routing

```
Route::get('foo', function(){});
Route::get('foo', 'ControllerName@function');
Route::controller('foo', 'FooController');
```

Triggering Errors

```
App::abort(404);
App::missing(function($exception){});
throw new NotFoundHttpException;
```

Route Parameters

```
Route::get('foo/{bar}', function($bar){});
Route::get('foo/{bar?}', function($bar = 'bar'){});
```

HTTP Verbs

```
Route::any('foo', function(){});
Route::post('foo', function(){});
Route::put('foo', function(){});
Route::patch('foo', function(){});
Route::delete('foo', function(){});
Route::resource('foo', 'FooController');
```

Secure Routes

```
Route::get('foo', array('https', function(){}));
Route Constraints
Route::get('foo/{bar}', function($bar){})
        ->where('bar', '[0-9]+');
Route::get('foo/{bar}/{baz}', function($bar, $baz){})
        ->where(array('bar' => '[0-9]+', 'baz' => '[A-Za-z]'))
```

Filters

```
Route::filter('auth', function(){});

Route::filter('foo', 'FooFilter');
Route::get('foo', array('before' => 'auth', function(){}));

Route::get('foo', array('before' => 'auth', function(){}));
Route::group(array('before' => 'auth'), function(){});

Route::when('foo/*', 'foo');

Route::when('foo/*', 'foo', array('post'));
```

Named Routes

```
Route::currentRouteName();
Route::get('foo/bar', array('as' => 'foobar', function(){}));
```

Route Prefixing

```
Route::group(array('prefix' => 'foo'), function(){})
```

Sub-Domain Routing

```
Route::group(array('domain' => '{sub}.example.com'), function(){});
```

URLs

```
URL::full();
URL::current();
URL::previous();
URL::to('foo/bar', $parameters, $secure);
URL::action('FooController@method', $parameters, $absolute);
URL::route('foo', $parameters, $absolute);
URL::secure('foo/bar', $parameters);
URL::asset('css/foo.css', $secure);
URL::secureAsset('css/foo.css');
URL::isValidUrl('http://example.com');
URL::getRequest();
URL::setRequest($request);
URL::getGenerator();
URL::setGenerator($generator);
```

Events

```
Event::fire('foo.bar', array($bar));
Event::listen('foo.bar', function($bar){});
Event::listen('foo.*', function($bar){});
Event::listen('foo.bar', 'FooHandler', 10);
Event::listen('foo.bar', 'BarHandler', 5);
Event::listen('foor.bar', function($event){ return false; });
Event::queue('foo', array($bar));
Event::flusher('foo', function($bar){});
Event::flush('foo');
Event::subscribe(new FooEventHandler);
```

Database

```
DB::connection('connection_name');
DB::statement('drop table users');
DB::listen(function($sql, $bindings, $time){ code_here; });
DB::transaction(function(){ transaction_code_here; });

DB::table('users')->remember($time)->get();

DB::raw('sql expression here');
```

Selects

```
DB::table('name')->get();
DB::table('name')->distinct()->get();
DB::table('name')->select('column as column_alias')->get();
DB::table('name')->where('name', '=', 'John')->get();
DB::table('name')->whereBetween('column', array(1, 100))->get();
DB::table('name')->whereIn('column', array(1, 2, 3))->get();
DB::table('name')->whereNotIn('column', array(1, 2, 3))->get();
DB::table('name')->whereNull('column')->get();
DB::table('name')->whereNotNull('column')->get();
DB::table('name')->groupBy('column')->get();
DB::table('name')->orderBy('column')->get();
DB::table('name')->having('count', '>', 100)->get();
DB::table('name')->skip(10)->take(5)->get();
DB::table('name')->first();
DB::table('name')->pluck('column');
```

```php
DB::table('name')->lists('column');

DB::table('name')->join('table', 'name.id', '=', 'table.id')
    ->select('name.id', 'table.email');
```

Inserts, Updates, Deletes

```php
DB::table('name')->insert(array('name' => 'John', 'email' => 'john@example.com'));
DB::table('name')->insertGetId(array('name' => 'John', 'email' => 'john@example.c\
om'));

DB::table('name')->insert(
        array('name' => 'John', 'email' => 'john@example.com')
        array('name' => 'James', 'email' => 'james@example.com')
);

DB::table('name')->where('name', '=', 'John')
        ->update(array('email' => 'john@example2.com'));

DB::table('name')->delete();

DB::table('name')->where('id', '>', '10')->delete();
DB::table('name')->truncate();
```

Aggregates

```php
DB::table('name')->count();
DB::table('name')->max('column');
DB::table('name')->min('column');
DB::table('name')->avg('column');
DB::table('name')->sum('column');
DB::table('name')->increment('column');
DB::table('name')->increment('column', $amount);
DB::table('name')->decrement('column');
DB::table('name')->decrement('column', $amount);
```

Raw Expressions

```
DB::select('select * from users where id = ?', array('value'));
DB::table('name')->select(DB::raw('count(*) as count, column2'))->get();
```

Eloquent

```
Model::create(array('key' => 'value'));

Model::fill($attributes);
Model::destroy(1);
Model::all();
Model::find(1);

Model::find(array('first', 'last'));

Model::findOrFail(1);

Model::findOrFail(array('first', 'last'));
Model::where('foo', '=', 'bar')->get();
Model::where('foo', '=', 'bar')->first();

Model::where('foo', '=', 'bar')->firstOrFail();
Model::where('foo', '=', 'bar')->count();
Model::where('foo', '=', 'bar')->delete();
Model::whereRaw('foo = bar and cars = 2', array(20))->get();
Model::remember(5)->get();
Model::remember(5, 'cache-key-name')->get();
Model::on('connection-name')->find(1);
Model::with('relation')->get();
Model::all()->take(10);
Model::all()->skip(10);
```

Soft Delete

```
Model::withTrashed()->where('cars', 2)->get();
Model::withTrashed()->where('cars', 2)->restore();
Model::where('cars', 2)->forceDelete();
Model::onlyTrashed()->where('cars', 2)->get();
```

Events

```
Model::creating(function($model){});
Model::created(function($model){});
Model::updating(function($model){});
Model::updated(function($model){});
Model::saving(function($model){});
Model::saved(function($model){});
Model::deleting(function($model){});
Model::deleted(function($model){});
Model::observe(new FooObserver);
```

Eloquent Configuration

```
Eloquent::unguard();

Eloquent::reguard();
```

The Remote Component

Executing Commands

```
SSH::run(array $commands);
SSH::into($remote)->run(array $commands);
// specify remote, otherwise assumes default

SSH::run(array $commands, function($line)
{
        echo $line.PHP_EOL;
});
```

Tasks

```
SSH::define($taskName, array $commands);
// define

SSH::task($taskName, function($line)
// execute
 {
        echo $line.PHP_EOL;
});
```

SFTP Uploads

```
SSH::put($localFile, $remotePath);
SSH::putString($string, $remotePath);
```

Schema

```
Schema::create('table', function($table)
{
    $table->increments('id');
});

Schema::connection('foo')->create('table', function($table){});
Schema::rename($from, $to);
Schema::drop('table');
Schema::dropIfExists('table');
Schema::hasTable('table');
Schema::hasColumn('table', 'column');

Schema::table('table', function($table){});
$table->renameColumn('from', 'to');
$table->dropColumn(string|array);
$table->engine = 'InnoDB';

$table->string('name')->after('email');
```

Indexes

```
$table->string('column')->unique();
$table->primary('column');

$table->primary(array('first', 'last'));
$table->unique('column');
$table->unique('column', 'key_name');

$table->unique(array('first', 'last'));
$table->unique(array('first', 'last'), 'key_name');
$table->index('column');
$table->index('column', 'key_name');

$table->index(array('first', 'last'));
$table->index(array('first', 'last'), 'key_name');
$table->dropPrimary('table_column_primary');
```

```
$table->dropUnique('table_column_unique');
$table->dropIndex('table_column_index');
```

Foreign Keys

```
$table->foreign('user_id')->references('id')->on('users');
$table->foreign('user_id')->references('id')->on('users')->onDelete('cascade');
$table->dropForeign('posts_user_id_foreign');
```

Column Types

```
$table->increments('id');
$table->bigIncrements('id');
$table->string('email');
$table->string('name', 100);
$table->integer('votes');
$table->bigInteger('votes');
$table->smallInteger('votes');
$table->float('amount');
$table->double('column', 15, 8);
$table->decimal('amount', 5, 2);
$table->boolean('confirmed');
$table->date('created_at');
$table->dateTime('created_at');
$table->time('sunrise');
$table->timestamp('added_on');
$table->timestamps();
$table->softDeletes();
$table->text('description');
$table->binary('data');
$table->enum('choices', array('foo', 'bar'));
->nullable()
->default($value)
->unsigned()
```

Input

```
Input::get('key');
Input::get('key', 'default');
Input::has('key');
Input::all();
Input::only('foo', 'bar');
Input::except('foo');
```

Session Input (flash)

```
Input::flash();
Input::flashOnly('foo', 'bar');
Input::flashExcept('foo', 'baz');
Input::old('key','default_value');
```

Files

```
Input::file('filename');

Input::hasFile('filename');

Input::file('name')->getRealPath();
Input::file('name')->getClientOriginalName();
Input::file('name')->getClientOriginalExtension();
Input::file('name')->getSize();
Input::file('name')->getMimeType();
Input::file('name')->move($destinationPath);

Input::file('name')->move($destinationPath, $fileName);
```

Cache

```
Cache::put('key', 'value', $minutes);
Cache::add('key', 'value', $minutes);
Cache::forever('key', 'value');
Cache::remember('key', $minutes, function(){ return 'value' });
Cache::rememberForever('key', function(){ return 'value' });
Cache::forget('key');
Cache::has('key');
Cache::get('key');
Cache::get('key', 'default');
Cache::get('key', function(){ return 'default'; });
```

```
Cache::increment('key');
Cache::increment('key', $amount);
Cache::decrement('key');
Cache::decrement('key', $amount);
Cache::section('group')->put('key', $value);
Cache::section('group')->get('key');
Cache::section('group')->flush();
```

Cookies

```
Cookie::get('key');

Cookie::forever('key', 'value');

Cookie::queue('key', 'value', 'minutes');

$response = Response::make('Hello World');
$response->withCookie(Cookie::make('name', 'value', $minutes));
```

Sessions

```
Session::get('key');
Session::get('key', 'default');
Session::get('key', function(){ return 'default'; });
Session::put('key', 'value');
Session::all();
Session::has('key');
Session::forget('key');
Session::flush();
Session::regenerate();
Session::flash('key', 'value');
Session::reflash();
Session::keep(array('key1', 'key2'));
```

Requests

```
Request::path();
Request::is('foo/*');
Request::url();
Request::segment(1);
Request::header('Content-Type');
Request::server('PATH_INFO');
Request::ajax();
Request::secure();
```

Responses

```
return Response::make($contents);
return Response::make($contents, 200);
return Response::json(array('key' => 'value'));
return Response::json(array('key' => 'value'))
        ->setCallback(Input::get('callback'));
return Response::download($filepath);
return Response::download($filepath, $filename, $headers);

$response = Response::make($contents, 200);
$response->header('Content-Type', 'application/json');
return $response;

 return Response::make($content)
        ->withCookie(Cookie::make('key', 'value'));
```

Redirects

```
return Redirect::to('foo/bar');
return Redirect::to('foo/bar')->with('key', 'value');
return Redirect::to('foo/bar')->withInput(Input::get());
return Redirect::to('foo/bar')->withInput(Input::except('password'));
return Redirect::to('foo/bar')->withErrors($validator);
return Redirect::back();
return Redirect::route('foobar');
return Redirect::route('foobar', array('value'));
return Redirect::route('foobar', array('key' => 'value'));
return Redirect::action('FooController@index');
return Redirect::action('FooController@baz', array('value'));
return Redirect::action('FooController@baz', array('key' => 'value'));
return Redirect::intended('foo/bar');
```

IoC

```
App::bind('foo', function($app){ return new Foo; });
App::make('foo');

App::make('FooBar');
App::singleton('foo', function(){ return new Foo; });
App::instance('foo', new Foo);
App::bind('FooRepositoryInterface', 'BarRepository');
App::register('FooServiceProvider');

App::resolving(function($object){});
```

Security

Passwords

```
Hash::make('secretpassword');
Hash::check('secretpassword', $hashedPassword);
Hash::needsRehash($hashedPassword);
```

Auth

```
Auth::check();
Auth::user();
Auth::attempt(array('email' => $email, 'password' => $password));

Auth::attempt($credentials, true);
Auth::once($credentials);
Auth::login(User::find(1));
Auth::loginUsingId(1);
Auth::logout();
Auth::validate($credentials);
Auth::basic('username');
Auth::onceBasic();
Password::remind($credentials, function($message, $user){});
```

Encryption

```
Crypt::encrypt('secretstring');
Crypt::decrypt($encryptedString);
Crypt::setMode('ctr');
Crypt::setCipher($cipher);
```

Mail

```
Mail::send('email.view', $data, function($message){});
Mail::send(array('html.view', 'text.view'), $data, $callback);
Mail::queue('email.view', $data, function($message){});
Mail::queueOn('queue-name', 'email.view', $data, $callback);
Mail::later(5, 'email.view', $data, function($message){});
Mail::pretend();
```

Messages

These can be used on the $message instance passed into Mail::send() or Mail::queue()

```
$message->from('email@example.com', 'Mr. Example');
$message->sender('email@example.com', 'Mr. Example');
$message->returnPath('email@example.com');
$message->to('email@example.com', 'Mr. Example');
$message->cc('email@example.com', 'Mr. Example');
$message->bcc('email@example.com', 'Mr. Example');
$message->replyTo('email@example.com', 'Mr. Example');
$message->subject('Welcome to the Jungle');
$message->priority(2);
$message->attach('foo\bar.txt', $options);

$message->attachData('bar', 'Data Name', $options);

$message->embed('foo\bar.txt');
$message->embedData('foo', 'Data Name', $options);

$message->getSwiftMessage();
```

Queues

```
Queue::push('SendMail', array('message' => $message));
Queue::push('SendEmail@send', array('message' => $message));
Queue::push(function($job) use $id {});
php artisan queue:listen
php artisan queue:listen connection
php artisan queue:listen --timeout=60
php artisan queue:work
```

Validation

```
Validator::make(
        array('key' => 'Foo'),
        array('key' => 'required|in:Foo')
);
Validator::extend('foo', function($attribute, $value, $params){});
Validator::extend('foo', 'FooValidator@validate');
Validator::resolver(function($translator, $data, $rules, $msgs)
{
        return new FooValidator($translator, $data, $rules, $msgs);
});
```

Rules

```
accepted
active_url
after:YYYY-MM-DD
before:YYYY-MM-DD
alpha
alpha_dash
alpha_num
between:1,10
confirmed
date
date_format:YYYY-MM-DD
different:fieldname
email
exists:table,column
image
in:foo,bar,baz
not_in:foo,bar,baz
integer
```

```
numeric
ip
max:value
min:value
mimes:jpeg,png
regex:[0-9]
required
required_if:field,value
required_with:foo,bar,baz
required_without:foo,bar,baz
same:field
size:value
unique:table,column,except,idColumn
url
```

Views

```
View::make('path/to/view');
View::make('foo/bar')->with('key', 'value');
View::make('foo/bar')->withKey('value');
View::make('foo/bar', array('key' => 'value'));
View::exists('foo/bar');

View::share('key', 'value');

View::make('foo/bar')->nest('name', 'foo/baz', $data);

View::composer('viewname', function($view){});

View::composer(array('view1', 'view2'), function($view){});

View::composer('viewname', 'FooComposer');
View::creator('viewname', function($view){});
```

Blade Templates

```
@extends('layout.name')

@section('name')

@stop

@show

@yield('name')
@include('view.name')
@include('view.name', array('key' => 'value'));
@lang('messages.name')
@choice('messages.name', 1);
@if
@else
@elseif
@endif
@unless
@endunless
@for
@endfor
@foreach
@endforeach
@while
@endwhile

 {{ $var }}

 {{{ $var }}}
{{-- Blade Comment --}}
```

Forms

```
Form::open(array('url' => 'foo/bar', 'method' => 'PUT'));
Form::open(array('route' => 'foo.bar'));
Form::open(array('route' => array('foo.bar', $parameter)));
Form::open(array('action' => 'FooController@method'));
Form::open(array('action' => array('FooController@method', $parameter)));
Form::open(array('url' => 'foo/bar', 'files' => true));
Form::close();
Form::token();
Form::model($foo, array('route' => array('foo.bar', $foo->bar)));
```

Form Elements

```
Form::label('id', 'Description');
Form::label('id', 'Description', array('class' => 'foo'));
Form::text('name');
Form::text('name', $value);
Form::text('name', $value, array('class' => 'name'));
Form::textarea('name');
Form::textarea('name', $value);
Form::textarea('name', $value, array('class' => 'name'));
Form::hidden('foo', $value);
Form::password('password');
Form::password('password', array('placeholder' => 'Password'));
Form::email('name', $value, array());
Form::file('name', array('class' => 'name'));
Form::checkbox('name', 'value');

 Form::checkbox('name', 'value', true, array('class' => 'name'));
Form::radio('name', 'value');

 Form::radio('name', 'value', true, array('class' => 'name'));
Form::select('name', array('key' => 'value'));
Form::select('name', array('key' => 'value'), 'key', array('class' => 'name'));
Form::submit('Submit!');
Form::macro('fooField', function()
{
        return '<input type="custom"/>';
});
Form::fooField();
```

HTML Builder

```
HTML::macro('name', function(){});
HTML::entities($value);
HTML::decode($value);
HTML::script($url, $attributes);
HTML::style($url, $attributes);
HTML::image($url, $alt, $attributes);
HTML::link($url, 'title', $attributes, $secure);
HTML::secureLink($url, 'title', $attributes);
HTML::linkAsset($url, 'title', $attributes, $secure);
HTML::linkSecureAsset($url, 'title', $attributes);
HTML::linkRoute($name, 'title', $parameters, $attributes);
HTML::linkAction($action, 'title', $parameters, $attributes);
HTML::mailto($email, 'title', $attributes);
HTML::email($email);
HTML::ol($list, $attributes);
HTML::ul($list, $attributes);
HTML::listing($type, $list, $attributes);
HTML::listingElement($key, $type, $value);
HTML::nestedListing($key, $type, $value);
HTML::attributes($attributes);
HTML::attributeElement($key, $value);
HTML::obfuscate($value);
```

Strings

```
Str::ascii($value)
Str::camel($value)
Str::contains($haystack, $needle)
Str::endsWith($haystack, $needles)

Str::finish($value, $cap)
Str::is($pattern, $value)
Str::length($value)
Str::limit($value, $limit = 100, $end = '...')
Str::lower($value)
Str::words($value, $words = 100, $end = '...')
Str::plural($value, $count = 2)

Str::random($length = 16)
Str::quickRandom($length = 16)
Str::upper($value)
Str::title($value)
```

```
Str::singular($value)
Str::slug($title, $separator = '-')
Str::snake($value, $delimiter = '_')
Str::startsWith($haystack, $needles)
Str::studly($value)
Str::macro($name, $macro)
```

Localization

```
App::setLocale('en');
Lang::get('messages.welcome');
Lang::get('messages.welcome', array('foo' => 'Bar'));
Lang::has('messages.welcome');
Lang::choice('messages.apples', 10);
```

Files

```
File::exists('path');
File::get('path');
File::getRemote('path');
File::getRequire('path');
File::requireOnce('path');
File::put('path', 'contents');
File::append('path', 'data');
File::delete('path');
File::move('path', 'target');
File::copy('path', 'target');
File::extension('path');
File::type('path');
File::size('path');
File::lastModified('path');
File::isDirectory('directory');
File::isWritable('path');
File::isFile('file');
File::glob($patterns, $flag);
File::files('directory');
File::allFiles('directory');
File::directories('directory');
File::makeDirectory('path',  $mode = 0777, $recursive = false);
File::copyDirectory('directory', 'destination', $options = null);
File::deleteDirectory('directory', $preserve = false);
File::cleanDirectory('directory');
```

Helpers

Arrays

```
array_add($array, 'key', 'value');
array_build($array, function(){});
array_divide($array);
array_dot($array);
array_except($array, array('key'));
array_fetch($array, 'key');
array_first($array, function($key, $value){}, $default);
array_flatten($array);
array_forget($array, 'foo');
array_forget($array, 'foo.bar');
array_get($array, 'foo', 'default');
array_get($array, 'foo.bar', 'default');
array_only($array, array('key'));
array_pluck($array, 'key');
array_pull($array, 'key');
array_set($array, 'key', 'value');
array_set($array, 'key.subkey', 'value');
array_sort($array, function(){});
head($array);
last($array);
```

Paths

```
app_path();
public_path();
base_path();
storage_path();
```

Strings

```
camel_case($value);
class_basename($class);
e('<html>');
starts_with('Foo bar.', 'Foo');
ends_with('Foo bar.', 'bar.');
snake_case('fooBar');
str_contains('Hello foo bar.', 'foo');
str_finish('foo/bar', '/');
str_is('foo*', 'foobar');
str_plural('car');
str_random(25);
str_singular('cars');
studly_case('foo_bar');
trans('foo.bar');
trans_choice('foo.bar', $count);
```

URLs and Links

```
action('FooController@method', $parameters);
link_to('foo/bar', $title, $attributes, $secure);
link_to_asset('img/foo.jpg', $title, $attributes, $secure);
link_to_route('route.name', $title, $parameters, $attributes);
link_to_action('FooController@method', $title, $params, $attrs);

asset('img/photo.jpg', $title, $attributes);

secure_asset('img/photo.jpg', $title, $attributes);

secure_url('path', $parameters);
route($route, $parameters, $absolute = true);
url('path', $parameters = array(), $secure = null);
```

Miscellaneous

```
csrf_token();
dd($value);
value(function(){ return 'bar'; });
with(new Foo)->chainedMethod();
```

Credits to: Jesse Obrien and Laravel Docs.

PART 5: BUILDING A COMPLETE CMS FROM SCRATCH

Chapter 4 - Building A Responsive Website From Scratch (TODO)

Coming Soon

I'm writing, please wait. If you have any ideas or want to learn something, don't hesitate to send me a message. If this book have helped you in anyway, then I would really appreciate if you would share the URL to the book with your friends. It's at www.learninglaravel.net[60] :D

[60] http://learninglaravel.net

APPENDICES

Basic HTML5, CSS3, Twitter BootStrap And PHP Knowledge

Coming Soon

I'm writing, please wait. If you have any ideas or want to learn something, don't hesitate to send me a message. If this book have helped you in anyway, then I would really appreciate if you would share the URL to the book with your friends. It's at www.learninglaravel.net[61] :D

[61] http://learninglaravel.net