ANGULAR 4 BOOK

LEARNING ANGULAR WITH LARAVEL 5

by NATHAN WU

Table of Contents

Part 1

Part 2

Part 3

Part 4

Part 5

About This Book

> This is the first Angular 4 (aka AngularJS 4) and Laravel 5 book!

It's time to learn Angular - one of the best frameworks for building modern and cross-platform applications. If you want to get up to speed in AngularJS as quickly as possible, this is the book that you will love!

The book dives straight into building a practical single-page application to explore all core concepts of Angular 4 such as Dependency Injection, Data Binding, Components, Events, Http Module, etc.

Throughout the book, you'll also create many real-world features: CRUD operations, form validations, JWT authentication, etc.

Along the way, we'll learn more about TypeScript, ES6 and Laravel - one of the best PHP frameworks.

Learn by doing!

By the end of this book, you'll gain a solid understanding of Angular 4 and how it interacts with a Laravel backend.

Let's get the book and become a full stack developer!

Requirements

The projects in this book are intended to help frontend and backend developers who want to get a head start in Angular 4 by building some practical applications. The fundamentals of Javascript are not covered, you will need to:

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

What You Will Get

- Lifetime access to the online book. (Premium Only)
- Digital books: PDF, MOBI, EPUB (Premium Only)
- Full source code (Premium Only)
- Access new chapters of the book while it's being written (Premium Only)
- A community of 20000+ students learning together.
- Amazing bundles and freebies to help you become a successful developer.
- iPhone, iPad and Android Accessibility.

Book Structure

Chapter 1 - Diving Into Angular World

In this chapter, we will learn how to set up the development environment and run our first Angular 4 app. After that, we talk about TypeScript and why we should use TypeScript to develop Angular applications.

Chapter 2 - Building Our First Angular Application

Our first app, which is a simple image gallery, will walk you through the structure of an Angular application. We'll learn how to integrate Twitter Bootstrap into your Angular app, and how to create Angular components. Along the way, we'll talk about data binding, events, services, routers other core concepts of Angular 4.

Chapter 3 - Building A Backend API With Laravel

It's time to learn how to build a backend for our Angular applications. We'll learn how to install Laravel PHP framework and build APIs with Laravel 5. After that, we'll learn how to connect our Angular app with the Laravel backend.

Chapter 4 - Building An Admin Control Panel Using Angular 2 and Laravel

In this final chapter, we will build a real-world Admin Control Panel. During the process, you'll know how to architect your Angular app, and how to authenticate your App with JSON Web Tokens. You'll also learn some Angular best practices and conventions.

You can be able to build your own powerful single-page applications in no time!

Feedback

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

Simply send an email to support@angularbooks.com.

We're always here.

Translation

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

Feel free to contact us at support@angularbooks.com.

Here is a list of our current translators:

List of Translators

Book Status, Changelog and Contributors

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

Book Status

Changelog

Contributors

Changelog

Current Version

Latest version the book:

- Version: 1.3
- Status: Complete.
- Updated: May 13 2017

Chapter 1 - Diving Into Angular World

Introduction

> **Note:** Angular 4 is an enhanced version of Angular 2. Angular 4 is now known simply as **Angular**. So when I mention **Angular**, it means **Angular 4**!

Welcome to the Angular world! In this chapter, we'll learn how to prepare our development environment and run our first Angular app.

After that, we'll talk about **TypeScript** and **ECMAScript 2015** (aka **ES6**). Don't worry if you have never heard of these technologies. They're very easy to learn. **ES6** is the newest version of JavaScript, and it has many great new features. **TypeScript** is a strict superset of Javascript, which is developed and maintained by Microsoft. Angular is actually written in TypesScript.

We'll learn more about ES6 and TypeScript later. For now, let's get started with the first step in getting our application up and running!

Project Files

All project files of this book can be downloaded at:

https://github.com/AngularBooks/angular4/releases

Feel free to download and use them at any stage of your development process.

Introducing CLI (Command Line Interface)

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

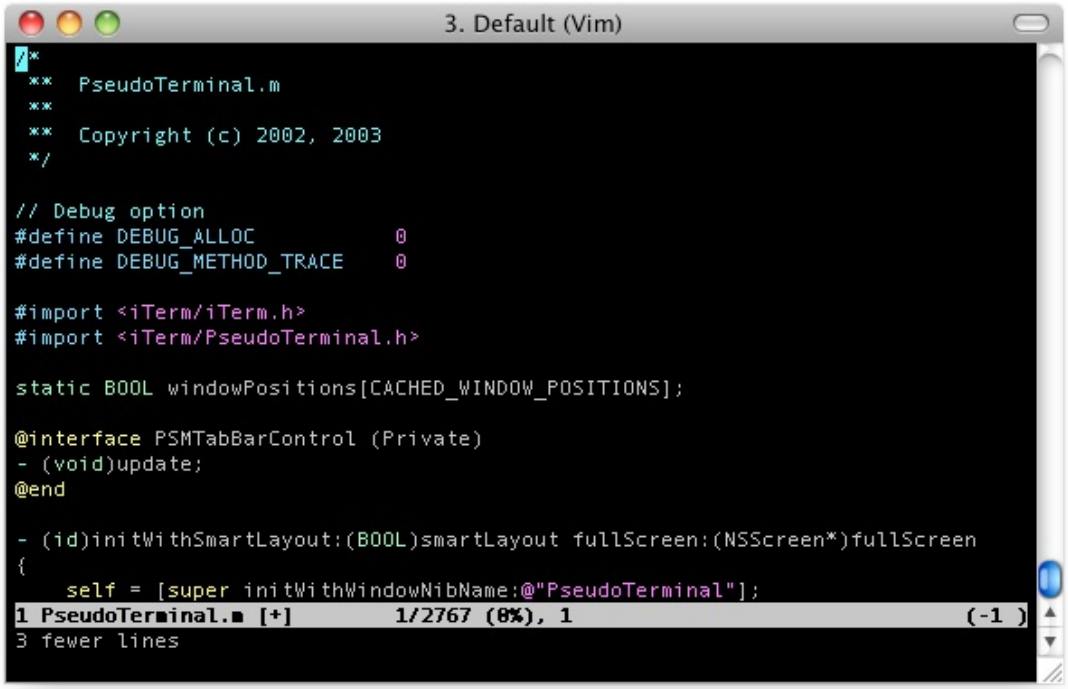
This section shows you how to use the command line on PC, Mac and Linux.

CLI for MAC OSX

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

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

Alternatively, you can use iTerms 2.



CLI for Windows

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

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

http://msysgit.github.io

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

Alternatively, you may use Cygwin.

CLI for Linux

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

Installing npm (Node.js Package Manager)

First, we have to install **npm** (**Node.js Package Manager**) because **Node.js** and **npm** are essential to Angular development.

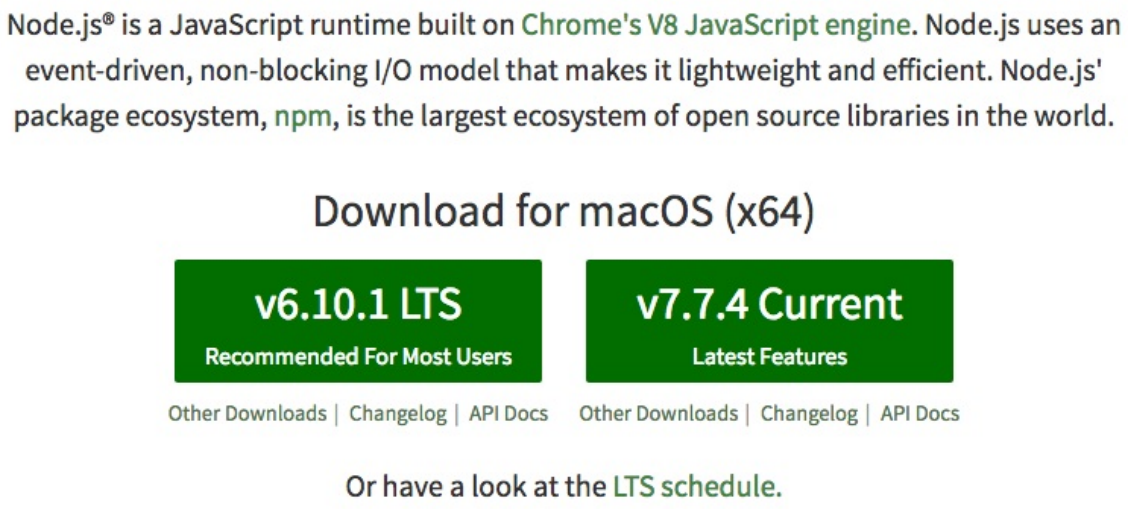
Angular 4 applications use some features and functionalities provided by a variety of third-party packages. These packages can be installed and updated with the **Node Package Manager**.

> **Tip:** Sounds familiar? In PHP, we also have **Composer** - which is a PHP version of **npm**. Actually, **Composer** is self-described as inspired by **npm**.

Now, let's go to:

https://nodejs.org

Download the latest version of **Node.js** and install it!

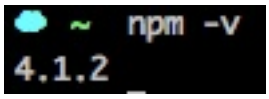


At the time of writing this book, the current LTS version is **6.9.2** and the normal version is **7.2.1**. Be sure to download **Node.js 7.2.1 or newer**.

> **Note:** You can find the direct download links of Node.js 7.2.1 at https://nodejs.org/en/blog/release/v7.2.1/

After installed, open your **CLI** (**Terminal** or **Git Bash**) and run this command to check what version of **npm** you're using:

```
npm -v
```



As you see, my current **npm** version is **4.1.2**.

Usually, when you install **Node.js**, **npm** is automatically installed for you.

If you want, you can install **npm** manually or **upgrade to the newest version** of **npm** by running this command:

For **Mac** and **Linux** users:

```
sudo npm install npm -g
```

For **Windows** users:

```
npm install npm -g
```

We should now have **npm** installed!

Installing TypeScript

As mentioned earlier, TypeScript is a superset of Javascript. To be exact, TypeScript is a superset of **ES6**. Currently, many browsers don't support ES6 and TypeScript yet. When we write **ES6/TypeScript** code, we need **a transpiler** to **"translate"** our code to **ES5 code** that current browsers understand. There are many transpilers that we can find, here are the most popular ones:

- Babel: Babel is a popular transpiler created by popular Javascript developers.
- Traceur: This is another good compiler, developed by Google.

- Official TypeScript transpiler: the core TypeScript team also wrote a simple transpiler that we can use to easily compile our ES6 code. This is the transpiler that we will use in this book.

Because we have **npm** installed already, we can use **npm** to install the **TypeScript compiler** in no time. Let's run this command:

```
npm install -g typescript
```

> **Note:** If you get a "permission error", try again using this command: **sudo npm install -g typescript**

```
~  sudo npm install -g typescript
/usr/local/bin/tsc -> /usr/local/lib/node_modules/typescript/bin/tsc
/usr/local/bin/tsserver -> /usr/local/lib/node_modules/typescript/bin/tsserver
/usr/local/lib
└── typescript@1.8.10
```

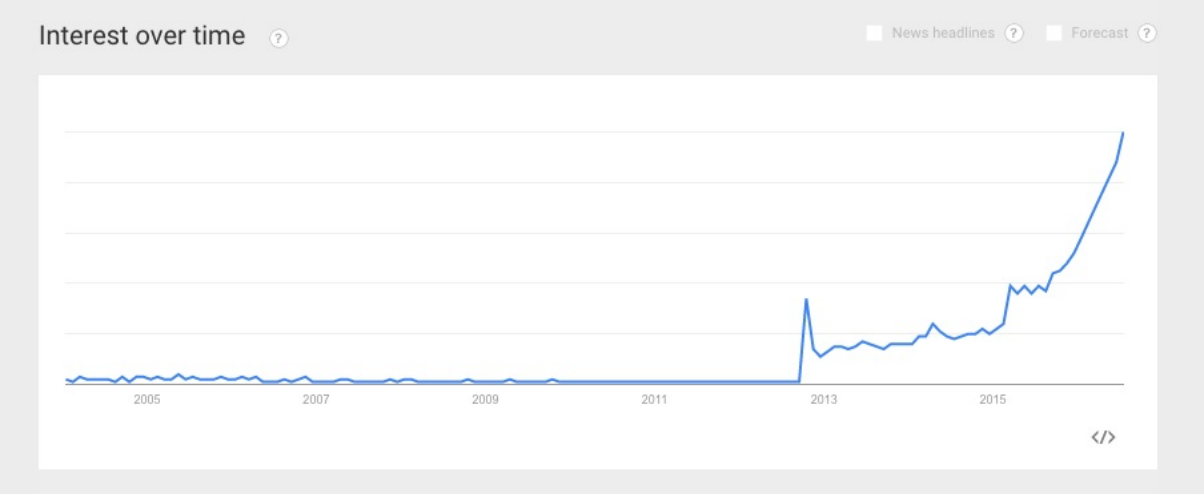
Well done! **TypeScript transpiler** has been installed! We can start writing TypeScript now.

But wait, do we have to use TypeScript to write Angular?

No. TypeScript is not required. Angular is quite flexible. We can use **ES5**, **ES6** or **TypeScript** to code an Angular app. However, we should use **TypeScript** because it has an **optional typing system** and some features that we can use to easily detect errors. The best thing is, TypeScript and **plain JS** (ES5, ES6, ES7, etc.) live well together. That means we can use TypeScript to write parts of our code and we can use **ES6** to write other parts if we like.

Is it worth our time to learn **TypeScript**?

TypeScript is gaining a huge popularity boost. Many professional coders are adopting it. Let's take a look at this Google Trends' interest over time to see how it grows:



Astounding, right?

In my humble opinion, it's worth taking our time to learn **TypeScript**.

Creating your first Angular application with angular-cli

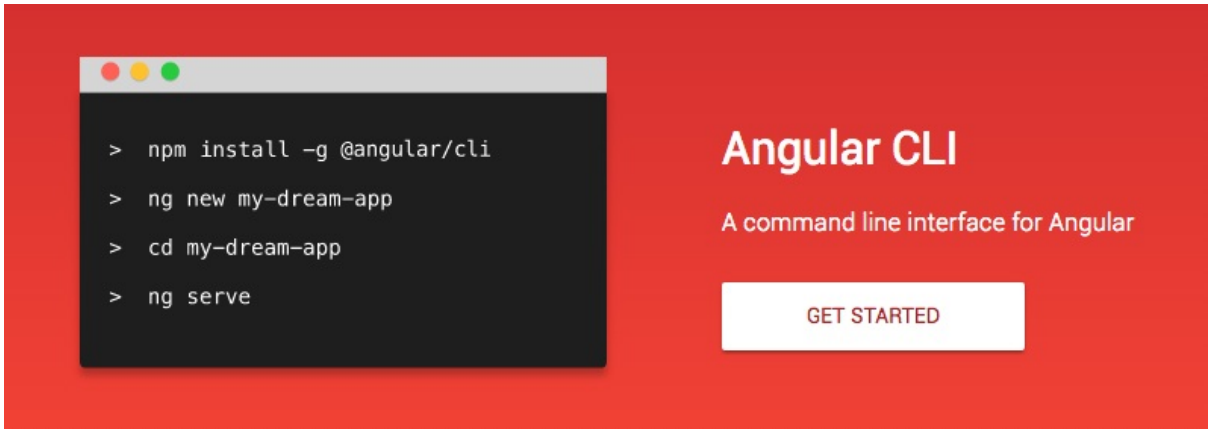
Introducing angular-cli

There are many ways to create a new Angular app, here are some common ways:

- We may use **npm** or **yarn** to download and install packages to create a new Angular app from scratch.
- We may use some **project generators** (such as **Yeoman**, **Slush**, etc.) to generate a new project.
- We can download some **starter templates** that have all required functionalities and start building our Angular app.

Many people are still using one of the methods above to getting started with Angular. There is nothing wrong with that. However, we now have a better method: using **angular-cli**.

angular-cli is the new tool that is made by the **official Angular team**. It's a **command line interface** (**cli**) for Angular.



We can use **angular-cli** to do many things:

- Create a new Angular application
- Generate components, routes, services and pipes
- Test our applications and format our code.
- Create a simple HTTP Server and run our app.
- Deploy our Angular applications to Github or other services.

There are endless possibilities. You can do almost anything with **angular-cli**. If you want, you can download the source code and add more features to it:

https://github.com/angular/angular-cli

Just a bit more information, **angular-cli** is actually not a new idea. It is based on another popular framework's cli: EmberJS's CLI (aka ember cli). At the time of writing this book, the tool is still new, but it will be a standard for creating AngularJS applications soon.

> **Tip:** If you're using Laravel, **angular-cli** works just like **Laravel's Artisan** (the **php artisan** things, you know) with some more useful features.

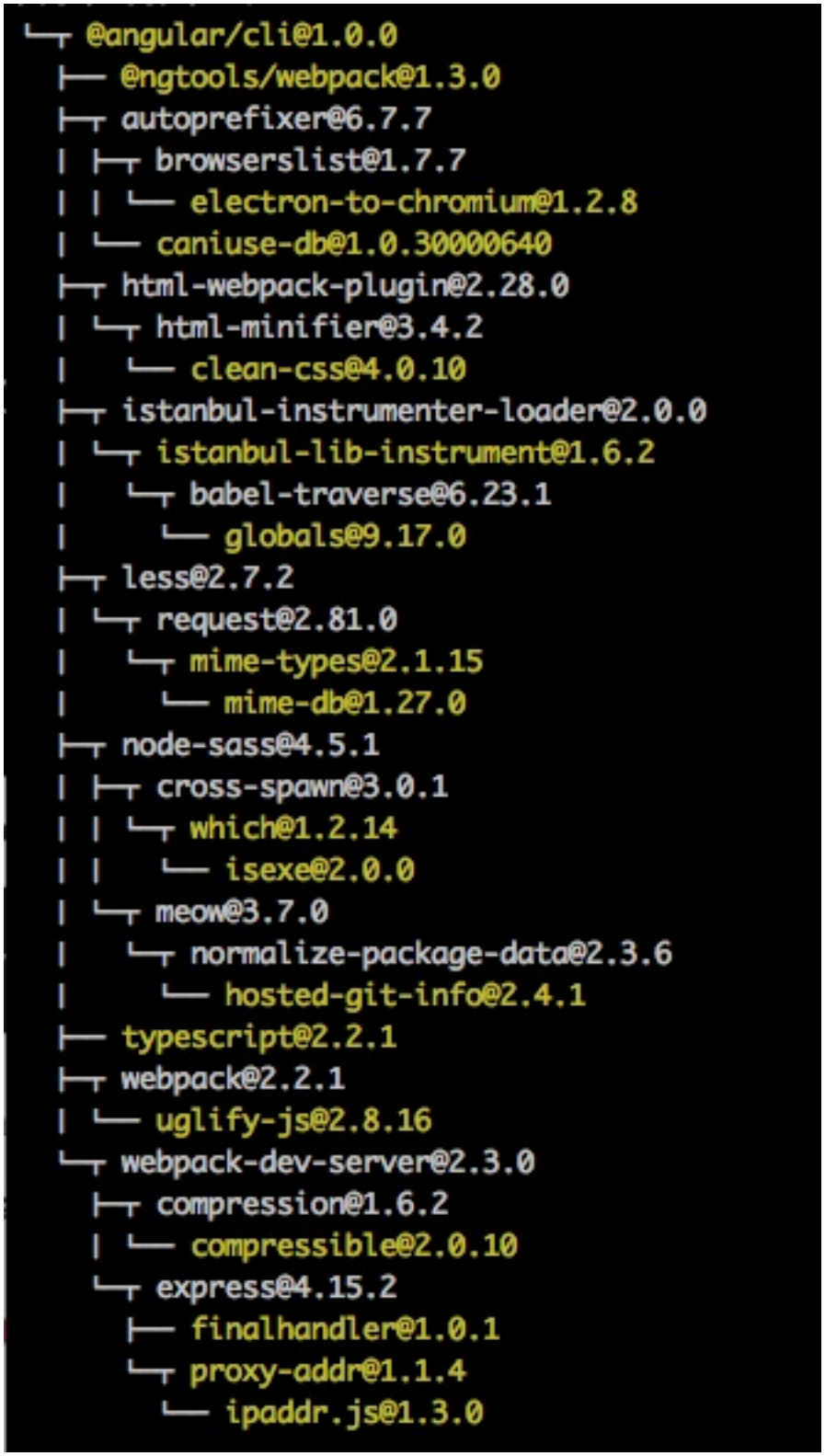
Now, let's install it!

Installing angular-cli

Installing **angular-cli** is pretty easy. We just need to run this command:

```
npm install -g @angular/cli
```

> **Note:** If you get a "permission error", try again using this command: **sudo npm install -g @angular/cli**. If you're using Windows, you have to run the command line **as an administrator**.



Alternatively, we may use this command to install the latest dev version of **angular-cli**:

```
npm install --save-dev @angular/cli@latest
```

Done! You've successfully installed angular-cli!

You may learn more commands by reading Angular CLI Wiki..

Creating a new Angular app with angular-cli

Let's say I want to **create a new app** called **ngbook** (Angular book), we can easily create the app with:

```
ng new ngbook
```

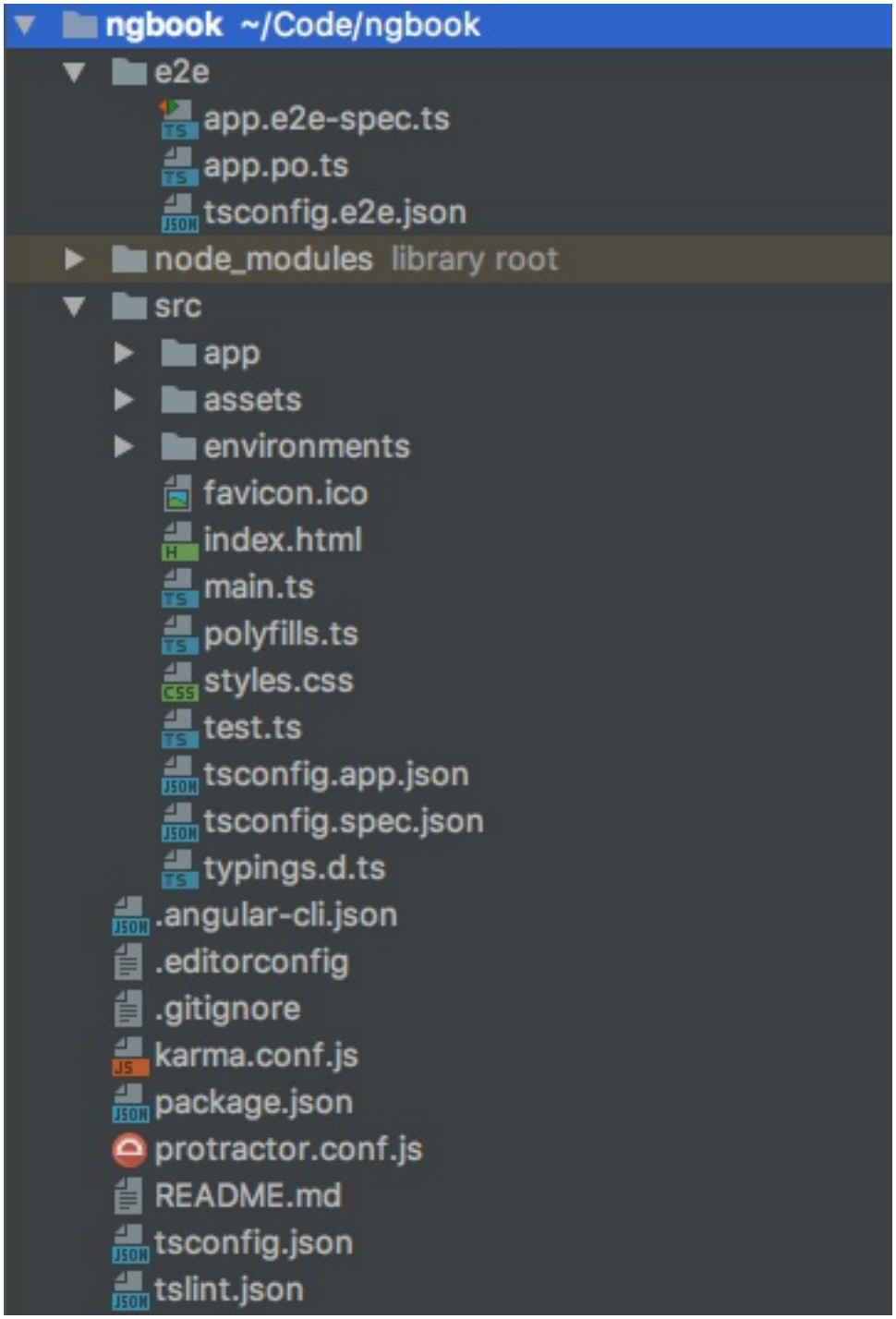
This command will create a new project skeleton (**ngbook**) and download all required components and packages for us.

```
Code  ng new ngbook
installing ng
  create .editorconfig
  create README.md
  create src/app/app.component.css
  create src/app/app.component.html
  create src/app/app.component.spec.ts
  create src/app/app.component.ts
  create src/app/app.module.ts
  create src/assets/.gitkeep
  create src/environments/environment.prod.ts
  create src/environments/environment.ts
  create src/favicon.ico
  create src/index.html
  create src/main.ts
  create src/polyfills.ts
  create src/styles.css
  create src/test.ts
  create src/tsconfig.app.json
  create src/tsconfig.spec.json
  create src/typings.d.ts
  create .angular-cli.json
  create e2e/app.e2e-spec.ts
  create e2e/app.po.ts
  create e2e/tsconfig.e2e.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tsconfig.json
  create tslint.json
Successfully initialized git.
Installing packages for tooling via npm.
Installed packages for tooling via npm.
Project 'ngbook' successfully created.
```

> **Note:** The app will be created in the location where you've executed the command. For **Laravel Homestead** users, you can navigate to the **Code directory** and install the project there. Basically, you might install the app anywhere.

Now, let's take a look at the app directory:

> **Note:** You can open the directory using **Finder** or **Explorer** or **any file explorer** that you like.



Here is a typical **Angular application's structure**. There are many files that you may not understand. But don't worry, we'll learn all about them soon.

For now, let's **navigate** to the **app directory**:

```
cd ngbook
```

Then we can start the app by running this command:

```
ng serve
```

We should see this line:

```
** NG Live Development Server is running on http://localhost:4200 **
```

http://localhost:4200/ is where we can access our Angular app.

Open your browser, and go to http://localhost:4200/

> **Note:** Your app **URL** could be different.

Congratulations! Your first Angular app is working!

The **ng serve** command started **a HTTP server** locally for us, so we don't have to use **MAMP**, **XAMPP** or **Homestead** to develop an Angular app.

The good news is, we also have a **Livereload server** running. If we modify a file and save it, the app will automatically refresh in our browser!

Creating your first component with Angular

One of the most important features of Angular is **Component**. Basically, an Angular app is built by many components. When we first visit our app, we should see this message: **"app works!"**

> **Note:** The message could be different if you're using a different version of angular-cli.

The message **is rendered** by a simple component, called **app.component.ts**.

app.component.ts is a **TypeScript file** (aka **typing file**), which can be found at **/src/app/**

Let's open this component using your favorite text editor (Sublime Text, PHPStorm, etc.):

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

If you're new to TypeScript, this syntax might look a bit messy and unfamiliar. Don't worry, let's take a look at the code, line by line.

```
import { Component } from '@angular/core';
```

First, we use the **import** statement to tell Angular that we want to access another component.

```
{ Component }
```

This **Component** is called **a decorator** (aka **Component Decorator**), which can be found in the **@angular/core** library. As mentioned earlier, Angular has many libraries that we can utilize.

Once the component is imported, we can then begin **configuring** our component using TypeScript's @ symbol:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

There are lots of options that we can use to configure our component. Here are the **Component options** that we're using:

selector option

We can define the name of a custom **HTML tag** here. Notice that we're using the **app-root** tag? Every time Angular finds a **app-root** tag in our template, our component (**AppComponent**) will be activated.

For example, when we use this in our **index.html**:

```
<app-root></app-root>
```

The **AppComponent** will be loaded.

> **Note:** the **app-root** tag can be found in the **src/index.html** file.

template option

We can declare a **HTML template** for our component:

```
templateUrl: 'app.component.html'
```

When defining a template, we can use an **external template** (which is the **app.component.html** file) or we can write it inline like this:

```
template: `<h1>
  {{title}}
</h1>`
```

> **Note:** We're using TypeScript's backtick ` here.

style option

We can use the **style option** to **style a component**:

```
styleUrls: ['app.component.css']
```

Using the **styleUrls** property, we can tell Angular where to find our **external stylesheet** (the **app.component.css** file). As you may have guessed, we can also link **multiple stylesheets**.

Lastly, we can use the **export** statement to **export our component**, so it can be imported to other files of our project:

```
export class AppComponent {
  title = 'app works!';
}
```

> **Tip:** The export statement is the **ES6** module syntax.

As you may have guessed, we may change **the title** here. Let's change it to:

```
export class AppComponent {
  title = 'Our first component!';
}
```

Save the file and go to your browser:

Great! The **title** has been changed!

What just happened? The `{{title}}` in our **app.component.html** was just replaced by our **new string** ("Our first component!").

In Angular, this concept is called **Interpolation**.

Interpolation

Open the **app.component.html**, we should see:

```
<h1>
      {{title}}
</h1>
```

`{{title}}` is a **property**, which is placed inside the double-curly braces `{{ }}`.

When our component is bootstrapped, Angular will evaluate the **title** property and replace it with a right value.

When we **export** our **AppComponent** component, we may tell Angular that we want it to set the **title** property to "**Our first component!**":

```
export class AppComponent {
      title = 'Our first component!';
    }"
```

When Angular sees the **app-root** tag, it creates an instance of our **AppComponent** class. After that, it **evaluates** the **title** property and then **converts it to our defined string**.

Now things get interesting. If we **change the value** of the **title** property, the template will be automatically updated! That means we don't have to open the **app.component.html** file and change the **title** manually.

That's how **interpolation** works!

Creating other components with Angular

Creating a new component is very easy. Let's say we want to create another component called **second**. All we have to do is running this command:

```
ng generate component second
```

> **Note:** Be sure to run this command at the root of our app. You may need to create a new tab/window to run the command.

Angular will automatically generate a new **second** folder and some files for us.

Let's open the **second.component.ts** file:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-second',
  templateUrl: './second.component.html',
  styleUrls: ['./second.component.css']
})
export class SecondComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

Here is an important note. When building an Angular app, be sure that our **selector** is **unique**. If our selector is the same with others' selector, there will be a conflict between the components.

Let's change:

```
app-second
```

to

```
ab-second
```

> **Note: ab** stands for **angular books**. You may use any name that you like.

Next, open **second.component.html** and change:

```
<p>
  second works!
</p>
```

to:

```
<p>
  Our second component!
</p>
```

Now we can display our second component. Open the **app.component.html** file and update it as follows:

```
<h1>
  {{title}}
</h1>
<ab-second></ab-second>
```

Visit your browser, you should see **"Our second component!"**.

As you may have noticed, we've just used a new custom tag

```
<ab-second></ab-second>
```

to display our second component.

Angular CLI reference and shortcuts

A good practice when creating Angular components is to use **shortcuts** (aka **alias**).

Instead of typing:

```
ng generate component second
```

We can simply type:

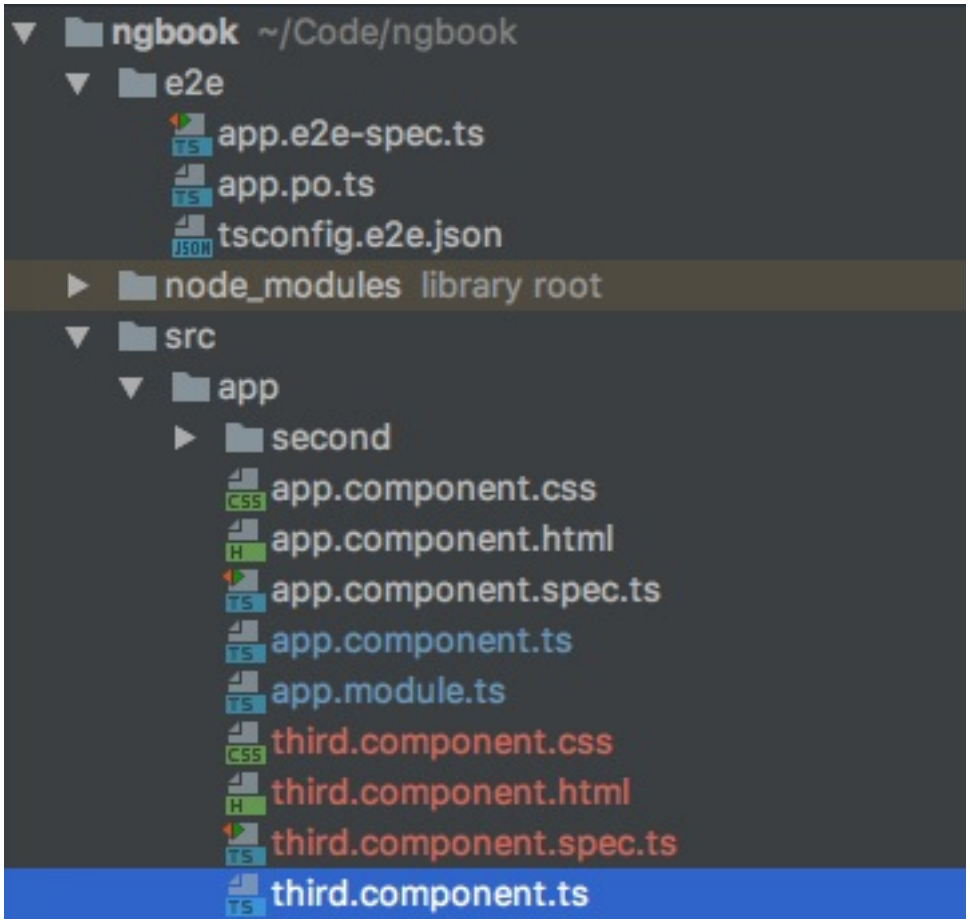
```
ng g c second
```

Really simple, isn't it?

Sometimes, if we don't want to generate a new directory when creating a component, we may use the **--flat** flag:

```
ng g c third --flat
```

Take a look at your app, the **third** component files are generated.



There are many Angular CLI commands that we can use. We will not learn all about them in this section, but we will use some of them throughout this book. If you wish to learn more commands, you may read and download this file:

Angular CLI reference

> **Note:** Angular CLI commands are changed/updated frequently.

Here is a good tip: Angular CLI automatically uses the **app** prefix for our selectors by default (app-third):

```
@Component({
  selector: 'app-third',
  templateUrl: './third.component.html',
  styleUrls: ['./third.component.css']
})
```

If you want to use a different prefix, you may use this command when creating a new app:

```
ng new ngbook --prefix ab
```

As you see, by adding the **--prefix ab** flag, Angular will use **ab** as the prefix for our selectors.

> **Note:** At the time of writing this book, we can only use the **--prefix** flag when **creating a new Angular** app.

Angular Module

NgModule (Angular Module) is a new module system which is introduced in Angular. Using NgModule, we can organize our code effectively.

Every time we generate a new component using the Angular CLI, the **app.module.ts** file will be automatically updated. Let's open it:

src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';
import { SecondComponent } from './second/second.component';
import { ThirdComponent } from './third.component';

@NgModule({
  declarations: [
    AppComponent,
    SecondComponent,
    ThirdComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

As you see, our **SecondComponent** and **ThirdComponent** have been added to the file. This file is **a class** called **AppModule**.

AppModule has the **@NgModule** decorator, which can be used to configure the module.

An Angular application always has at least one module. Usually, the **first module** (the **AppModule**) is known as the **root module**. Of course, we may add more modules later.

Remember that when creating a new component, we must declare it in some **NgModule**.

> **Note:** We have only one module now, so our component will be declared in our **AppModule**.

```
@NgModule({
  declarations: [
    AppComponent,
    SecondComponent,
    ThirdComponent
  ],
```

If you forget to put your component into the **declarations field** of one module, the component won't work.

Every module has a **main component**, and we use the **bootstrap** field to let Angular know which is the module's main component.

```
bootstrap: [AppComponent]
```

Noticed here that our **AppModule** has the **AppComponent** as its **main component**. When we bootstrap our app, the **AppComponent** will be started first.

Creating a new Angular app manually

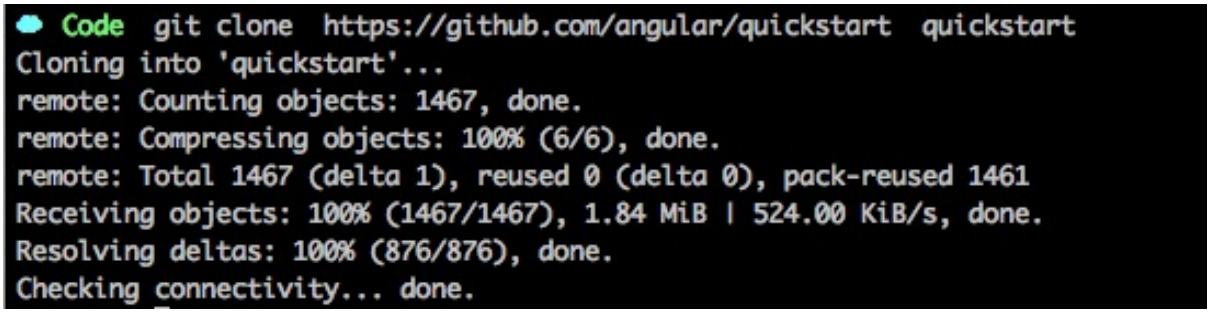
If you don't want to use **Angular CLI**, you may create a new Angular manually as well.

In this section, we'll create a new Angular app using **Angular QuickStart Source**, which can be found at:

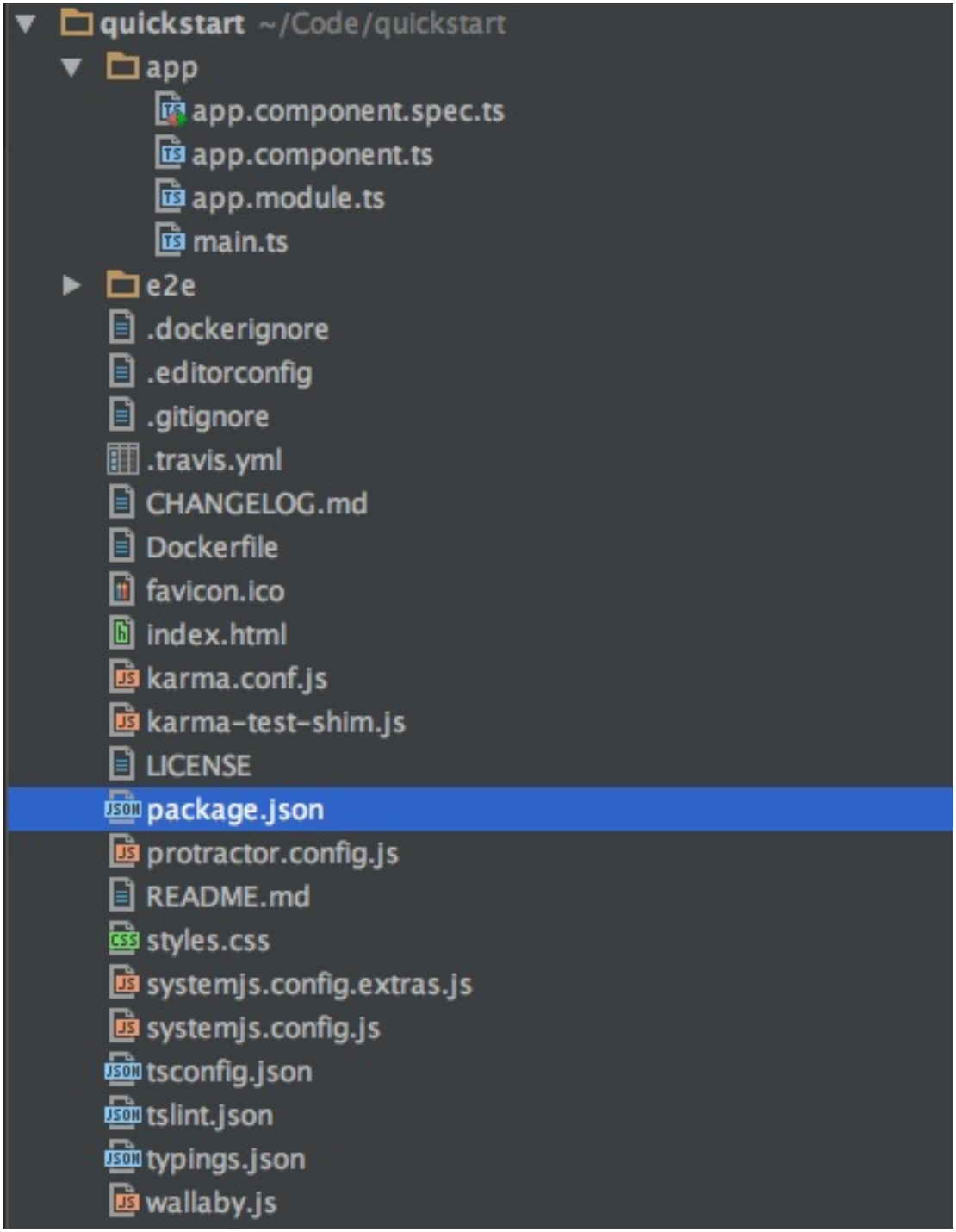
https://github.com/angular/quickstart/blob/master/README.md

First, let's clone the repo into new project folder:

```
git clone  https://github.com/angular/quickstart  quickstart
```



A new app called **quickstart** will be created.



We may navigate to the **app root** by using:

```
cd quickstart
```

Next, we will have to **install dependencies** with **npm** using this command:

```
npm install
```

If you don't see the **typings** folder after running **npm install**, you have to run this command to install it manually:

```
npm run typings install
```

> **Note:** Windows users have to run these command as an administrator.

You may find all Angular dependencies in the **package.json** file:

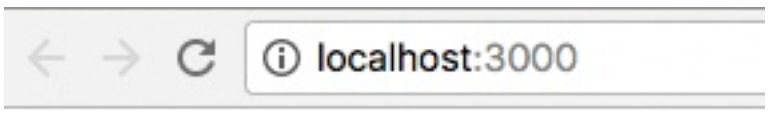
```
 {
  "name": "angular-quickstart",
  "version": "1.0.0",
  "description": "QuickStart package.json from the documentation, supplemented with te
sting support",
  "scripts": {
    "build": "tsc -p src/",
    "build:watch": "tsc -p src/ -w",
    "build:e2e": "tsc -p e2e/",
    "serve": "lite-server -c=bs-config.json",
    "serve:e2e": "lite-server -c=bs-config.e2e.json",
    "prestart": "npm run build",
    "start": "concurrently \"npm run build:watch\" \"npm run serve\"",
    "pree2e": "npm run build:e2e",
    "e2e": "concurrently \"npm run serve:e2e\" \"npm run protractor\" --kill-others --
success first",
    "preprotractor": "webdriver-manager update",
    "protractor": "protractor protractor.config.js",
    "pretest": "npm run build",
    "test": "concurrently \"npm run build:watch\" \"karma start karma.conf.js\"",
    "pretest:once": "npm run build",
    "test:once": "karma start karma.conf.js --single-run",
    "lint": "tslint ./src/**/*.ts -t verbose"
  },
  "keywords": [],
  "author": "",
  "license": "MIT",
  "dependencies": {
    "@angular/common": "~4.0.0",
    "@angular/compiler": "~4.0.0",
    "@angular/core": "~4.0.0",
    "@angular/forms": "~4.0.0",
    "@angular/http": "~4.0.0",
    "@angular/platform-browser": "~4.0.0",
    "@angular/platform-browser-dynamic": "~4.0.0",
    "@angular/router": "~4.0.0",

    "angular-in-memory-web-api": "~0.3.0",
    "systemjs": "0.19.40",
    "core-js": "^2.4.1",
    "rxjs": "5.0.1",
```

```
    "zone.js": "^0.8.4"
  },
  "devDependencies": {
    "concurrently": "^3.2.0",
    "lite-server": "^2.2.2",
    "typescript": "~2.1.0",

    "canonical-path": "0.0.2",
    "tslint": "^3.15.1",
    "lodash": "^4.16.4",
    "jasmine-core": "~2.4.1",
    "karma": "^1.3.0",
    "karma-chrome-launcher": "^2.0.0",
    "karma-cli": "^1.0.1",
    "karma-jasmine": "^1.0.2",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~4.0.14",
    "rimraf": "^2.5.4",

    "@types/node": "^6.0.46",
    "@types/jasmine": "2.5.36"
  },
  "repository": {}
}
```

Every Angular app also has a **package.json** file, because Angular apps and Angular itself depend upon functionality and features provided by **npm packages**.

Now we can run our new Angular app by using:

```
npm start
```



Yes! Our app is running!

As you see, this app is slightly different from the Angular CLI app, it has some other configuration files:

- tsconfig.json
- typings.json
- systemjs.config.js

If you wish to learn more about these files and create the app manually from scratch, you may read the quick start guide:

https://angular.io/docs/ts/latest/quickstart.html

Chapter 1 Summary

In this chapter, you've learned how to create a simple Angular application by using different methods. The app is still simple, but this is your first step to create amazing apps in the future. Let's see what you have known so far:

- We've learned about Angular CLI.
- You can now create a simple Angular app.
- You've understood the structure of an Angular app.
- You've learned one of the most important Angular features: NgModule.
- You can be able to create components and other files using Angular Shortcuts.

The next chapter is where all the fun begin! We will learn to create a simple image gallery using Angular!

Chapter 2 - Building An Image Gallery with Angular

In this chapter, we will build a simple image gallery to learn about components, data binding, directives, services, routing and other basic features of Angular.

Creating a new Angular app

Let's start by creating a new Angular app. Our app will be called **nggallery** (Angular Gallery):

```
ng new nggallery --prefix ng
```

```
Code  ng new nggallery --prefix ng
installing ng
  create .editorconfig
  create README.md
  create src/app/app.component.css
  create src/app/app.component.html
  create src/app/app.component.spec.ts
  create src/app/app.component.ts
  create src/app/app.module.ts
  create src/assets/.gitkeep
  create src/environments/environment.prod.ts
  create src/environments/environment.ts
  create src/favicon.ico
  create src/index.html
  create src/main.ts
  create src/polyfills.ts
  create src/styles.css
  create src/test.ts
  create src/tsconfig.app.json
  create src/tsconfig.spec.json
  create src/typings.d.ts
  create .angular-cli.json
  create e2e/app.e2e-spec.ts
  create e2e/app.po.ts
  create e2e/tsconfig.e2e.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tsconfig.json
  create tslint.json
Successfully initialized git.
Installing packages for tooling via npm.
Installed packages for tooling via npm.
Project 'nggallery' successfully created.
```

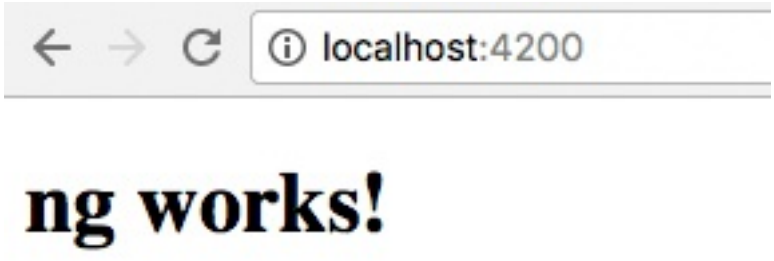
Remember the **--prefix** flag? By adding the **--prefix ng** flag, Angular will use **ng** as the prefix for our selectors.

Next step, let's navigate to the app directory and start the app:

```
cd nggallery

ng serve
```

Now we may visit the app in our browser:

http://localhost:4200



We should see **"ng works!"**

> **Note:** If you see: "**Port 4200 is already in use. Use '--port' to specify a different port**.". You may have another app running, you can close the console tab to stop the app, or go to that app and press **CTRL + C.**

Integrating Twitter Bootstrap

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

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

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

You can get Bootstrap and read its official documentation here:

http://getbootstrap.com

> **Note:** The current **stable version** of Bootstrap is **3.3.7**, which we'll be using in this book. You may use a newer version if you want.

There are three main ways to integrate Bootstrap:

- Using Bootstrap CDN
- Using Precompiled Bootstrap Files
- Using Bootstrap Source Code (Less or Sass)

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

Next, let's open the **src/index.html** file:

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Nggallery</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <ng-root>Loading...</ng-root>
</body>
</html>
```

Find:

```
</head>
```

Place these links above:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/boots
trap.min.css">
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/boots
trap-theme.min.css">
```

These are **Bootstrap's CSS files**, which are used for styling our application.

You may find these links at:

http://getbootstrap.com/getting-started

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

Creating a navigation bar for our app

Every application usually needs a navigation bar, right?

We can use Bootstrap to create a nice navigation bar easily.

First, let's create a new component called **navbar**.

```
ng g c navbar -is --flat
```

> **Note:** Be sure to run this command at the root of our app (nggallery). You may need to create a new tab/window to run the command.

As you see, I've used a new **-is** (inline-style) flag here. When using this flag, the **navbar.component.css** file will not be generated. We're using Bootstrap CSS, so we don't have to create another CSS file for our navbar (navigation bar).

Here is a little tip. If you want to know all the available commands and shortcuts when generating Angular files (components, directives, classes, etc.), you may use this command:

```
ng generate c --help
```

```
nggallery [master] ⚡ng generate c --help

Available blueprints
  class <name> <options...>
    aliases: cl
    --spec (Boolean) Specifies if a spec file is generated.
    --app (String) Specifies app name to use.
      aliases: -a <value>
  component <name> <options...>
    aliases: c
    --flat (Boolean) Flag to indicate if a dir is created.
      aliases: -flat
    --inline-template (Boolean) Specifies if the template will be in the ts file.
      aliases: -it, --inlineTemplate
    --inline-style (Boolean) Specifies if the style will be in the ts file.
      aliases: -is, --inlineStyle
    --prefix (String) (Default: null) Specifies whether to use the prefix.
      aliases: --prefix <value>
    --spec (Boolean) Specifies if a spec file is generated.
      aliases: -spec
    --view-encapsulation (String) Specifies the view encapsulation strategy.
      aliases: -ve <value>, --viewEncapsulation <value>
    --change-detection (String) Specifies the change detection strategy.
      aliases: -cd <value>, --changeDetection <value>
    --skip-import (Boolean) (Default: false) Allows for skipping the module import.
      aliases: --skipImport
    --module (String) Allows specification of the declaring module.
      aliases: -m <value>, --module <value>
    --export (Boolean) (Default: false) Specifies if declaring module exports the component.
      aliases: --export
    --app (String) Specifies app name to use.
      aliases: -a <value>, -app <value>
  directive <name> <options...>
    aliases: d
    --flat (Boolean) Flag to indicate if a dir is created.
    --prefix (String) (Default: null) Specifies whether to use the prefix.
    --spec (Boolean) Specifies if a spec file is generated.
    --skip-import (Boolean) (Default: false) Allows for skipping the module import.
    --module (String) Allows specification of the declaring module.
      aliases: -m <value>
    --export (Boolean) (Default: false) Specifies if declaring module exports the component.
    --app (String) Specifies app name to use.
      aliases: -a <value>
```

Next step, let's open **navbar.component.html**, replace eveything with:

src/app/navbar.component.html

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

```
      <a class="navbar-brand" href="#">Angular Book</a>
    </div>

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

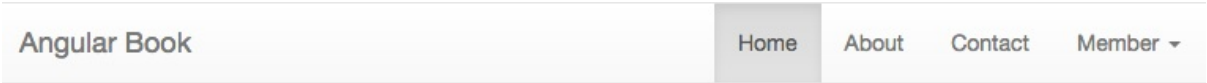
> **Note:** This is just a basic Bootstrap navbar, which can be found at:
>
> http://getbootstrap.com/components/#navbar

We've just created a new navbar! But nothing changes, because we don't place the navbar inside the **main** component (**app.component.html**) yet.

To display the navbar, open **app.component.html**, and then replace everything with:

```
<ng-navbar></ng-navbar>
```

Well done! Visit the app to see a nice navigation bar!



Creating our gallery component

Once we have the navbar, it's time to build our gallery by generating a new component:

```
ng g c gallery
```

```
nggallery [master] ⚡ng g c gallery
installing component
  create src/app/gallery/gallery.component.css
  create src/app/gallery/gallery.component.html
  create src/app/gallery/gallery.component.spec.ts
  create src/app/gallery/gallery.component.ts
  update src/app/app.module.ts
```

With this command, a new **gallery** component was created.

Here is the code for **gallery.component.html**:

gallery.component.html

```
<div class="container">
      <div class="row">
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img1.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img2.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img3.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img4.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img5.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img6.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img7.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img8.jpg" /></div></a>
      </div>
</div>
```

As you see, I've just added some images and some Bootstrap classes here. You may use your own images if you want. The thing to notice here is that the size of an image should be **300 x 200**.

To finish off the gallery, let's apply some custom CSS styling to our CSS file (**gallery.component.css**):

gallery.component.css

```
img {
  -webkit-box-shadow: 0px 1px 6px 1px rgba(0,0,0,0.75);
  -moz-box-shadow: 0px 1px 6px 1px rgba(0,0,0,0.75);
  box-shadow: 0px 1px 6px 1px rgba(0,0,0,0.75);
  margin-bottom:20px;
}

img:hover {
  filter: gray; /* IE6-9 */
  -webkit-filter: grayscale(1); /* Google Chrome, Safari 6+ & Opera 15+ */
}
```

Finally, we may display our gallery by adding this tag to our **app.component.html** file:

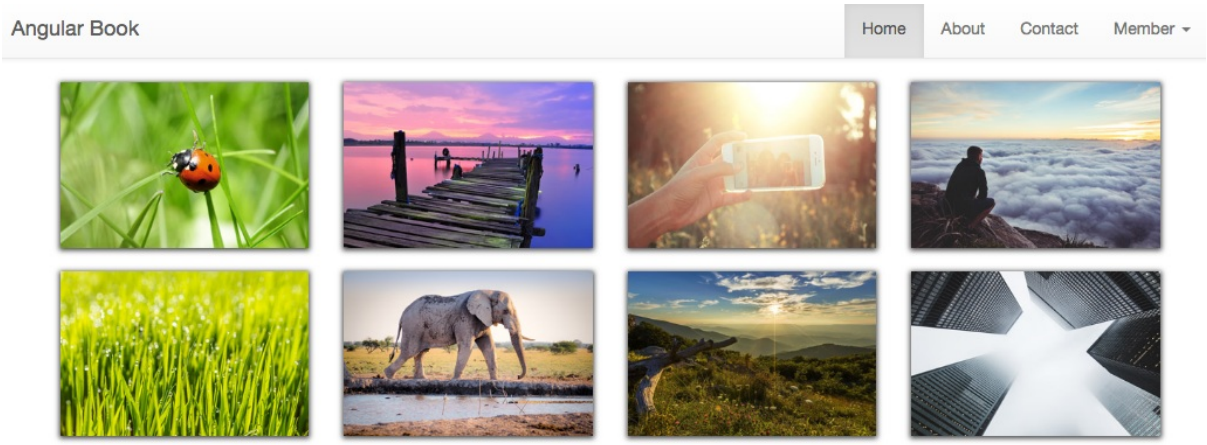
```
<ng-gallery></ng-gallery>
```

The **app.component.html** file should look like this:

app.component.html

```
<ng-navbar></ng-navbar>
<ng-gallery></ng-gallery>
```

Go ahead and refresh the home page, we should see:



Well done! We now have a cool image gallery!

But it's not over yet. We should not put everything in one file, and we should not hard code the image links like that. Instead, let's do this in the Angular way. We will separate the gallery component into two components:

- **image list component:** stores a collection of images.
- **image component:** stores a single image.

Creating image list component

Because the **image list** component is a child component of the **gallery** component, it should be placed within the **gallery** component.

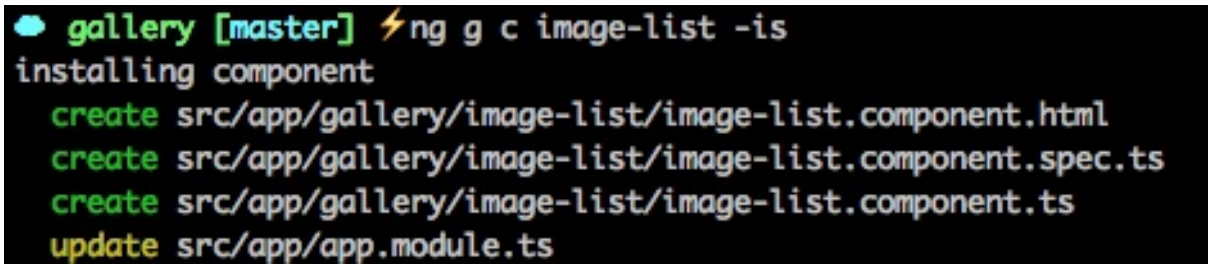
Navigate to the gallery component using:

```
cd src/app/gallery
```

> **Note:** Be sure that you're in the **nggallery** directory. If you're in the **app** directory, you may use: **cd gallery**

And then create the **image list** component there:

```
ng g c image-list -is
```

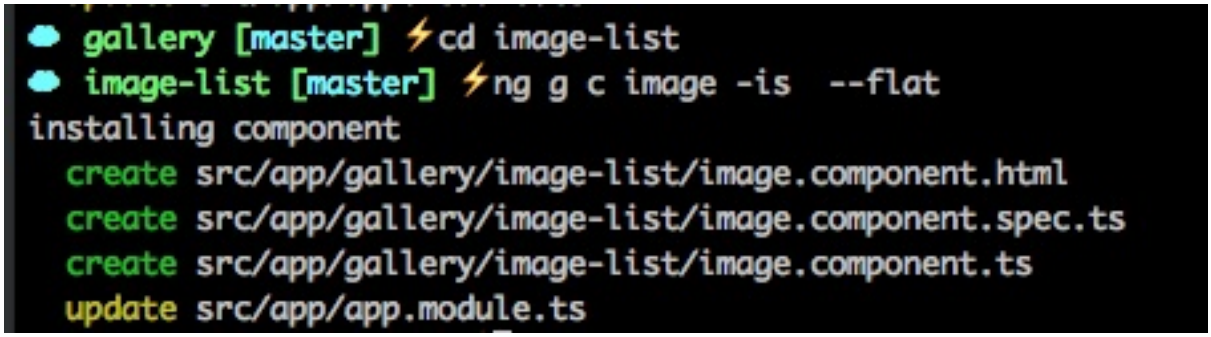


Creating image component

So now that we have the **image list** component. The **image list** component should contain the **image** component, right?

As you may know, we can easily create the **image** component using these commands:

```
cd image-list

ng g c image -is  --flat
```



> **Note:** I don't want to create another folder, so I use the **--flat** flag, you may create a new folder if you want.

Cool. Now we have all the components that we need. However, before editing our components, we should create an **image model** first.

Creating image model

What is the **image model** and why do we need to create it?

In **Object Oriented Programming** (OOP), we usually create multiple objects. Objects can be anything that has properties and actions. For example, a mobile phone can be an object. Each model of a phone has its own blueprint. You can buy a new case for your phone, or you can change its home screen. But no matter you customize it, it's still based on the blueprint that was created by the manufacturer.

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

> **Tip:** Model is usually known as the **M** in the **MVC** system (**Model-View-Controller**). If you're a Laravel developer, I guess you may have already known the concept.

Even though Angular is following **MVC pattern**, there are many data architectures and many different kinds of models that we can use with Angular. In this book, our models will just be **plain Javascript objects**.

Basically, we'll create an **image model** and use it to create images.

Got the concept?

Good! It's time to create our first model!

We may place the model everywhere, but I'd like to place the **image** model in a **models** directory. Therefore, let's navigate back to the **app** directory using this command:

```
cd ../../
```

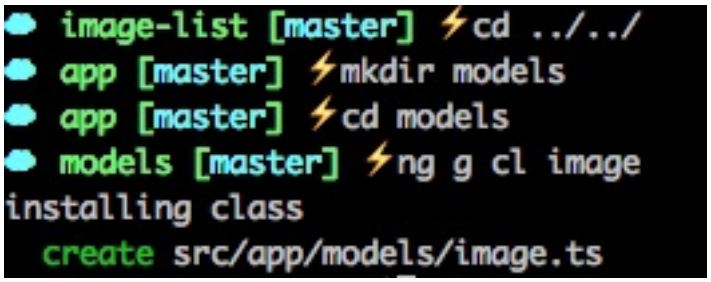
We don't have the **models** directory yet, so let's create it:

```
mkdir models
```

> **Tip:** you may create the directory manually

Go to the **models** directory, and create the **image** model as follows:

```
cd models
ng g cl image
```



A new file called **image.ts** was created. Here is what a basic model looks like:

image.ts

```
export class Image {
  constructor(public id:string, public title:string, public description:string, public
 thumbnail:string, public imageLink:string) {
  }
}
```

Our image model is just a simple class, which has **a constructor** that takes **five arguments**. There are **five public variables**:

- **id**: ID of the image.
- **title:** The title of the image.
- **description:** description of the image.
- **thumbnail:** the image's thumbnail link.
- **imageLink:** the direct link to the image.

When writing **title:string**. We let Angular know that the variable should be **of type String**.

> **Note:** you may just write **public title**, it still works fine.

Our **image model** is now ready to use!

Updating our gallery component

The **image list** component will render a list of images, so let's open **gallery.component.html**, copy the following code:

```
<div class="container">
      <div class="row">
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img1.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img2.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img3.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img4.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img5.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img6.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img7.jpg" /></div></a>
        <a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive"
 src="https://angularbooks.com/img/angular4/img8.jpg" /></div></a>
      </div>
</div>
```

and paste into the **image-list/image-list.component.html**.

Open **gallery.component.html** again, replace everything with this selector to display the **image list** component:

```
<ng-image-list></ng-image-list>
```

Next, open **image-list.component.html**, update it as follows:

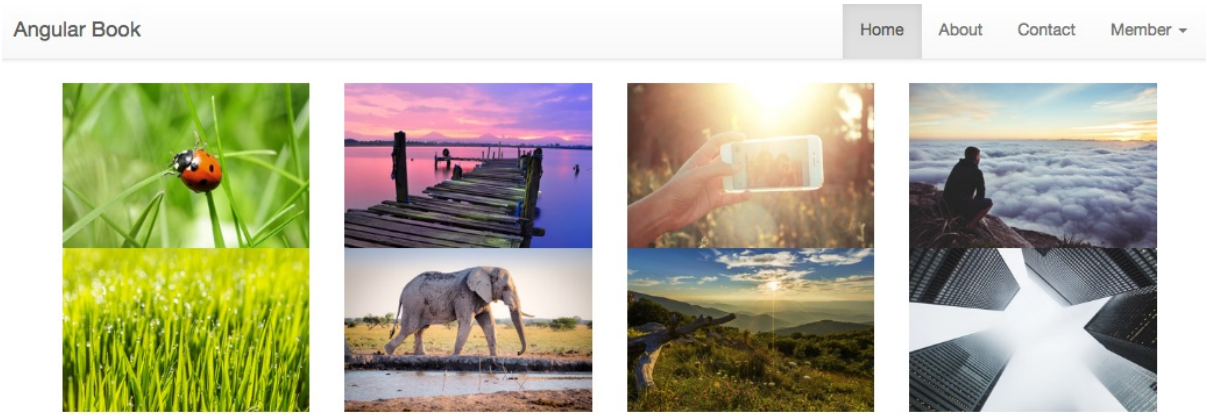
```
<div class="container">
      <div class="row">
           <ng-image></ng-image>
      </div>
</div>
```

As you see, we replace all images with the **ng-image** tag.

Now the **image** component will render the images! Update the **image.component.html** as follows:

```
<a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive" src="h
ttps://angularbooks.com/img/angular4/img1.jpg" /></div></a>
<a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive" src="h
ttps://angularbooks.com/img/angular4/img2.jpg" /></div></a>
<a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive" src="h
ttps://angularbooks.com/img/angular4/img3.jpg" /></div></a>
<a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive" src="h
ttps://angularbooks.com/img/angular4/img4.jpg" /></div></a>
<a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive" src="h
ttps://angularbooks.com/img/angular4/img5.jpg" /></div></a>
<a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive" src="h
ttps://angularbooks.com/img/angular4/img6.jpg" /></div></a>
<a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive" src="h
ttps://angularbooks.com/img/angular4/img7.jpg" /></div></a>
<a href="#"><div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive" src="h
ttps://angularbooks.com/img/angular4/img8.jpg" /></div></a>
```

If we visit our app, we should see:



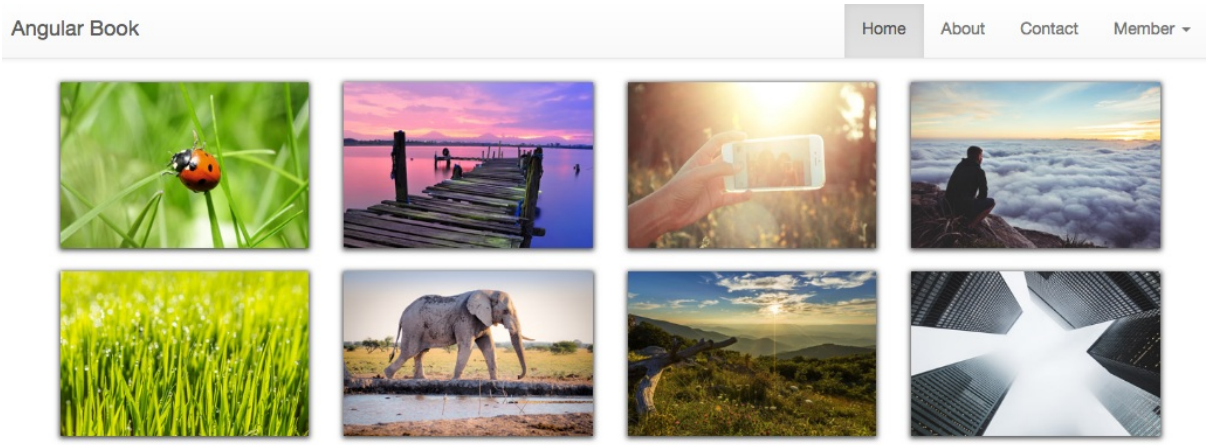
Our CSS rules are not working! Because our HTML code are now placed within the **image.component.html**. If we want to apply CSS stylings to the images, we should write our CSS code in **image.component.css** or **image.component.ts**.

As you may know, we can write our CSS rules inside our **ts** file like this:

image.component.ts

```
@Component({
  selector: 'ng-image',
  templateUrl: './image.component.html',
  styles: [`
    img {
      -webkit-box-shadow: 0px 1px 6px 1px rgba(0,0,0,0.75);
      -moz-box-shadow: 0px 1px 6px 1px rgba(0,0,0,0.75);
      box-shadow: 0px 1px 6px 1px rgba(0,0,0,0.75);
      margin-bottom:20px;
    }

    img:hover {
      filter: gray; /* IE6-9 */
      -webkit-filter: grayscale(1); /* Google Chrome, Safari 6+ & Opera 15+ */
    }
  `]
})
```

Now everything is working fine:



However, something is still not right. We should not store the image links in the **image.component.html** file, and we need a better way to handle our images.

Remember our **image** model? It's time to use it to create some images.

Open **image-list.component.ts** and find:

```
import { Component, OnInit } from '@angular/core';
```

Add below:

```
import { Image } from '../../models/image';
```

We use **import** to import the **image** model into our **image list** component. Now we can use the **image** model to create a new image.

Find:

```
export class ImageListComponent implements OnInit {
```

Add below:

```
image = new Image ('First image', 'First image description', 'https://angularbooks.com
/img/angular4/img1.jpg', 'https://angularbooks.com/img/angular4/img1-l.jpg');
```

Here is the updated **image-list.component.ts**:

image-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Image } from '../../models/image';

@Component({
  selector: 'ng-image-list',
  templateUrl: './image-list.component.html',
  styles: []
})
export class ImageListComponent implements OnInit {
  image = new Image ('1', 'First image', 'First image description', 'https://angularbo
oks.com/img/angular4/img1.jpg', 'https://angularbooks.com/img/angular4/img1-l.jpg');

  constructor() { }

  ngOnInit() {
  }

}
```

Done! You've just created your first image!

Next, open **image-list.component.html**, and find:

```
<ng-image></ng-image>
```

Modify to:

```
<ng-image [image]="image"></ng-image>
```

This is the key feature of Angular: **input binding** (aka input property). We use the **square brackets** (**[ ]**) to pass data into child components. In this case, we use **[image]** to send data (the **image**) to our **image** component.

As you may notice, we have to update the **image** component so that it can get the data (the image) from the **image list** component.

Open **image.component.ts**, and find:

```
import {Component, OnInit} from '@angular/core';
```

Modify to:

```
import {Component, OnInit, Input} from '@angular/core';
```

```
import { Image } from '../../models/image';
```

In Angular, we may use the **@Input** annotation (aka **@Input** decorator) to tell a component to **take an input** from another component. To use the **@Input** annotation, we have to import **Input** into our component first.

We also use the **image** model in this component, so we have to import it as well.

Now we can get the data from the **image list** component like this:

```
export class ImageComponent implements OnInit {
  @Input() image: Image;

  constructor() { }

  ngOnInit() {
  }

}
```

Here is the updated **image.component.ts**:

image.component.ts

```
import {Component, OnInit, Input} from '@angular/core';
import { Image } from '../../models/image';

@Component({
  selector: 'ng-image',
  templateUrl: './image.component.html',
  styles: [`
    img {
      -webkit-box-shadow: 0px 1px 6px 1px rgba(0,0,0,0.75);
      -moz-box-shadow: 0px 1px 6px 1px rgba(0,0,0,0.75);
      box-shadow: 0px 1px 6px 1px rgba(0,0,0,0.75);
      margin-bottom:20px;
    }

    img:hover {
      filter: gray; /* IE6-9 */
      -webkit-filter: grayscale(1); /* Google Chrome, Safari 6+ & Opera 15+ */
    }
  `]
})
export class ImageComponent implements OnInit {
  @Input() image: Image;
  constructor() { }
```

```
  ngOnInit() {
  }

}
```

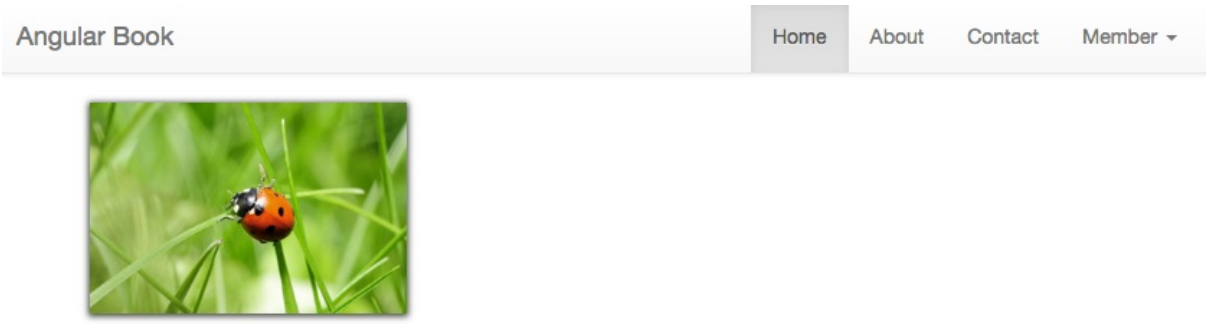
Once we have the image, we may use it in our view (**image.component.html**). Let's change our template to:

image.component.html

```
<a href="#">
  <div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive" src="{{image.thu
mbnail}}" /></div>
</a>
```

The `{{ }}` syntax is another important feature of Angular. We call it **template binding**. In this case, we just use `{{image.thumbnail}}` to display the thumbnail's link of the image.

Go ahead and take a look at your app:



Great! Our image is displayed!

Creating a list of images

Currently, we only show one single image because we have just created one image. If we want to create a list of images, we may write like this:

Find:

```
image = new Image ('1', 'First image', 'First image description', 'https://angularbook
s.com/img/angular4/img1.jpg', 'https://angularbooks.com/img/angular4/img1-l.jpg');
```

Replace with:

```
images: Image[] = [
  new Image('1', 'First image', 'First image description', 'https://angularbooks.com/img/angular4/img1.jpg', 'https://angularbooks.com/img/angular4/img1-l.jpg'),
  new Image('2', 'Second image', 'Second image description', 'https://angularbooks.com/img/angular4/img2.jpg', 'https://angularbooks.com/img/angular4/img2-l.jpg'),
  new Image('3', 'Third image', 'Third image description', 'https://angularbooks.com/img/angular4/img3.jpg', 'https://angularbooks.com/img/angular4/img3-l.jpg'),
  new Image('4', 'Fourth image', 'Fourth image description', 'https://angularbooks.com/img/angular4/img4.jpg', 'https://angularbooks.com/img/angular4/img4-l.jpg'),
  new Image('5', 'Fifth image', 'Fifth image description', 'https://angularbooks.com/img/angular4/img5.jpg', 'https://angularbooks.com/img/angular4/img5-l.jpg'),
  new Image('6', 'Sixth image', 'Sixth image description', 'https://angularbooks.com/img/angular4/img6.jpg', 'https://angularbooks.com/img/angular4/img6-l.jpg'),
  new Image('7', 'Seventh image', 'Seventh image description', 'https://angularbooks.com/img/angular4/img7.jpg', 'https://angularbooks.com/img/angular4/img7-l.jpg'),
  new Image('8', 'Eighth image', 'Eighth image description', 'https://angularbooks.com/img/angular4/img8.jpg', 'https://angularbooks.com/img/angular4/img8-l.jpg'),
];
```

The variable **image** has been changed to **images**, and its type is **Image[ ]**. When using **[ ]**, we let Angular know that this is an **array of images**.

Displaying multiple images using ngFor

Once we have a list of images, we will update our template to render the images. In Angular, we will use a **directive** called **ngFor** to iterate over the list of images and generate a new tag for each one.

Open **image-list.component.html**, and find:

```
<ng-image [image]="image"></ng-image>
```

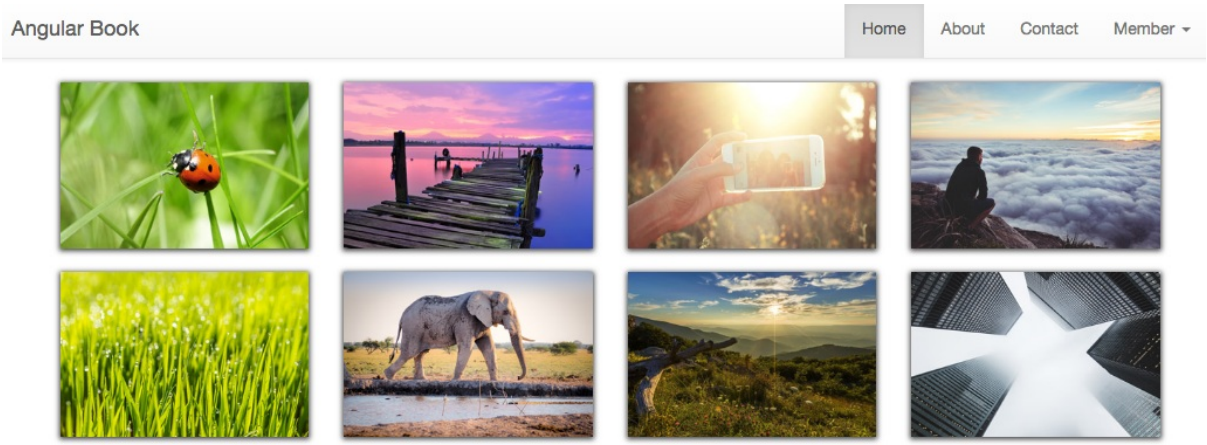
Replace with:

```
<ng-image *ngFor="let image of images" [image]="image"></ng-image>
```

As you notice, we use ***ngFor** to tell Angular that we want to use **ngFor** directive on the **ng-image** tag.

> **Tip:** the **ngFor** directive works just like the **for loop** or the **foreach loop**.

Check our app again in the browser and we should see all images:



All about directives

Now that you know how to use the **ngFor** directive. In Angular, there are three kinds of directives:

- **Components:** components - that we've been creating - are actually directives.
- **Structural directives:** these directives change the structure of the view. **ngFor** is a **structural directive**, because it changes the DOM elements. Two other directives are **ngIf** and **ngSwitch**.
- **Attribute directives:** these directives are used to change the appearance or behavior of an element. We usually use them to change the styles (color, background-color, border, etc.) of the elements. The two main attribute directives are **ngClass** and **ngStyle**.

If you want to learn more about directives, you may take a look at the official documentation:

https://angular.io/docs/ts/latest/guide/attribute-directives.html#!#directive-overview

We will use some directives in this book later.

Introducing services

Our app is running fine, but there are two problems:

- Creating images in the **image list** component is not the right way to do.
- What if we want to share the images with other components and views?

In Angular, we may create a **service** and use it to manage our list of images. The service can also be used across all components.

Let's navigate to our **app** directory:

```
cd ../
```

and create a new directory called **services**:

```
mkdir services
```

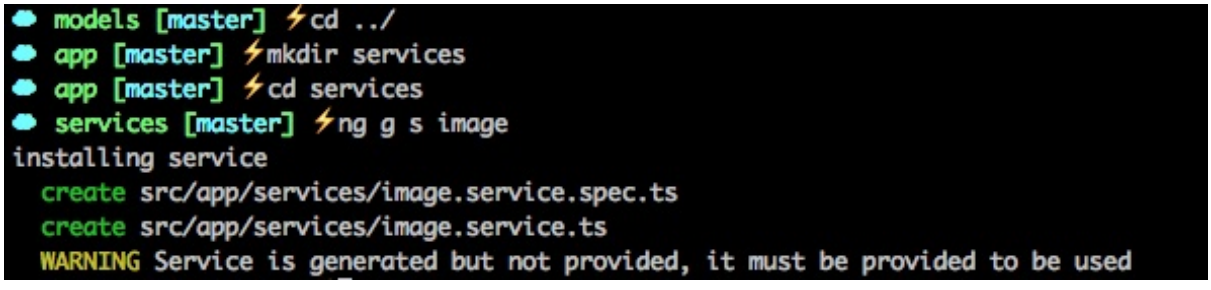
Then go to the **services** directory:

```
cd services
```

> **Note:** from now on, I won't show you how to navigate to a directory anymore to avoid possible confusion. Please try to use the **cd** command and master it!

and run this command to create a new service:

```
ng g s image
```



A new **image service** will be created.

```
WARNING Service is generated but not provided, it must be provided to be used
```

Did you see this warning message?

To use this **image service**, we have to open **app.module.ts**, and find:

```
providers: [],
```

Update to:

```
providers: [ImageService],
```

And don't forget to import the **ImageService** at the top of the file:

```
import {ImageService} from './services/image.service';
```

Now, instead of putting the array of images into **image-list.component.ts**, we can put it in the new **image service**:

image.service.ts

```
import { Injectable } from '@angular/core';
import {Image} from '../models/image';

@Injectable()
export class ImageService {

  images: Image[] = [
    new Image('1', 'First image', 'First image description', 'https://angularbooks.com
/img/angular4/img1.jpg', 'https://angularbooks.com/img/angular4/img1-l.jpg'),
    new Image('2', 'Second image', 'Second image description', 'https://angularbooks.c
om/img/angular4/img2.jpg', 'https://angularbooks.com/img/angular4/img2-l.jpg'),
    new Image('3', 'Third image', 'Third image description', 'https://angularbooks.com
/img/angular4/img3.jpg', 'https://angularbooks.com/img/angular4/img3-l.jpg'),
    new Image('4', 'Fourth image', 'Fourth image description', 'https://angularbooks.c
om/img/angular4/img4.jpg', 'https://angularbooks.com/img/angular4/img4-l.jpg'),
    new Image('5', 'Fifth image', 'Fifth image description', 'https://angularbooks.com
/img/angular4/img5.jpg', 'https://angularbooks.com/img/angular4/img5-l.jpg'),
    new Image('6', 'Sixth image', 'Sixth image description', 'https://angularbooks.com
/img/angular4/img6.jpg', 'https://angularbooks.com/img/angular4/img6-l.jpg'),
    new Image('7', 'Seventh image', 'Seventh image description', 'https://angularbooks
.com/img/angular4/img7.jpg', 'https://angularbooks.com/img/angular4/img7-l.jpg'),
    new Image('8', 'Eighth image', 'Eighth image description', 'https://angularbooks.c
om/img/angular4/img8.jpg', 'https://angularbooks.com/img/angular4/img8-l.jpg'),
  ];

  constructor() { }

  getImages() {
    return this.images;
```

```
  }
}
```

When modifying the file, we also add a new function called **getImages()** so that we can fetch and display our images in other places:

```
getImages() {
  return this.images;
}
```

One more thing to notice is the **@Injectable()** decorator. When creating a new service, it's a best practice to put the **@Injectable()** decorator at the top of the file so that we can use this service in other components.

Using our first service

In order to make use of the **image service**, we need to import it first:

Open **image-list.component.ts**, and find:

```
import { Component, OnInit } from '@angular/core';
import { Image } from '../../models/image';
```

Add below:

```
import {ImageService} from '../../services/image.service';
```

Now we can inject the **image service** into our **image list** component's constructor:

```
constructor(private imageService: ImageService) { }
```

Next, we will create an empty array of images:

```
    images: Image[] = [];
```

And then we can call the service and get all images:

```
ngOnInit() {
  this.images = this.imageService.getImages();
```

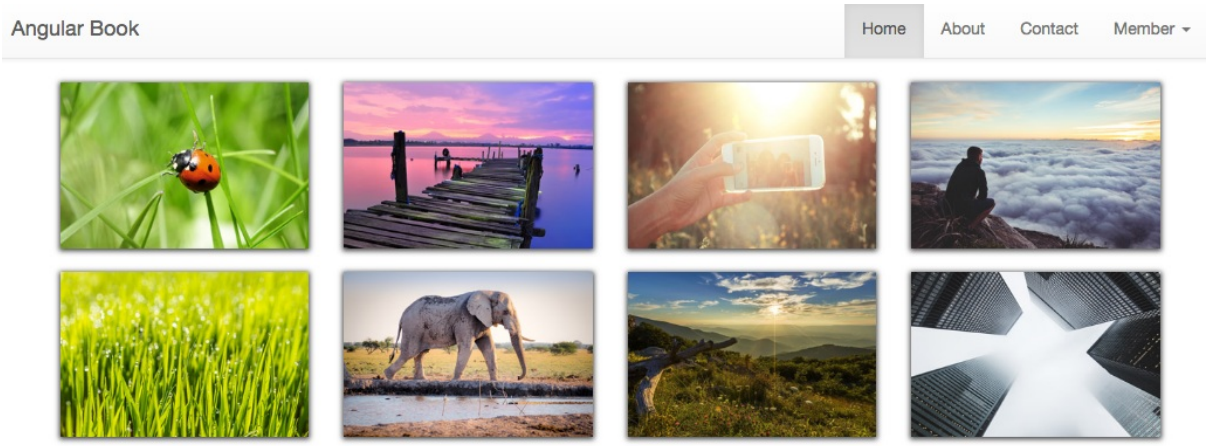
```
}
```

Here is the full **image-list.component.ts**:

image-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Image } from '../../models/image';
import {ImageService} from '../../services/image.service';

@Component({
  selector: 'ng-image-list',
  templateUrl: './image-list.component.html',
  styles: []
})
export class ImageListComponent implements OnInit {
  images: Image[] = [];

  constructor(private imageService: ImageService) { }

  ngOnInit() {
    this.images = this.imageService.getImages();
  }

}
```

Be sure to check our app again:



We still see all the images!

Notice the **ngOnInit** method? In Angular, we call it **ngOnInit Lifecycle Hook**.

All about Lifecycle Hooks

When Angular creates, changes and destroy a component or a directive, it will call some methods. These methods are called **lifecycle hooks**.

All components and directives also have **lifecycle hooks**. Here are all lifecycle events of a component/directive:

- **constructor:** The constructor is always called first.
- **ngOnInit:** This lifecycle hook is called after the constructor.
- **ngOnchanges:** When the value of an input is changed, this lifecycle hook is executed.
- **ngDoCheck:** When the input properties are checked, this method will be called.
- **ngAfterContentInit:** When the content of the component is initialized, this hook is called. A component-only hook.
- **ngAfterContentChecked:** Called after the content is checked. A component-only hook.
- **ngAfterViewInit:** After a view is initialized, this hook is called. A component-only hook.
- **ngAfterViewChecked:** After a view is checked, this hook is called. A component-only hook.
- **ngOnDestroy:** This method is called before Angular destroy the component/directive.

We can use a **lifecycle hook** to response with **certain actions** when the lifecycle state changes. For example, we've used **ngOnInit()** to tell Angular to use the **image service** to get all images, when the **image list** component is initialized.

Remember that, when using a **lifecycle hook**, we should add its **lifecycle hook interface** as well. For example, to use **ngOnInit**, we should add the **OnInit interface** to the component:

```
import { OnInit } from '@angular/core';

export class ImageListComponent implements OnInit {
  ngOnInit() {
  }
}
```

> **Note:** Adding interfaces is not required (optional), but it's good practice.

To learn more about lifecycle hook, you may visit:

https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html#!#docheck

Creating image detail component

Our gallery isn't yet interactive; we cannot click on an image to view its full size and other information (such as title, description, etc.).

Let's hook up our first bit of interactivity by adding an **image detail** component.

Go to the **gallery** directory and run this command:

```
ng g c image-detail
```

A new **image detail** component has been created!

User Input Events

User actions such as clicking a button, entering a text trigger **DOM events**. Angular has **event bindings** that we can use to respond to any **DOM event**.

This section will show you how we can use **click event** to display a **full-size** image when a user clicks on its **thumbnail**.

In order to do this, open **image-list.component.html** and find:

```
<ng-image *ngFor="let image of images" [image]="image"></ng-image>
```

Change to:

```
<ng-image *ngFor="let image of images" [image]="image" (click)="onSelect(image)"></ng-image>
```

Every time the user clicks on the image, it calls the **onSelect** method. In Angular, this technique is known as **output binding**. The syntax is:

```
(output) = "action"
```

> **Tip:** We use **[ ]** for **input binding**, and **( )** for **output binding**.

We usually use **output binding** to response to **an event**.

We'll talk more about **events** later. For now, let's create the **onSelect** method:

Open **image-list.component.ts**, and find:

```
images: Image[] = [];
```

Add below:

```
selectedImage: Image;
```

Find:

```
ngOnInit() {
  this.images = this.imageService.getImages();
}
```

Add below:

```
onSelect(image: Image) {
  this.selectedImage = image;
}
```

This method is simple. We have a new **selectedImage** variable, which is of type **Image**. When we click on an image, the **onSelect** method sets the **selectedImage property** to the **image** that we click.

Once we have the **selected image**, we can send it to our **child component** (**image-detail**) and display it.

Open **image-list.component.html**, and add the following at the top of the file:

```
<ng-image-detail [selectedImage]="selectedImage"></ng-image-detail>
```

As you know, to receive the **selected image**, we will use **@Input** annotation.

Open **image-detail.component.ts**, and find:

```
export class ImageDetailComponent implements OnInit {
```

Add below:

```
@Input() selectedImage: Image;
```

Don't forget to import *Input **and** Image**:

```
import {Component, Input, OnInit} from '@angular/core';
import {Image} from '../../models/image';
```

Here is the template for **image-detail.component.html**:

image-detail.component.html

```
<div *ngIf="selectedImage">
  <div class="container image-detail">
    <div class="row">
      <div class="col-sm-8 col-xs-12">
        <img [src]="selectedImage.imageLink" class="img-responsive">
      </div>
      <div class="col-sm-4 col-xs-12">
        <h1>{{selectedImage.title}}</h1>
        <p>{{selectedImage.description}}</p>
      </div>
    </div>
  </div>
</div>
```

image-detail.component.css

```
.image-detail {
  margin: 20px auto;
}
```

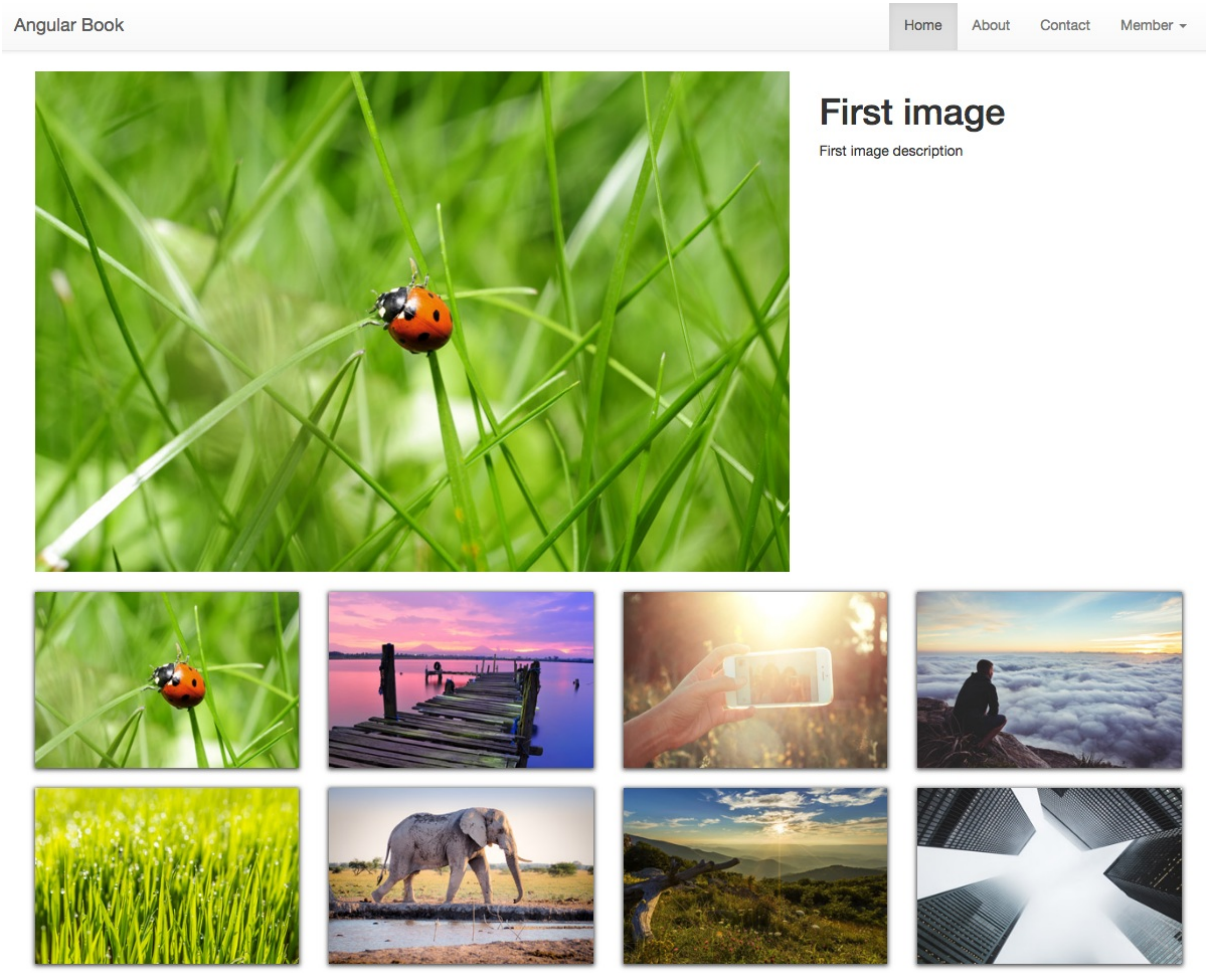
If we don't want to see any full-size image before clicking on an image, we may use **ngIf** directive to hide the image:

```
<div *ngIf="selectedImage">
```

Because the **selectedImage** variable is empty when the app is loaded, the **image-detail** template will not be displayed.

> **Tip**: **ngIf** is used to show or hide certain sections of the template when certain conditions are met.

Let's take a look at our gallery:



Try to click on an image, you should be able to view its full-size image and its details.

Here are some new tips. Instead of writing like this:

```
<img src="{{selectedImage.imageLink}}" class="img-responsive">
```

We may write this way:

```
<img [src]="selectedImage.imageLink" class="img-responsive">
```

This is a best practice. If our browser loads the template before Angular runs, it would show a broken image if we don't use the **[src]** attribute.

Emitting custom events using EventEmitter

Our app is working fine, but it's not quite right.

Currently, in order to use our **image detail component** (**ng-image-detail**), we have to put it in the **image list** component:

image-list.component.html

```
<ng-image-detail [selectedImage]="selectedImage"></ng-image-detail>

<div class="container image-list">
  <div class="row">
    <ng-image *ngFor="let image of images" [image]="image" (click)="onSelect(image)"></
ng-image>
  </div>
</div>
```

Let's say I want to put it in the **gallery** component. For example:

gallery.component.html

```
<ng-image-detail [selectedImage]="selectedImage"></ng-image-detail>
<ng-image-list></ng-image-list>
```

How can I do that?

Raise a new event

As you know, the **image list** component is a child component of the **gallery** component. We can use **Input** to send data from the **gallery** component to its **child components**, but we can't use **Input** to send data from **child components** back to their **parent component**.

Luckily, as I've mentioned before, we may use the **@Output decorator** to response to an event and send data to where we want.

> **Tip:** We pass data into child components using **Input**, and we send data back to a component's container using **Output**.

In Angular, we can define an event using **EventEmitter** object. Here is how we use it:

Open **image-list.component.ts**, and find:

```
constructor(private imageService: ImageService) { }
```

Add above:

```
@Output() selectedEvent: EventEmitter<Image> = new EventEmitter<Image>();
```

We created **a new instance** of an **EventEmitter** object. The type of the object instance must be **Image**. After that, we attach the **EventEmitter** to an **output property** called **selectedEvent**.

Like what we did with the **@Input** decorator, we're using the **@Output** decorator and **EventEmitter** here, so we have to import them into our component:

```
import {Component, OnInit, EventEmitter, Output} from '@angular/core';
```

Good! We have just created a new custom event (**selectedEvent**)!

> **Note:** you may name the event whatever you want. For example, you can call it **selectRequest**, **imageSelect**, etc.

Now we can use **EventEmitter.emit(payload)** to **fire an event**.

Find:

```
onSelect(image: Image) {
  this.selectedImage = image;
}
```

Modify to:

```
onSelect(image: Image) {
  this.selectedEvent.emit(image);
}
```

That is how we **emit** the **Image** object!

What does it mean? When a user clicks on an image, the **image list** component will dispatch an event (called **selectedEvent**) to its parent components. The parent components can then listen for the event and receive the **payload** (which is the **image**). Simply put, we can send the **image** to parent components using the **selectedEvent**!

Response when the event occurs

Here's how we listen and response to an event:

gallery.component.html

```
<ng-image-detail [selectedImage]="selectedImage"></ng-image-detail>

<ng-image-list (selectedEvent)="selectImage($event)"></ng-image-list>
```

As you see, we use our custom event here, which is **selectedEvent**.

> **Tip:** Similar? The syntax is just like the **click** event!

When the **selectedEvent** fires, we want it to call a new **selectImage** method, passing in the **$event** object. The **$event** object holds our **event payload**, which is the **image** that the user has selected.

Let's create the **selectImage** method:

Open **gallery.component.ts** and find:

```
export class GalleryComponent implements OnInit {
```

Add below:

```
selectedImage: Image;
```

Find:

```
ngOnInit() {
}
```

Add below:

```
selectImage(image: Image) {
  this.selectedImage = image;
}
```

Don't forget to import **Image** because we're using it here:

```
import {Image} from '../models/image';
```

Just like what we've done before, we also created a new **selectedImage** variable, which is of type **Image**. When the **selectedEvent** fires, we assign **the payload** (the **image**) to the **selectedImage** variable.

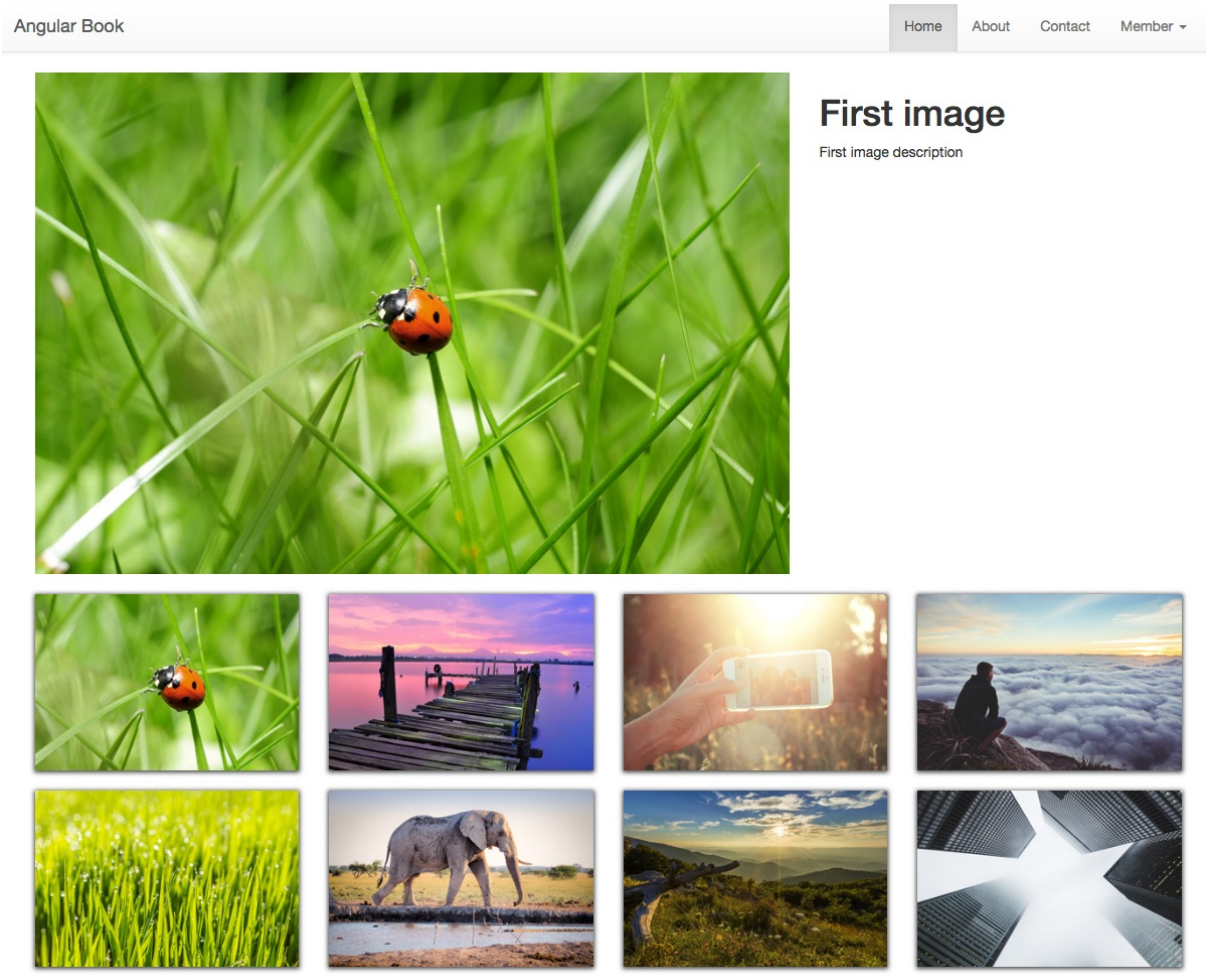
When we have the **selectedImage**, we can send it to our **image detail** component and display its full-size image.

Resulting in the following code:

gallery.component.html

```
<ng-image-detail [selectedImage]="selectedImage"></ng-image-detail>

<ng-image-list (selectedEvent)="selectImage($event)"></ng-image-list>
```

Finally, here's what our current gallery looks like:



If you've managed to get this far, well done!

Angular EventEmitter is a powerful tool. Using it we can create rich, interactive components and send data to where we want.

In the next section, we're going to explore how to navigate and pass data between pages using another powerful Angular feature: **Routing**.

Routing

One of the most crucial features of Angular is **routing**, which is handled by **Angular Router**.

Using **Angular Router**, we can divide our app into **different areas** (**pages**) and tell Angular which **URLs** map to which **functions**. For example, when we go to http://localhost:4200/contact, we can access the **contact** page.

Because Angular apps are **Single Page Apps (SPA)**, we actually have only a single page. To render different pages, Angular uses a technique called **hash-based routing** to load different parts of an application.

As you may know, in HTML, we use **anchor tags** and the **hash mark (#)** to jump to a specific part of a page. Angular and other client-side frameworks come up with a different way. The **anchor tags** and the **hash mark (aka hashtag)** are used as a part our the URL to simulate different content. For example, http://localhost:4200/#/images route leads to displaying a list of images.

Alternatively, Angular also use HTML5 History API to create clean URLs and remove the hashtag from the URL. Instead of seeing http://localhost:4200/#/images, we have a nice URL: http://localhost:4200/images.

However, some old browsers don't support HTML5, so if we want to target more audiences we should use **hash-based** routing.

Creating our first routes

Currently, our app has 3 main pages, so let's create 3 routes:

- Gallery page - A **gallery** page route. The path is: **/#/gallery**. When the user visit our home page (/), it will redirect to the gallery path.
- About page - An **about** page route. The path is: **/#/about**.
- Contact page - A **contact** page route. The path is: **/#contact**.

> **Note:** We'll create the **member** pages later.

First, we don't have the **Contact** Component and **About** Component yet, so let's create them. Go to the **app** directory and run these commands:

```
ng g c contact

ng g c about
```

To define routes, open **app.module.ts** and find:

```
@NgModule({
```

Add above:

```
const appRoutes: Routes = [
  { path: '', redirectTo:'/gallery', pathMatch: 'full'},
  { path: 'gallery', component: GalleryComponent},
  { path: 'contact', component: ContactComponent},
  { path: 'about', component: AboutComponent}
];
```

We just created a new array of routes called **appRoutes**. Each route has **a path** (e.g. **gallery**) and a component (e.g. **GalleryComponent**), which is used to handle the route.

As you see, the first route is a **redirect route**, which is used to redirect a path to an existing route. In this case, when users visit the **root path**(our home page), they will be redirected to the gallery (path: **/gallery**)

```
{ path: '', redirectTo:'/gallery', pathMatch: 'full'},
```

A redirect route must have a **pathMatch** property, which is used to tell how we want to **match a URL** to **the path of a route**. We want that the entire URL should match **' '**, so the **pathMatch** value is **full**.

The other value is **prefix**, which tells the router to match **URL prefixed** with the redirect path.

You may learn more about redirects here.

Because we're using **Routes** and **RouterModule** (from the **Angular Router** module) so we have to import them:

```
import {Routes, RouterModule} from '@angular/router';
```

After that, we can use **RouterModule.forRoot(appRoutes)** to tell Angular that we want to use the **appRoutes** array as our routes.

Find:

```
imports: [
  BrowserModule,
  FormsModule,
  HttpModule
],
```

Update to:

```
imports: [
  RouterModule.forRoot(appRoutes),
  BrowserModule,
  FormsModule,
  HttpModule
],
```

Here is the updated **app.module.ts**:

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
```

```
import { AppComponent } from './app.component';
import { NavbarComponent } from './navbar.component';
import { GalleryComponent } from './gallery/gallery.component';
import { ImageListComponent } from './gallery/image-list/image-list.component';
import { ImageComponent } from './gallery/image-list/image.component';
import {ImageService} from './services/image.service';
import { ImageDetailComponent } from './gallery/image-detail/image-detail.component';
import { ContactComponent } from './contact/contact.component';
import { AboutComponent } from './about/about.component';
import {Routes, RouterModule} from '@angular/router';

const appRoutes: Routes = [
  { path: '', redirectTo:'/gallery', pathMatch: 'full'},
  { path: 'gallery', component: GalleryComponent},
  { path: 'contact', component: ContactComponent},
  { path: 'about', component: AboutComponent}
];

@NgModule({
  declarations: [
    AppComponent,
    NavbarComponent,
    GalleryComponent,
    ImageListComponent,
    ImageComponent,
    ImageDetailComponent,
    ContactComponent,
    AboutComponent
  ],
  imports: [
    RouterModule.forRoot(appRoutes),
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [ImageService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now all the defined routes are activated and ready to use!

Render contents of each route using router-outlet

Once we have all the routes, the next thing that we have to do is to tell Angular where to display contents of each route (where a component will be inserted into a page).

In order to do that, we'll use a special Angular tag called **router-outlet**. Basically, it's just an **Angular directive**, which is used to indicate where contents of each route will be rendered.

Open **app.component.html** and find:

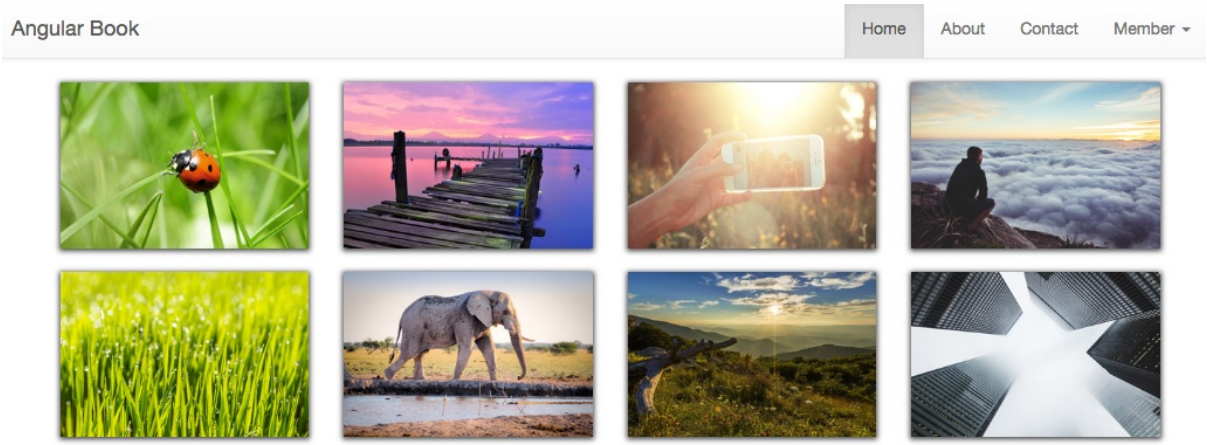
```
<ng-navbar></ng-navbar>
<ng-gallery></ng-gallery>
```

Modify to:

```
<ng-navbar></ng-navbar>
<router-outlet></router-outlet>
```

As you may guess, I want to keep the navigation bar at the top and display the contents below it, so I just replace **ng-gallery** tag with the **router-outlet** tag.

Try to visit the app, you will be redirected to http://localhost:4200/gallery.



This is our gallery page! Notice that we don't use the **ng-gallery** tag but the gallery is still displayed. That means the **GalleryComponent** is loaded properly and our routes are working!

When you visit http://localhost:4200/about, you should see the **about page**:

Navigating to other routes using routerLink

Now that we have our routes working, but we still need to provide our users a way to navigate to other pages. You might think that we can link to other routes like this:

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

Unfortunately, it's not the right way to do, because it triggers a page reload. To link to other routes without reloading the page, we have to use another Angular directive called **routerLink**.

The syntax is simple, take a look at the following:

```
<a [routerLink]="['/about']">About</a>
```

Pretty easy, right?

Let's open **navbar.component.html** and find:

```
<li class="active"><a href="/">Home</a></li>
      <li><a href="/about">About</a></li>
      <li><a href="/contact">Contact</a></li>
```

Replace all the links with:

```
<li class="active"><a [routerLink]="['/']">Home</a></li>
<li><a [routerLink]="['/about']">About</a></li>
<li><a [routerLink]="['/contact']">Contact</a></li>
```

Visit our app and click on the navbar's links to go to other pages. All the links should be working!

Check if the route is active using routerLinkActive

You may notice that when we go to other pages, the **Home** menu is **still active**. How can we make the current route active?

The answer is, we can use the **routerLinkActive** directive to check if the link points to the current route. After that, we can easily **style a menu item** as selected.

Find:

```
<li class="active"><a [routerLink]="['/']">Home</a></li>
<li><a [routerLink]="['/about']">About</a></li>
<li><a [routerLink]="['/contact']">Contact</a></li>
```

Modify to:

```
<li routerLinkActive="active"><a [routerLink]="['/gallery']">Home</a></li>
<li routerLinkActive="active"><a [routerLink]="['/about']">About</a></li>
<li routerLinkActive="active"><a [routerLink]="['/contact']">Contact</a></li>
```

Done! That's how we use the **routerLinkActive** directive.

> **Note:** When visiting the home page, we'll be redirected to the gallery route, so the link of the **Home** menu item should be **/gallery**

The documentation of routerLinkDirective can be found here.

Let's visit our app again and go to another page. We should see that the current menu item is now active.

Fixing the gallery bug

Notice that when clicking on an image thumbnail, the page is reloaded. The problem is, we still use the 'normal HTML link' (with the **href** attribute) in our **image** component:

image.component.html

```
<a href="#">
  <div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive" src="{{image.thu
mbnail}}" /></div>
</a>
```

The easiest way to fix this bug is to remove the **a** tag. However, if you still want to keep the **a** tag, you just need to remove the **href** attribute.

```
<a>
  <div class="col-md-3 col-sm-4 col-xs-6"><img class="img-responsive" src="{{image.thu
mbnail}}" /></div>
</a>
```

If you still want to have the **hand cursor** display over the images, you may use the following CSS code:

Open **image.component.ts**, and find:

```
img:hover {
  filter: gray; /* IE6-9 */
  -webkit-filter: grayscale(1); /* Google Chrome, Safari 6+ & Opera 15+ */
}
```

Add below:

```
a:hover {
  cursor:pointer;
}
```

Now everything should be working fine!

Storing routes in another file

We currently keep our routes in the **app.module.ts** file, which is certainly a good choice for a small app. However, this would be messy when building bigger apps. If we store our routes in another file, every time we need to add, remove or edit a route, it would be more convenient.

So what we are going to do is create a new file called **app.routes.ts** and place it in the **app** folder (**nggallery/src/app**).

We can then move the **appRoutes** (from **app.module.ts**) into this file:

app.routes.ts

```
import {Routes, RouterModule} from "@angular/router";
import {GalleryComponent} from "./gallery/gallery.component";
import {ContactComponent} from "./contact/contact.component";
import {AboutComponent} from "./about/about.component";
import {ModuleWithProviders} from "@angular/core";

const appRoutes: Routes = [
  { path: '', redirectTo:'/gallery', pathMatch: 'full'},
  { path: 'gallery', component: GalleryComponent},
  { path: 'contact', component: ContactComponent},
  { path: 'about', component: AboutComponent}
];

export const routes:ModuleWithProviders = RouterModule.forRoot(appRoutes);
```

We also use **Routes**, **RouterModule** and other components here, so be sure to **import all of them**.

After that, we export our **routes** to use them in another file using:

```
export const routes:ModuleWithProviders = RouterModule.forRoot(appRoutes);
```

The **routes** variable is of type **ModuleWithProviders**, so we have to import **ModuleWithProviders** from **@angular/core** as well.

Once our **routes** have been exported, we can then import them into our **app** module.

Open **app.module.ts**, remove:

```
import {Routes, RouterModule} from '@angular/router';
```

and add this line at the top to import the **routes**:

```
import {routes} from './app.routes';
```

Find:

```
RouterModule.forRoot(appRoutes),
```

Replace with:

```
routes,
```

Here is the updated **app.module.ts** file:

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';
import { NavbarComponent } from './navbar.component';
import { GalleryComponent } from './gallery/gallery.component';
import { ImageListComponent } from './gallery/image-list/image-list.component';
import { ImageComponent } from './gallery/image-list/image.component';
import {ImageService} from './services/image.service';
import { ImageDetailComponent } from './gallery/image-detail/image-detail.component';
import { ContactComponent } from './contact/contact.component';
import { AboutComponent } from './about/about.component';
import {routes} from './app.routes';

@NgModule({
  declarations: [
    AppComponent,
    NavbarComponent,
    GalleryComponent,
    ImageListComponent,
    ImageComponent,
    ImageDetailComponent,
    ContactComponent,
    AboutComponent
  ],
  imports: [
    routes,
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [ImageService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Well done! Let's check our app again. All routes should still be working.

Chapter 2 Summary

Congratulations! We now have a fully responsive gallery! This gallery is simple, and it's not perfect, but you may use all the techniques to build a wide range of applications.

In this chapter, you've learned many things:

- You've known how to integrate Twitter Bootstrap into your Angular app.
- You've learned about template binding. You can now send data to other components.
- You've learned important directives: ngFor, ngIf, etc.
- You've known what Service is.
- Creating a custom event is easy, right?
- You've known how to use Angular Router to handle your app routes.

In the next chapter, we will learn how to create basic APIs using Laravel. After that, we'll use a **Http Client** to communicate with the server. Along the way, we'll learn more about routes and how to use other important features of Angular!

Chapter 3: Building A Backend API With Laravel

In this chapter, we'll learn how to build a backend service for our Angular applications.

> **Note:** Some sections are extracted from the Learning Laravel 5 book and the Laravel 5 Cookbook. If you wish to learn more about Laravel, you may check them out.

What is REST API?

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

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

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

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

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

A typical REST API has these following **constraints**:

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

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

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

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

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

What is Laravel?

There are a number of great frameworks that developers can use to build RESTful APIs. In this book, we'll use Laravel, which is one of the most popular PHP frameworks.

Laravel has many advanced features and it's very easy to learn.

If this is your first time hearing about Laravel, you may visit the official Laravel website to learn more about it.

https://laravel.com

Learning Laravel is also a great resource to learn how to install Laravel and read more tutorials:

https://learninglaravel.net

Installing Laravel Using Homestead

> **Note:** This section can be found in Learning Laravel 5 book. If you know how to install Laravel already, you may skip this section.

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

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

What is Homestead?

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

We call this: "Virtualization."

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

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

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

Here is a list of included software:

- Ubuntu 16.04
- Git
- PHP 7.1
- Nginx
- MySQL
- MariaDB
- Sqlite3
- Postgres

- Composer
- Node (With Yarn, PM2, Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd

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

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

How to install Homestead?

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

There are three steps to install Homestead using this method.

- Step 1: Install VirtualBox
- Step 2: Install Vagrant
- Step 3: Install Homestead

Let's start by installing VirtualBox and Vagrant first.

Step 1 - Installing VirtualBox

First, we need to go to:

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

Choose a VirtualBox for your platform and install it.

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

The **stable release is version 5.1.4**. You can use a newer version if you want, but if you have any problems, try to use this version.

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

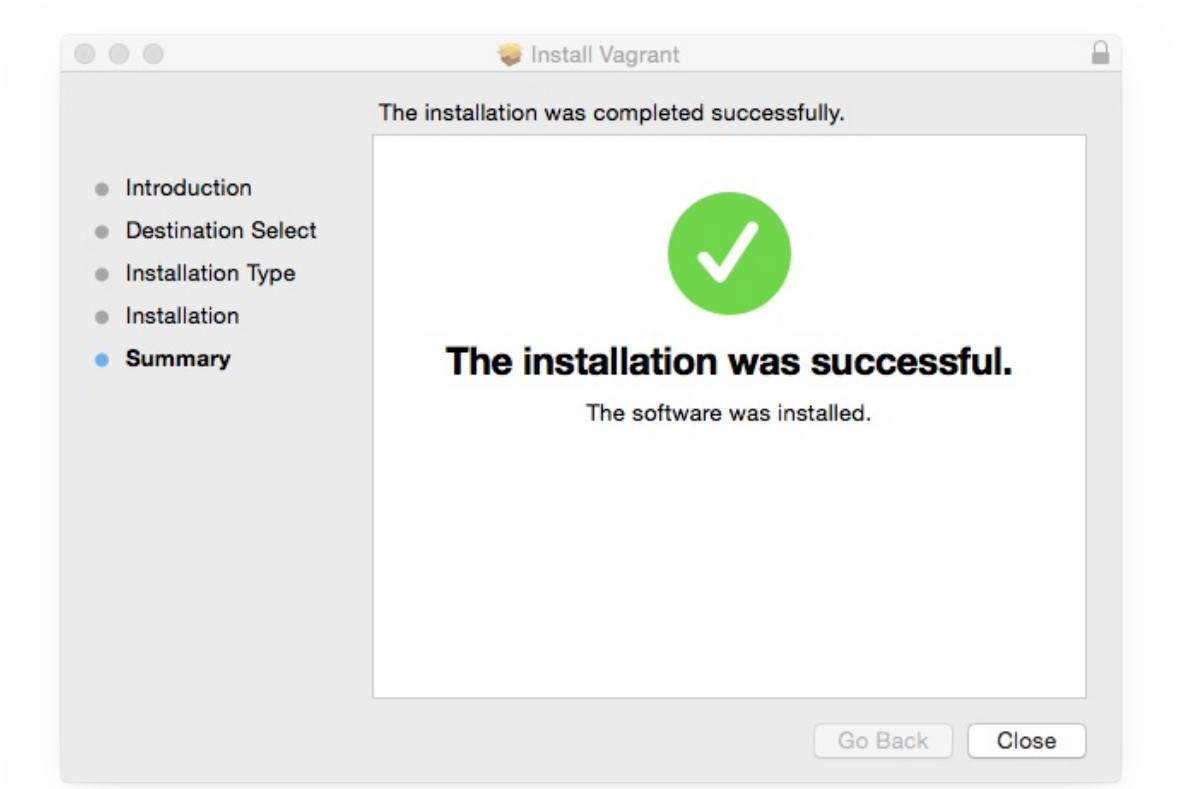
If you're using Mac, simply open the VirtualBox **.dmg** file and click on the **.pkg** file to install.

Step 2 - Installing Vagrant

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

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

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



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

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

Step 3 - Install Homestead (Using Git Clone)

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

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

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

Install Git on Mac

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

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

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

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

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

Install Git on Windows

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

https://windows.github.com

Install Git on Linux/Unix

You can install Git by running this command:

```
sudo yum install git
```

If you're on a Debian-based distribution, use this:

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

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

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

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

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

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

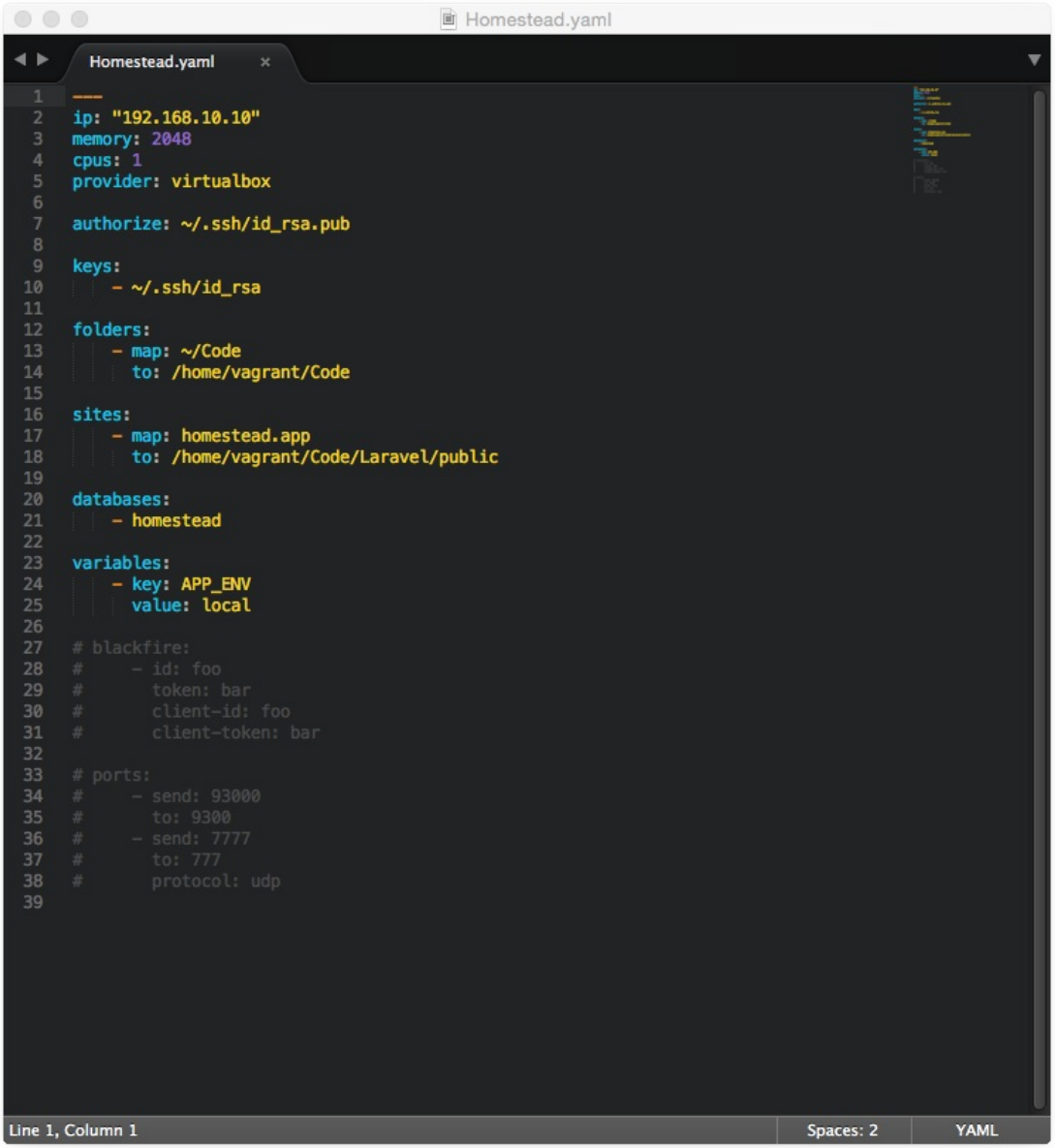
```
cd Homestead
```

Run this command to create `Homestead.yaml` file

```
bash init.sh
```

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

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



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

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

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

```
open ~/Homestead/Homestead.yaml
```

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

Configure Homestead

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

First section - Configure VM

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

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

Second and third section - Configure SSH

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

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

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

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

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

Fourth section - Configure shared folder

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

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

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

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

Fifth section - Map a domain

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

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

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

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

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

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

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

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

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

```
192.168.10.10  homestead.app
```

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

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

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

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

Sixth section - Configure database

```
databases:
    - homestead
```

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

Seventh section - Add custom variables

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

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

Launching Homestead

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

```
vagrant up
```

It may take a few minutes...

If you see this error:

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

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

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

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

```
sudo mkdir ~/Code
```

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

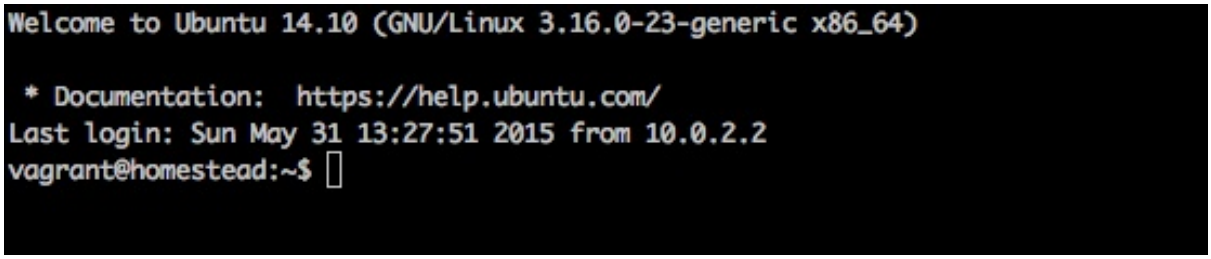
```
chmod -R 0777 ~/Code
```

If everything is going fine, we should see:

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

Now we can access our VM using:

```
vagrant ssh
```



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

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

Excellent! Let's start installing Laravel!

Installing Laravel

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

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

```
cd Code
```

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

Install Laravel Via Laravel Installer

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

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

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

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

```
laravel new nameOfYourSite
```

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

```
laravel new nameOfYourSite --5.4
```

This command is used to download **Laravel 5.4**.

> **Note:** If the latest version of Laravel is **5.4**, you can't run this command. You may install Laravel 5.3 instead by using the **--5.3** flag. It is recommended to use Laravel 5.4 to learn the basics of Laravel Framework. You can upgrade to a newer version later. However, feel free to use the latest version if you want because the book will be updated frequently to support newer versions.

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

For instance, in `Homestead.yam` , we specify the name of our app is `Laravel`

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

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

```
laravel new Laravel
```

You should see this:

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

```
laravel: command not found
```

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

```
nano ~/.bashrc
```

Add this line at end of the file:

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

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

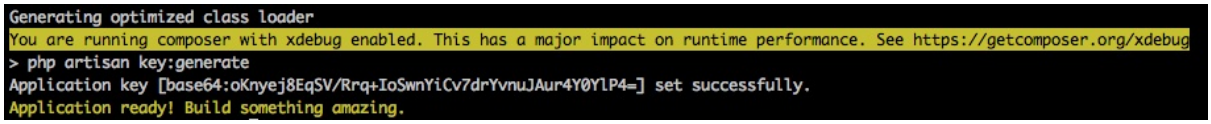
Lastly, run this command:

```
source ~/.bashrc
```

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

```
laravel new Laravel
```

You should see this:



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

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

Install Laravel Via Composer Create-Project

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

```
composer create-project laravel/laravel nameOfYourSite "~5.4.11"
```

This command is used to download **Laravel 5.4.11**, which is a stable version. If you want to use the latest version, use:

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

> **Note:** It is recommended to use Laravel 5.4 to learn the basics of Laravel Framework. You can upgrade to a newer version later. However, feel free to use the latest version if you want because the book will be updated frequently to support newer versions.

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

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

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

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

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

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

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

You should see this:

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

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

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

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

Checking Laravel version

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

```
php artisan --version
```

A line will be printed out:

```
Laravel Framework 5.4.11
```

As you see, I'm using **Laravel 5.4.11**.

Create multiple Laravel apps on Homestead

Creating a new Laravel app

Let's create a new Laravel app. Be sure that you're in the **Code** directory:

```
cd ~/Code
```

Run this command to create a new app:

```
laravel new angularbook
```

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

Activating your new app

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

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

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

```
cd ~/.homestead
```

> **Note:** This directory can be found on your local machine.

And edit the `Homestead.yaml` file:

```
vim Homestead.yaml
```

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

```
open Homestead.yaml
```

Find:

```
sites:
    - map: homestead.app
```

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

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

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

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

Save the file.

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

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

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

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

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

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

```
192.168.10.10  angularbook.app
```

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

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

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

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

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

Good job! If everything is working correctly, we should see our app's home page when visiting http://angularbook.app

Creating a new database

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

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

Default database information

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

host: 127.0.0.1

database: homestead

username: homestead

password: secret

port: 33060 (MySQL) or 54320 (Postgres)

Create a database using the CLI

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

```
mysql -u homestead -p
```

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

To create a new database, use this command:

```
CREATE DATABASE your_database_name;
```

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

I want to create a new database called angularbook, so this is my command:

```
CREATE DATABASE angularbook;
```

To see all databases, run this command:

```
show databases;
```

Finally, you may leave MySQL using this command:

```
exit
```

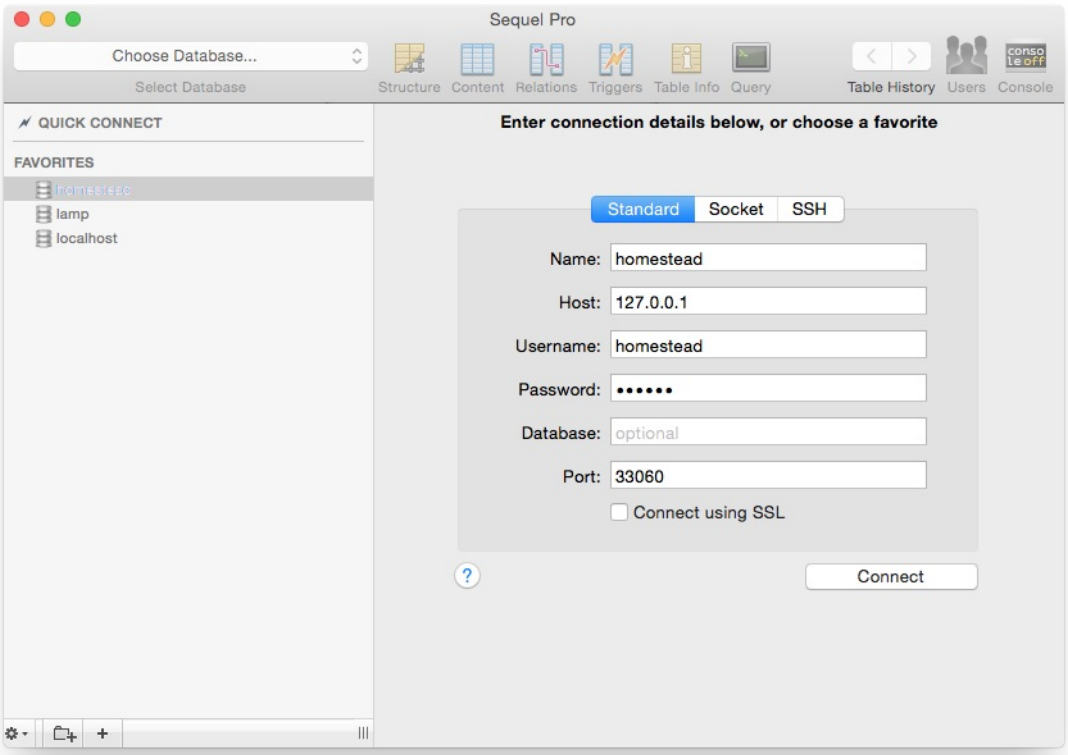
Even though we can easily create a new database via the **CLI**, we should use a **Graphical User Interface (GUI)** to manage databases easily.

Create a database on Mac

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

www.sequelpro.com

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



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

Alternatively, you may use Navicat.

Create a database on Windows

On Windows, three popular GUIs for managing databases are:

SQLYog (Free)

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

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

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

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

HeidiSQL (Free)

http://www.heidisql.com

Navicat

http://www.navicat.com

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

Connecting your app to a database

I'm using a new database called **angularbook**, so I have to update my **.env** file to connect Laravel to that database:

.env file

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

Be sure to update your **.env** file as well if you use a new database. If you want to use the default database (**homestead**), you may skip this step.

Writing APIs with Laravel

Creating our Eloquent model and its migration

To store and manage data in our database, we will create an **Eloquent** model called **Image** and a new **images table**.

In Laravel 5, we can create a new model and generate the **images** migration - which is used to create the **images** table - at the same time by using this command:

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

```
vagrant@homestead:~/Code/angularbook$ php artisan make:model Image -m
Model created successfully.
Created Migration: 2017_02_18_053449_create_images_table
```

Now you should see two new files:

1. app/Image.php: the **Image** model.
2. database/migrations/timestamp_create_images_table.php: the **images** migration file.

It's time to add more columns to our table so that we can store **title**, **description**, **thumbnail**, **imageLink**, and **user_id** (the id of the one that created the image).

Open **timestamp_create_images_table.php** and update the **up()** method as follows:

```
public function up()
{
    Schema::create('images', function (Blueprint $table) {
        $table->increments('id');
        $table->string('title', 255);
        $table->text('description')->nullable();
        $table->string('thumbnail')->nullable();
        $table->string('imageLink')->nullable();
        $table->integer('user_id')->nullable();
        $table->timestamps();
    });
}
```

Next, run this command to create the **images** table:

```
php artisan migrate
```

```
vagrant@homestead:~/Code/angularbook$ php artisan migrate
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
Migrated: 2017_02_18_053449_create_images_table
```

As you see, Laravel also creates **users** table and **password_resets** table for us. We might need these tables in the future, so that's fine.

The last thing to do is open our **Image** model (Image.php), and update it as follows:

Image.php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Image extends Model
{
    protected $fillable = [
        'title', 'description', 'thumbnail', 'imageLink', 'user_id'
    ];
}
```

The **$fillable** property make the columns mass assignable. This is a Laravel feature that helps to prevent **mass-assignment vulnerability**. If you don't add this **$fillable** property, you may see an error when updating your model.

Once we have the **Image** model and the **images** table ready, we can then easily manage our images. Eloquent will help us to do the magic.

> **Note:** Eloquent (aka Eloquent ORM) is a cool feature of Laravel. We may use Eloquent ORM to create, edit, manipulate, deletes our images without writing a single line of SQL!

Seeding our database

We don't have any data yet, so let's create some sample images to test our application. Instead of creating sample data manually, we will user **Faker** - a popular PHP library - to generate fake data automatically.

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

```
php artisan make:seeder ImagesTableSeeder
```

Next, open **imagesTableSeeder** file, and find:

```
class ImagesTableSeeder extends Seeder
```

Add above:

```
use App\Image;
use Faker\Factory as Faker;
```

Update the **run** method as follows:

app/database/seeds/imagesTableSeeder.php

```
public function run()
{
    $faker = Faker::create();
    $imageLinks = array(
        "https://angularbooks.com/img/angular4/img1",
        "https://angularbooks.com/img/angular4/img2",
        "https://angularbooks.com/img/angular4/img3",
        "https://angularbooks.com/img/angular4/img4",
        "https://angularbooks.com/img/angular4/img5",
        "https://angularbooks.com/img/angular4/img6",
        "https://angularbooks.com/img/angular4/img7",
        "https://angularbooks.com/img/angular4/img8",
    );

    foreach($imageLinks as $imageLink)
    {

        Image::create([
            'title' => $faker->text(80),
            'description' => $content = $faker->paragraph(18),
            'thumbnail' => $imageLink.".jpg",
            'imageLink' => $imageLink."-l.jpg",
            'user_id' => $faker->numberBetween($min = 1, $max = 5),
        ]);
    }
}
```

As you may guess, we use **Faker** to generate fake **title**, **description** and **user_id** for us. As we already have the image links (in the last chapter), we will use them here.

> **Tip:** You may use your own links here. You can also use a **multidimensional array** to store image links if you want.

Next step, we have to **activate** our seeder by opening **app/database/seeds/DatabaseSeeder.php**, update the **run** method as follows:

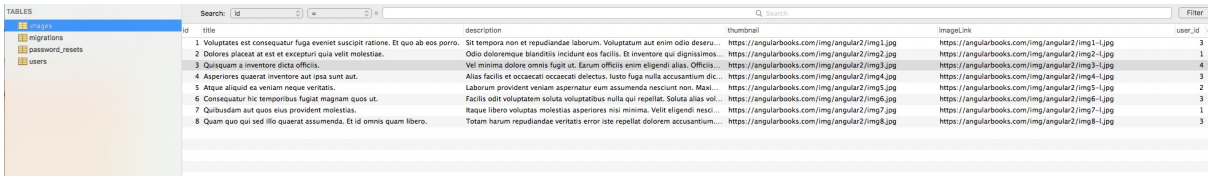
app/database/seeds/DatabaseSeeder.php

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

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

```
php artisan db:seed
```

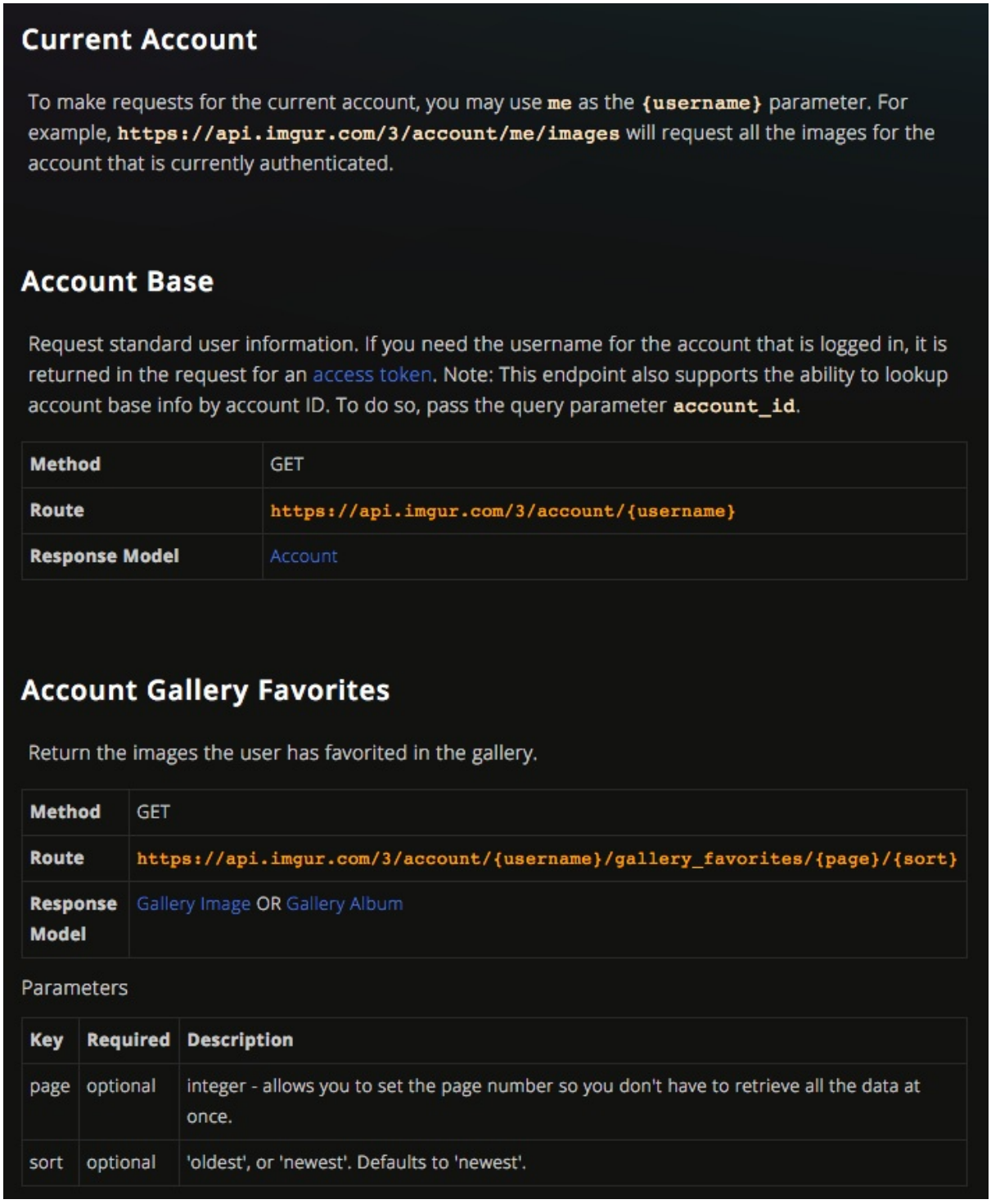
Check your database, you should see new images:



Creating an API endpoint

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

Example **Imgur** API's endpoints:



Current Account

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

Account Base

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

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

Account Gallery Favorites

Return the images the user has favorited in the gallery.

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

Parameters

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

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

Laravel 5 has a feature called **middleware groups**. When you put routes into the **web.php** file, Laravel apply the **web middleware group** to all the routes.

Let's open the **app/Http/Kernel.php** file:

```
protected $middlewareGroups = [
    'web' => [
```

```
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        // \Illuminate\Session\Middleware\AuthenticateSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],

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

As you see, the **web middleware group** can be found here. There is another **middleware group** called **api**. When building APIs with Laravel 5, it's better to utilize this middleware group, because when using APIs, we don't need Session or other stuff (CSRFToken, Cookie, etc.).

Now, it's time to add a new route. If we want to use the **api middleware group**, we can just put our routes into **api.php** file.

> **Note:** What is **route**? Basically, routing means that you will tell Laravel to get URL requests and assign them to specific actions that you want. For instance, when someone visits **angularbook.app**, which is the home page of our current application, Laravel will think: "Oh, this guy is going to the home page, I need to display something!"
>
> All route files are stored in the **routes** directory. We have three files: **web.php**, **console.php** and **api.php**. If you want to know more information, please read the Learning Laravel 5 book.

Open **api.php** and add:

```
Route::resource('images', 'ImagesController');
```

As you see, we use **Route:resource**. In Laravel, this is a **resourceful route**.

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

Simply put, instead of creating multiple routes manually:

```
Route::get('images', 'ImagesController@index');
Route::post('images', 'ImagesController@store');
...
```

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

When having a new route, we may need a new controller. Let's create one by running this **Artisan command**:

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

A new **ImagesController** will be created. By adding a **--resource** flag, Laravel generates a new **resource controller** for us, instead of a plain controller.

app/Http/Controller/ImagesController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ImagesController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
```

```
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        //
    }

    /**
     * Display the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        //
    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function destroy($id)
    {
        //
    }
```

```
}
```

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

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

```
php artisan route:list
```

```
vagrant@homestead:~/Code/angularbook$     php artisan route:list
+--------+-----------+-------------------------+---------------+-------------------------------------------------+--------------+
| Domain | Method    | URI                     | Name          | Action                                          | Middleware   |
+--------+-----------+-------------------------+---------------+-------------------------------------------------+--------------+
|        | GET|HEAD  | /                       |               | Closure                                         | web          |
|        | GET|HEAD  | api/images              | images.index  | App\Http\Controllers\ImagesController@index     | api          |
|        | POST      | api/images              | images.store  | App\Http\Controllers\ImagesController@store     | api          |
|        | GET|HEAD  | api/images/create       | images.create | App\Http\Controllers\ImagesController@create    | api          |
|        | GET|HEAD  | api/images/{image}      | images.show   | App\Http\Controllers\ImagesController@show      | api          |
|        | PUT|PATCH | api/images/{image}      | images.update | App\Http\Controllers\ImagesController@update    | api          |
|        | DELETE    | api/images/{image}      | images.destroy| App\Http\Controllers\ImagesController@destroy   | api          |
|        | GET|HEAD  | api/images/{image}/edit | images.edit   | App\Http\Controllers\ImagesController@edit      | api          |
|        | GET|HEAD  | api/user                |               | Closure                                         | api,auth:api |
+--------+-----------+-------------------------+---------------+-------------------------------------------------+--------------+
```

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

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

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

Alternatively, you may use the following:

```
public function index()
{
    $images = Image::all();
    return $images;
}
```

Don't forget to tell Laravel that we're using **Image** and **Response** here:

Find:

```
class ImagesController extends Controller
```

Add above:

```
use App\Image;
use Illuminate\Support\Facades\Response;
```

Go ahead and visit http://angularbook.app/api/images, you should see all images in **JSON** format:


```
[
 - {
      id: 1,
      title: "This is a new title! The message is cool!",
      description: "Sit tempora non et repudiandae laborum. Voluptatum aut enim odio deserunt. Facere voluptas voluptate necessitatibus
      magnam. Quo eos eum possimus sed quo sequi voluptas natus. Fuga ut cum aut dolorum deleniti est. Quos aut ab reiciendis rerum.
      Omnis harum voluptas veniam possimus modi qui. Natus sint provident earum ratione dicta ea cum. Consequatur provident et eaque
      molestias facere esse. Voluptatum et facilis praesentium itaque. Vero magni sapiente perspiciatis accusamus. Et earum consequatur
      maiores quasi quia ducimus et. In amet atque odit alias tempora facilis. Nulla reiciendis nihil voluptas. Iste rem hic
      praesentium. Ab asperiores tempora quae sint voluptates. Sunt temporibus perspiciatis accusantium sed et dolor earum. Aut vel
      illum perspiciatis et neque. Voluptatem quas quis odio minus delectus corrupti. Aut sed occaecati aut corrupti. Occaecati nobis
      voluptatem odio sequi ut aliquid quas. Qui molestiae temporibus odit ratione. Illum eveniet explicabo distinctio voluptatem. Quod
      nihil nobis laboriosam sint non aut enim. Non perspiciatis unde alias in. Earum quia sit nobis quia. Iste et soluta nam est et
      quae error inventore. Atque officia enim deleniti nisi. Dolorem sequi labore quidem suscipit tempore. Qui qui ex qui natus. de",
      thumbnail: "https://angularbooks.com/img/angular4/img1.jpg",
      imageLink: "https://angularbooks.com/img/angular4/img1-1.jpg",
      user_id: 3,
      created_at: "2017-02-18 09:58:48",
      updated_at: "2017-03-25 02:40:43"
   },
 - {
      id: 2,
      title: "Dolores placeat at est et excepturi quia velit molestiae.",
      description: "Odio doloremque blanditiis incidunt eos facilis. Et inventore qui dignissimos omnis animi ipsam alias. Est
      voluptatem minus vel ratione culpa ipsa autem. Doloremque temporibus odit dolore molestiae. Omnis aliquam ex quia velit ipsam et.
      In facilis consequatur nostrum quod eligendi temporibus. Quibusdam dolor quia dolores dignissimos ullam modi magnam est. Non
      dolorem molestiae voluptatem praesentium qui velit architecto. Debitis consectetur sapiente consequuntur eius velit sint. Dolores
      dolores labore delectus expedita asperiores nisi. Et non quaerat velit beatae perferendis adipisci. Rerum esse sit magnam ab sunt
      enim. Amet quia sint necessitatibus possimus excepturi a autem. Aut ullam cum eligendi veritatis unde ea earum. Debitis et esse
      maiores officia sequi et vel. Et debitis beatae ullam a. Expedita minima quasi vero porro mollitia nobis. At voluptas ut non
      cumque hic doloribus. Minus nihil cumque suscipit optio dolores.",
      thumbnail: "https://angularbooks.com/img/angular4/img2.jpg",
      imageLink: "https://angularbooks.com/img/angular4/img2-1.jpg",
      user_id: 1,
      created_at: "2017-02-18 09:58:48",
      updated_at: "2017-02-18 09:58:48"
   },
 - {
      id: 3,
      title: "Quisquam a inventore dicta officiis. 3",
      description: "Vel minima dolore omnis fugit ut. Earum officiis enim eligendi alias. Officiis dolorem modi eaque ex. In laboriosam
      nisi veniam natus dolorem inventore. Aut aperiam consequatur nesciunt at accusantium ea. Aliquadm reprehenderit ad quo blanditiis
      eligendi. Aut vel ut quia blanditiis pariatur. Perferendis assumenda et minus et dolorem. Dignissimos ab cum suscipit numquam hic
      harum veritatis. Et mollitia molestiae eum quia adipisci autem. Commodi aliquam recusandae voluptates dolorem sint et libero.
      Rerum consequatur atque occaecati velit dolorem. Amet praesentium animi atque quod. Aut aut fuga quia nobis aspernatur eos
      doloremque assumenda. Molestias quae rem animi. Esse atque inventore ratione et iusto aliquam. Est eum nam aut molestias
      consectetur voluptate. Voluptas sunt et maiores et. Id id minima ea. Maxime doloremque libero veritatis et ea. Occaecati optio est
      inventore laboriosam. Magni ipsa omnis neque minima ipsam. Accusamus perferendis occaecati autem consequatur hic harum voluptas.
      Dolore voluptas ab et sunt dolor assumenda quia odit. Quasi eaque enim voluptates impedit unde corrupti quasi minima. Unde tempore
      delectus error voluptatem inventore aperiam nihil doloremque.",
      thumbnail: "https://angularbooks.com/img/angular4/img3.jpg",
      imageLink: "https://angularbooks.com/img/angular4/img3-1.jpg",
      user_id: 4,
```


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

However, we need to do one more thing.

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

Open **api.php**, add **our images resourceful route** like this:

```
Route::group(['prefix' => 'v1'], function(){
    Route::resource('images', 'ImagesController');
});
```

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

Just a quick tip, If you don't want to use the **api.php** file, you can put routes into the **api middleware group** as follows:

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

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

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

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

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

```
429: Too Many Attempts
```

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

Feel free to change the limit to whatever you want.

Using Postman to test our API

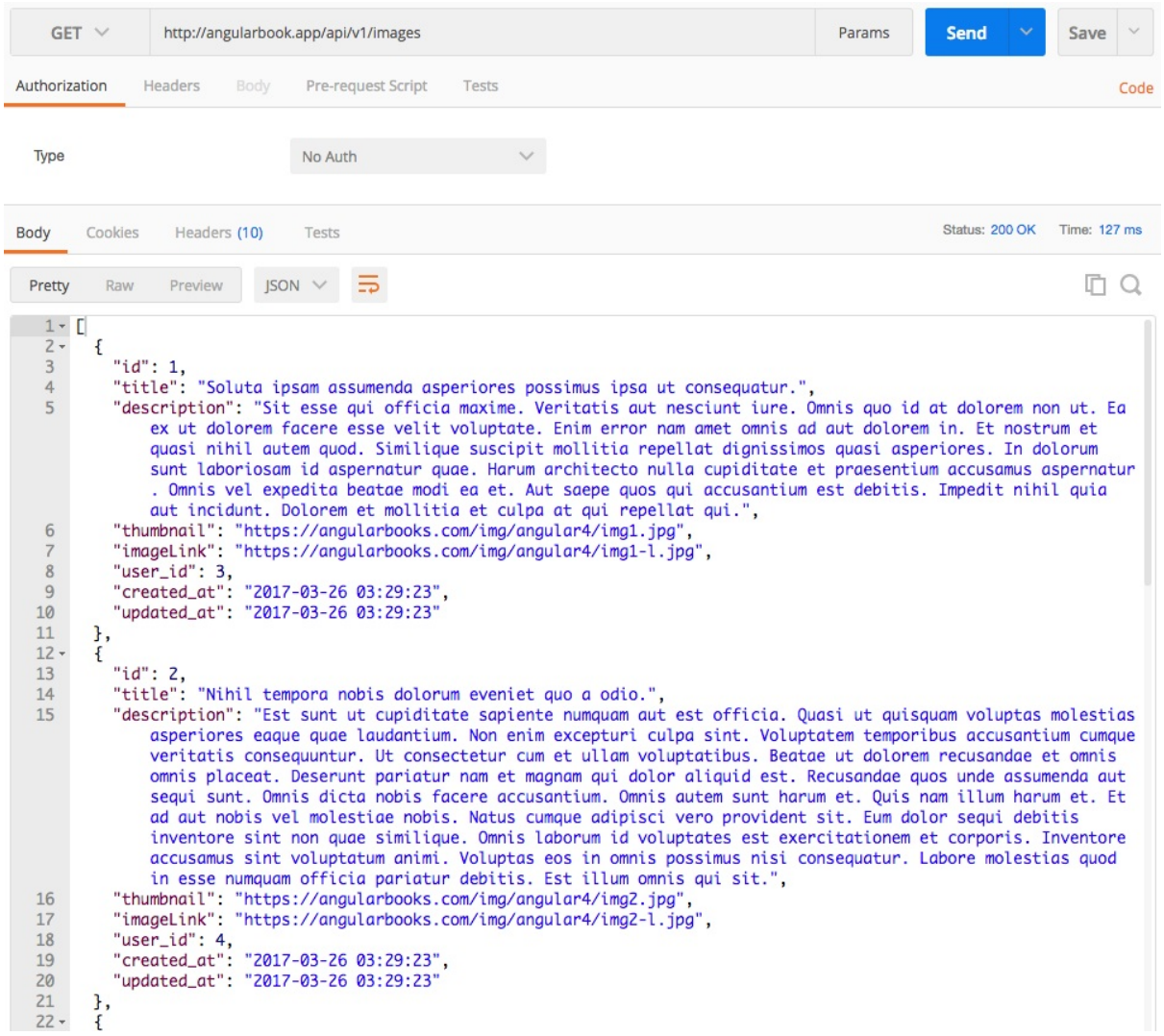
When working with APIs, we should use **Postman** - a Google Chrome extension.

> **Note:** You have to install Google Chrome first, and then use **Google Chrome** to install the extension.

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

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

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



Pretty simple so far, right?

The output is very useful. We can see the **status code** (`200`), the **execution time** (`174 ms`) and our **images** in pretty JSON format.

Adding CORS

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

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

To enable CORS, we have many ways to do. In this section, I'll show you two popular methods.

First Method: Adding CORS using a custom middleware

Fábio Vedovelli shows us a nice way to build a custom middleware and add CORS header to our response. This is a simple method.

Let's create a custom **Cors** middleware:

```
php artisan make:middleware Cors
```

Next, update the handle method as follows:

app/Http/Middleware/Cors.php

```
<?php

namespace App\Http\Middleware;

use Closure;

class Cors
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        return $next($request)
            ->header('Access-Control-Allow-Origin', '*')
            ->header('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, OPTIONS')
;
```

```
    }
}
```

Finally, open **app/Http/Kernel.php**, and add **cors** to the **$routeMiddleware**:

```
protected $routeMiddleware = [
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'cors' => \App\Http\Middleware\Cors::class,
];
```

Now if we want to enable **CORS** for our routes, we can do as follows:

```
Route::group(['prefix' => 'v1', 'middleware' => 'cors'], function(){
    Route::resource('images', 'ImagesController');
});
```

Second Method: Adding CORS using Laravel CORS package

The simplest one is to use a popular package called laravel-cors.

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

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

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

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

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

```
Route::group(['prefix' => 'v1', 'middleware' => 'cors'], function(){
    Route::resource('images', 'ImagesController');
});
```

That's it!

Our API is now working properly!

Using Angular HTTP Service to load data from our backend

Back to our Angular app! In this section, we will learn how to use the **HTTP Service** to fetch data from our Laravel backend.

> **Note:** Please note that we'll be **using the Angular app (nggallery)**.

Register HTTP services

Angular is flexible. We may use any tools or modules that we like to fetch data from an external source. However, there is a good separate add-on module called **@angular/http** that we can use to call out to APIs.

To use the module, we have to import **HttpModule**, which is a complete collection of the HTTP services.

> **Note:** You may skip this step if the HttpModule have been imported already.

Open **app.module.ts**, and import the **HttpModule**:

```
import { HttpModule } from "@angular/http";
```

Finally, we add the **HttpModule** to the **imports** list, so that we can access the services from anywhere in our application:

```
imports: [
routes,
BrowserModule,
FormsModule,
HttpModule
],
```

Done! Now we can use all HTTP services.

Make a simple GET request to our API

Let's try to send a **GET** request to our API to get all the images.

Open **image.service.ts**, and define our constructor:

```
constructor(private http:Http) {
}
```

Here we inject the **Http** module, so that we can use it in our **image service**.

Remember to import the **Http** module into our **image service** by adding this line at the top of the file:

```
import {Http} from '@angular/http';
```

We don't need the array of images anymore, so we can safely **remove** it:

```
images: Image[] = [
new Image('First image', 'First image description', 'https://angularbooks.com/img/angu
lar4/img1.jpg', 'https://angularbooks.com/img/angular4/img1-l.jpg'),
new Image('Second image', 'Second image description', 'https://angularbooks.com/img/an
gular4/img2.jpg', 'https://angularbooks.com/img/angular4/img2-l.jpg'),
new Image('Third image', 'Third image description', 'https://angularbooks.com/img/angu
lar4/img3.jpg', 'https://angularbooks.com/img/angular4/img3-l.jpg'),
new Image('Fourth image', 'Fourth image description', 'https://angularbooks.com/img/an
gular4/img4.jpg', 'https://angularbooks.com/img/angular4/img4-l.jpg'),
new Image('Fifth image', 'Fifth image description', 'https://angularbooks.com/img/angu
lar4/img5.jpg', 'https://angularbooks.com/img/angular4/img5-l.jpg'),
new Image('Sixth image', 'Sixth image description', 'https://angularbooks.com/img/angu
lar4/img6.jpg', 'https://angularbooks.com/img/angular4/img6-l.jpg'),
new Image('Seventh image', 'Seventh image description', 'https://angularbooks.com/img/
angular4/img7.jpg', 'https://angularbooks.com/img/angular4/img7-l.jpg'),
new Image('Eighth image', 'Eighth image description', 'https://angularbooks.com/img/an
gular4/img8.jpg', 'https://angularbooks.com/img/angular4/img8-l.jpg'),
];
```

Now we can modify the **getImages()** function and make our first HTTP request as follows:

```
getImages(): Observable<Image[]> {
return this.http.get('http://angularbook.app/api/v1/images')
  .map((response: Response) => response.json());
}
```

Well, I know the code is hard to read if you don't use **ES6** or **RxJS** frequently. Don't worry, I'll explain everything to you soon. But first, let's try to make it work by importing these:

```
import 'rxjs/RX';
import {Observable} from 'rxjs;
```

As you may already see, we used **Response** here, so let's import it as well:

```
import {Http, Response} from '@angular/http';
```

Because the **getImages()** function will return an **Observable**, we have to import **Observable**, too:

```
import 'rxjs/RX';
import {Observable} from 'rxjs/Observable';
```

Next, open **image-list.component.ts**, and find:

```
images: Image[] = [];
```

Modify to:

```
images: Observable<Image[]>;
```

and import **Observable** to this component as well:

```
import {Observable} from 'rxjs/Observable';
```

We've just changed the type of our **images**, from an array to an **Observable**.

One last step, open **image-list.component.html**, and find:

```
<a2g-image *ngFor="let image of images" [image]="image" (click)="onSelect(image)"></a2
g-image>
```

Modify to:

```
<ng-image *ngFor="let image of images | async" [image]="image" (click)="onSelect(image)
"></ng-image>
```

Here we just added the **Async** pipe and use it with **ngFor** to iterate over our list of images.

Visit our app and you should see:

Congratulations! You've successfully loaded data from our Laravel backend!

But what is **Observable** and **RxJS**? Why do we have to import some "mysterious things" into our app?

Introducing Observable

This is an era of **asynchronous programming** (aka async programming)! When sending requests, there are some new techniques that we can use to make our requests **asynchronous**. Which means we don't have to wait for our requests to finish before moving to another task.

In the past, when talking about asynchronous code, people usually thought about **callbacks**. And then, many people preferred **Promise**, which was also a cool approach when dealing with async code. However, Promise only allows us to handle one event when an asynchronous operation completes or fails. Recently, people love **Observable**, which can be thought as an **improved version of Promise**. Observable allows us to handles multiple events over time. More than that, Observable is **cancellable**. For example, we can send a request to look for something on a server; when we don't need to do that operation anymore, we can just simply cancel it without waiting for the request to return a result.

To use **Observable**, we will have to make use of an external library, because ES6 doesn't support it by default. One of the most popular libraries is RxJS (The Reactive Extensions for JavaScript). RxJS provides some Observable operators that we can use to transform, filtering, composing and querying streams of data. Here are some useful **Observable** operators:

- map
- filter
- find
- take
- skip
- merge

For a full list of operators, you may check out the official RxJS docs or RxMarbales.

How our HTTP request works

Now that you know about **Observable** and **RxJS**!

To use **Observable** in Angular, we have to import it and the **RxJS** library first:

```
import 'rxjs/RX';
import {Observable} from "rxjs";
```

When we use the **http.get** to perform a request, it will return an **Observable**:

```
getImages(): Observable<Image[]> {
    return this.http.get('http://angularbook.app/api/v1/images')
  }
```

Our request also emits a **Response** object, which usually contains our data. We then use the **map** operator to transform the response to **JSON**:

```
getImages(): Observable<Image[]> {
return this.http.get('http://angularbook.app/api/v1/images')
  .map((response: Response) => response.json());
}
```

To render an **Observable** with **ngFor**, we have to use a special **Async pipe** in our template:

```
<ng-image *ngFor="let image of images | async" [image]="image" (click)="onSelect(image)
"></ng-image>
```

> **Note:** we talk more about **Pipes** and how to use them in the next section.

That's how we send a request and process the data in Angular using **Observable**! Simple, isn't it?

Just a few more things to note. Take a look at this:

```
(response: Response) => response.json()
```

This is an **arrow function** (aka **fat arrow function**), which is a new feature in ES6. The arrow functions have a shorter syntax than a regular function expression. If you're not familiar with ES6, you may learn more about the arrow function here. Basically, the code above is equivalent to:

```
function(response: Response) {
    return response.json();
}
```

> **Tip:** to learn more about ES6, you can read this free book: Exploring ES6.

Optionally, if we know that our data is **of type Image**, we can code as follows:

```
getImages(): Observable<Image[]> {
return this.http.get('http://angularbook.app/api/v1/images')
  .map((response: Response) => response.json() as Image[]);
}
```

Introducing Pipes

Ok, I know that some of you might think about pipes in a video game. Unfortunately, we're not talking about Mario pipes here, but **Angular pipes**!

Angular pipes are a great feature of Angular. We use them to format output in our views (templates). For example, we may convert a date and time, from **2017-02-14 14:02:00** to **February 14, 2007 02:02:00 PM**, using pipes.

Bear in mind that we can use pipes to transform many things, such as currency, date, JSON, async, etc.

How to use pipes

The great thing is, Angular provides many built-in pipes that we can use out of the box. Here are some of them:

- uppercase
- lowercase
- json
- async
- date
- percent

For a full list of pipes, check this out:

Angular built-in pipes

Let's try to use one of the built-in pipes called **uppercase**!

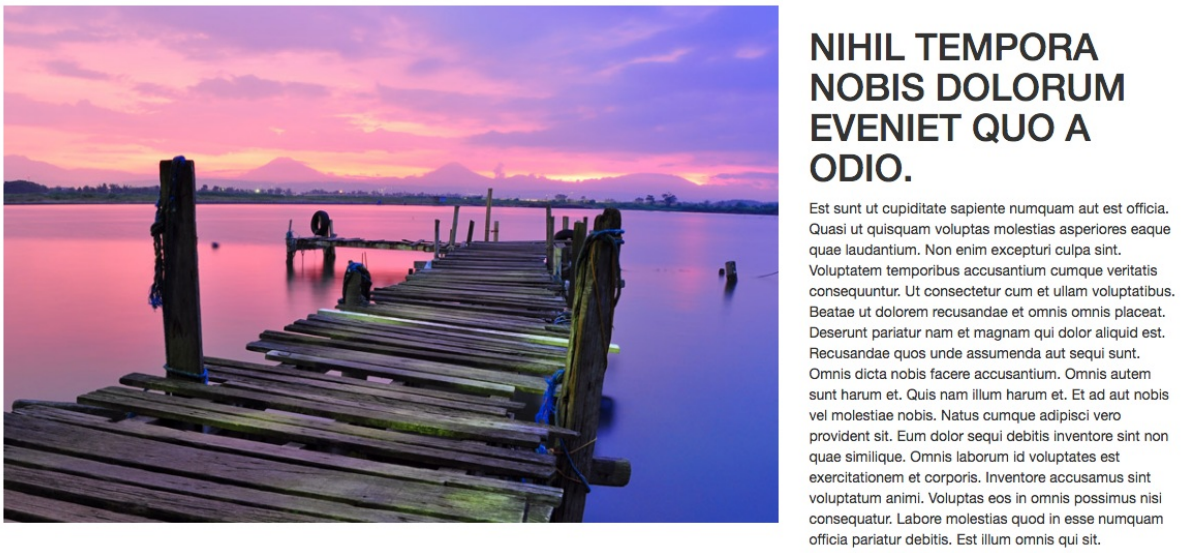
Open **image-detail.component.html**, and find:

```
<h1>{{selectedImage.title} </h1>
```

Change to:

```
<h1>{{selectedImage.title | uppercase}} </h1>
```

As you may have guessed, by adding **| uppercase**, we will transform the image title into uppercase!



We can also chain pipes together. For example:

```
{{ birthday | date | uppercase }}
```

You may even create a custom pipe if you want!

To learn more about pipes, be sure to check out the official documentation:

Pipe documentation

Chapter 3 Summary

In this chapter, you've gone through the different steps involved in building REST APIs using Laravel. Let's review what you've learned so far:

- You've known how to create a new Laravel app.
- You've understood what REST API is.
- You've known how to enable CORS in Laravel 5.
- You can now build APIs using Laravel!
- You've learned about Observable and RxJS.
- Sending a request using Angular's **HttpModule** is not that hard, right?
- Pipes are wonderful! Be sure to learn more pipes and use them in your project!

In the next chapter, we will build an admin control panel using Angular! You may use this application to manage posts, images or anything that you like!

The real journey begins now!

Chapter 4 - Building An Admin Control Panel Using Angular and Laravel

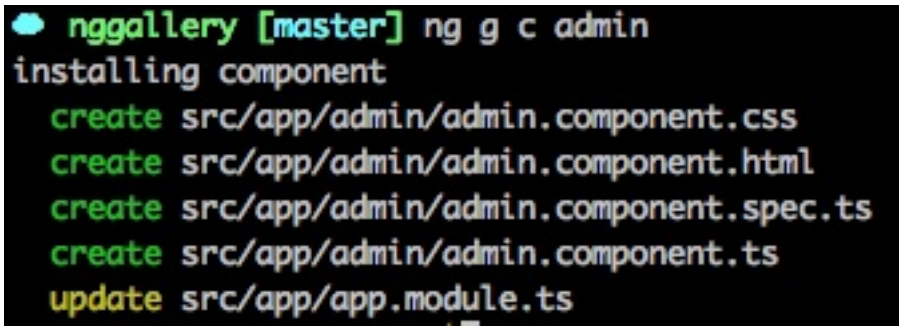
In this chapter, we will build an admin control panel using Angular. Along the way, we'll learn more about HttpModule, forms, JWT authentication and other important features of Angular.

Building an Admin area

To manage our application, we will create a place that only administrators can access. This place is called **admin area**.

First, let's create a new **Admin Component**:

```
ng g c admin
```



Next, open **app.routes.ts**, and find:

```
const appRoutes: Routes = [
  { path: '', redirectTo:'/gallery', pathMatch: 'full'},
  { path: 'gallery', component: GalleryComponent},
  { path: 'contact', component: ContactComponent},
  { path: 'about', component: AboutComponent}
];
```

Update to:

```
const appRoutes: Routes = [
  { path: '', redirectTo:'/gallery', pathMatch: 'full'},
```

```
  { path: 'gallery', component: GalleryComponent},
  { path: 'contact', component: ContactComponent},
  { path: 'about', component: AboutComponent},
  { path: 'admin', component: AdminComponent}
];
```

Don't forget to import the **Admin Component**:

```
import {AdminComponent} from './admin/admin.component';
```

Visit http://localhost:4200/admin to make sure that our **Admin** route is working.

admin works!

Cool! Let's add some links so that we can easily access and manage our data.

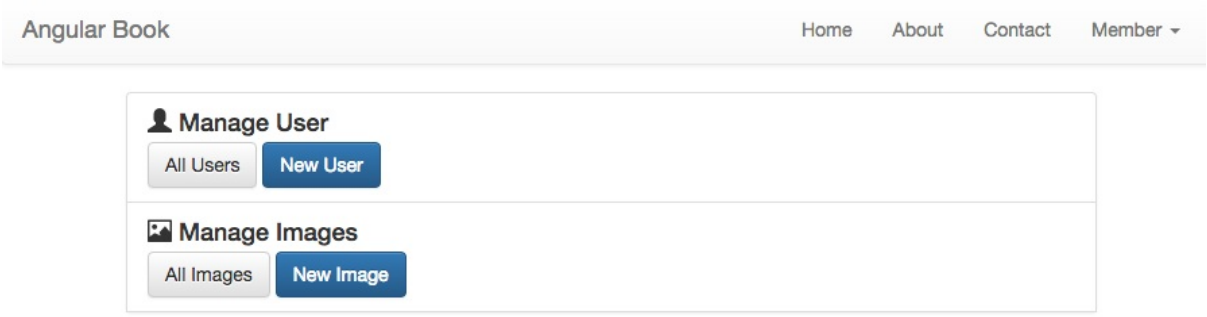
Open **admin.component.html**, and update it as follows:

admin/admin.component.html

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

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

      <div class="list-group">
        <div class="list-group-item">
          <div class="row-action-primary">
            <h4 class="list-group-item-heading"><span class="glyphicon glyphicon-user">
</span> Manage User</h4>
          </div>
          <div class="row-content">
            <a [routerLink]="['/admin/users']" class="btn btn-default btn-raised">All
Users</a>
            <a [routerLink]="['/admin/users/create']" class="btn btn-primary btn-raised
">New User</a>
          </div>
        </div>
        <div class="list-group-separator"></div>
        <div class="list-group-item">
          <div class="row-action-primary">
            <h4 class="list-group-item-heading"><span class="glyphicon glyphicon-pictu
re"></span> Manage Images</h4>
          </div>
          <div class="row-content">
```

```
            <a [routerLink]="['/admin/images']" class="btn btn-default btn-raised">All
 Images</a>
            <a [routerLink]="['/admin/images/create']" class="btn btn-primary btn-rais
ed">New Image</a>
          </div>
        </div>
        <div class="list-group-separator"></div>
      </div>

    </div>

  </div>
</div>
```

Here is our new admin control panel:



Creating Admin Routes as child routes

If you're working with Laravel, you may know that we can create a **route group**, which allows us to share route attributes (namespace, middleware, etc.) across a large number of routes. In Angular, we can do the same thing!

First, let's create a new file called **admin.routes.ts** to **store all our admin routes**:

admin/admin.routes.ts

```
import {Routes} from "@angular/router";
import {AdminImageListComponent} from "./admin-image-list/admin-image-list.component";
import {DashboardComponent} from "./dashboard.component";

export const adminRoutes: Routes = [
  { path: '', component: DashboardComponent},
  { path: 'images', component: AdminImageListComponent}
];
```

Similar to what we have done in Chapter 2, we created a new array of routes called **adminRoutes**. Each route also has a path (e.g. images) and a component (e.g. AdminImageListComponent), which is used to handle the route.

As you may guess, we don't have the **Dashboard Component** and **Admin Image List Component** yet, so let's create them!

Be sure that we're in the **admin directory** first.

```
cd src/app/admin
```

Then run these commands to generate new components:

```
ng g c admin-image-list

ng g c dashboard --flat
```

> **Note:** You may put the **Dashboard Component** inside a directory if you want.

```
nggallery [master] ⚡cd src/app/admin
admin [master] ⚡ng g c admin-image-list
installing component
  create src/app/admin/admin-image-list/admin-image-list.component.css
  create src/app/admin/admin-image-list/admin-image-list.component.html
  create src/app/admin/admin-image-list/admin-image-list.component.spec.ts
  create src/app/admin/admin-image-list/admin-image-list.component.ts
  update src/app/app.module.ts
admin [master] ⚡ng g c dashboard --flat

installing component
  create src/app/admin/dashboard.component.css
  create src/app/admin/dashboard.component.html
  create src/app/admin/dashboard.component.spec.ts
  create src/app/admin/dashboard.component.ts
  update src/app/app.module.ts
```

The **Dashboard Component** will be the **new admin home page** (replacing the **Admin Component**), so you can **copy** the code from the **Admin Component (admin.component.html)** and **paste** it into **dashboard.component.html**.

admin/dashboard.component.html

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

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

      <div class="list-group">
        <div class="list-group-item">
          <div class="row-action-primary">
            <h4 class="list-group-item-heading"><span class="glyphicon glyphicon-user">
</span> Manage User</h4>
          </div>
          <div class="row-content">
            <a [routerLink]="['/admin/users']" class="btn btn-default btn-raised">All
Users</a>
            <a [routerLink]="['/admin/users/create']" class="btn btn-primary btn-raised
">New User</a>
          </div>
        </div>
        <div class="list-group-separator"></div>
        <div class="list-group-item">
          <div class="row-action-primary">
            <h4 class="list-group-item-heading"><span class="glyphicon glyphicon-pictu
re"></span> Manage Images</h4>
          </div>
          <div class="row-content">
            <a [routerLink]="['/admin/images']" class="btn btn-default btn-raised">All
 Images</a>
            <a [routerLink]="['/admin/images/create']" class="btn btn-primary btn-rais
ed">New Image</a>
          </div>
        </div>
        <div class="list-group-separator"></div>
      </div>

    </div>

  </div>
</div>
```

Next, open **app.routes.ts**, and find:

```
{ path: 'admin', component: AdminComponent}
```

Modify to:

```
{ path: 'admin', component: AdminComponent, children: adminRoutes}
```

We add a new **children** property, which indicates that the **admin** path has some **child routes** (**adminRoutes**).

Don't forget to Import **adminRoutes** into the file:

```
import {adminRoutes} from './admin/admin.routes';
```

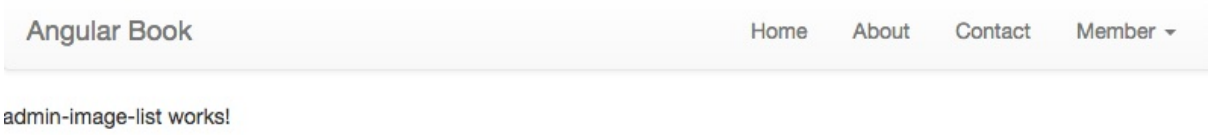
Finally, open **admin.component.html**, **remove all** the code, and add:

admin/admin.component.html

```
<router-outlet></router-outlet>
```

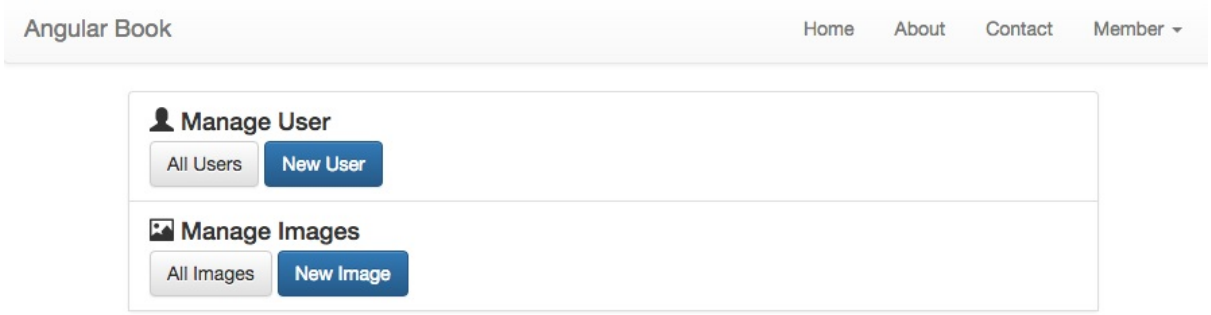
This **router-outlet** tag will tell Angular that we want to load our **child routes' content** here.

Visit http://localhost:4200/admin/images, you should see:



admin-image-list works!

The dashboard (http://localhost:4200/admin) still works as expected:



Well done! Our admin routes are working!

> **Tip:** Please note that you don't have to create child routes if you don't like. Angular is flexible. Feel free to arrange and design your app structure in your own way.

List all images

In this section, we're going to create a place to display all images. When visiting this page, we can see all images, and we can edit or delete an image.

Like what we've done before, let's create an **Observable** called **images** first.

Open **admin-image-list.component.ts**, and find:

```
export class AdminImageListComponent implements OnInit {
```

Add below:

```
images: Observable<Image[]>;
```

We can then use the **Image Service** to **load images** from our backend:

```
constructor(private imageService: ImageService) { }

ngOnInit() {
  this.images = this.imageService.getImages();
}
```

Don't forget to import these:

```
    import {Observable} from 'rxjs/Observable';
    import { Image } from '../../models/image';
    import {ImageService} from '../../services/image.service';
```

Our **Admin Image List Component** should now look like this:

admin-image-list-component.ts

```
import { Component, OnInit } from '@angular/core';
import {Observable} from 'rxjs/Observable';
import { Image } from '../../models/image';
import {ImageService} from '../../services/image.service';

@Component({
  selector: 'ng-admin-image-list',
  templateUrl: './admin-image-list.component.html',
  styleUrls: ['./admin-image-list.component.css']
})
export class AdminImageListComponent implements OnInit {
  images: Observable<Image[]>;
```

```
  constructor(private imageService: ImageService) { }

  ngOnInit() {
    this.images = this.imageService.getImages();
  }

}
```

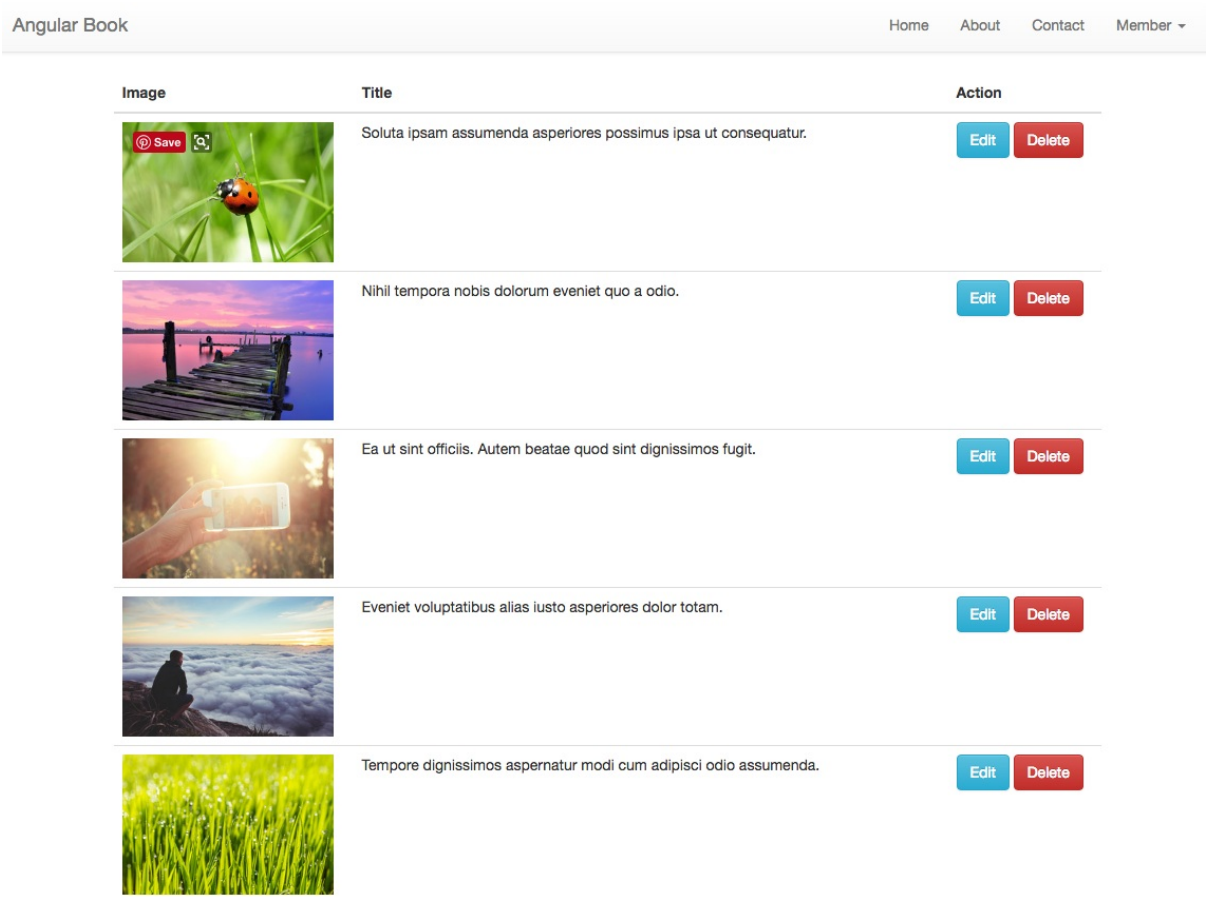
Next step, let's display images by updating our **admin-image-list** template:

admin-image-list.component.html

```
<div class="col-md-10 col-md-offset-1">
  <div class="table-responsive">
    <table class="table">
      <thead>
      <tr>
        <th>Image</th>
        <th>Title</th>
        <th>Action</th>
      </tr>
      </thead>
      <tbody>

      <tr *ngFor="let image of images | async">
        <td><img src="{{image.thumbnail}}" width="200"/></td>
        <td>{{image.title}} </td>
        <td>
          <a [routerLink]="['/images/edit']" class="btn btn-info"> Edit</a>
          <a [routerLink]="['/images/delete']" class="btn btn-danger"> Delete</a>
        </td>
      </tr>

      </tbody>
    </table>
  </div>
</div>
```

As you see, we don't have to create a new image tag (such as **ng-image**), we can use **ngFor** to create multiple **tr tag** here. When working with tables, you can use this technique to generate columns easily. A nice trick, right?

Visit http://localhost:4200/admin/images to see all images!

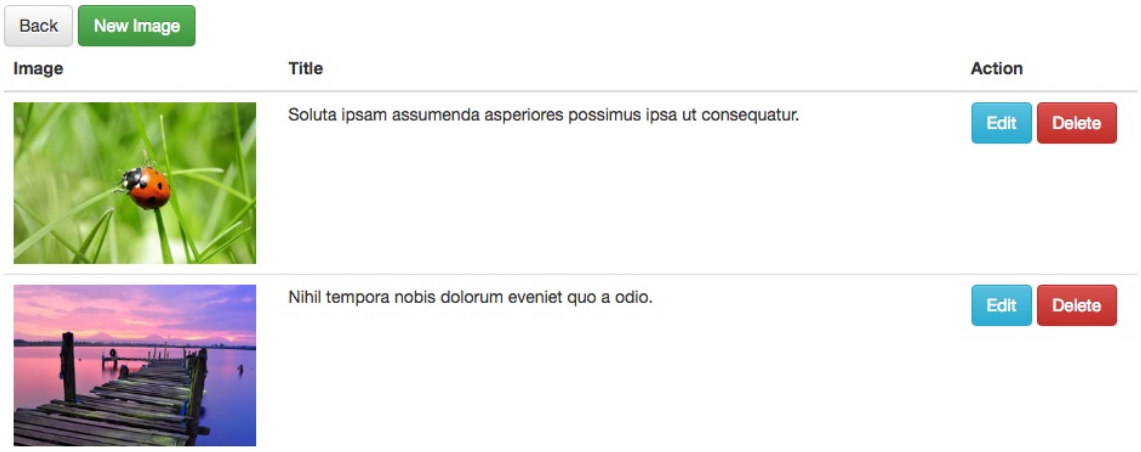


Cool! If we want to have a better UX layout, we can add some buttons to create images quickly and go back to our admin panel.

Open **admin-image-list.component.html**, and add this code at the top:

```
<div class="col-md-10 col-md-offset-1">
  <div>
      <a [routerLink]="['/admin/']" class="btn btn-default"> Back</a>
      <a [routerLink]="['/admin/images/create']" class="btn btn-success"> New Image</a>

  </div>
</div>
```

Here are our new buttons:



Forms in Angular: template-driven approach

Forms are usually the most crucial part of an application. We use forms to do many things: validate users' input, display errors, transform data, etc. Luckily, Angular provides everything that we need to build a good form.

There are two ways that we can use to write a form in Angular:

- **Template-driven**: as the name suggests, we will build our forms mostly in our templates. This method is suitable for simple forms that require simple validation.
- **Code-driven**: this is a new but powerful method that we can use to build reactive forms. We'll create forms directly in our component.

You may read the Official Documentation to learn more about forms.

We will use both methods in this book!

Creating our first form in Angular

In this section, we'll be creating a form using the **template-driven** way. The form will be used to add a new image.

To build a form using the **template-driven** way, we will have to import **FormsModule**.

> **Note:** You can skip this step if **FormsModule** has been imported already.

Open **app.module.ts**, and make sure that we've imported the **FormsModule**:

```
import { FormsModule } from '@angular/forms';
```

And add it into the **imports** array:

```
imports: [
  routes,
  BrowserModule,
  FormsModule,
  HttpModule
],
```

FormsModule contains some directives that we can add to a normal form and turn it into a powerful Angular form.

It's time to create a new **Admin Image Create Component**. When you're in the **admin** directory, run this command:

```
ng g c admin-image-create
```

```
admin [master] ⚡ng g c admin-image-create
installing component
  create src/app/admin/admin-image-create/admin-image-create.component.css
  create src/app/admin/admin-image-create/admin-image-create.component.html
  create src/app/admin/admin-image-create/admin-image-create.component.spec.ts
  create src/app/admin/admin-image-create/admin-image-create.component.ts
  update src/app/app.module.ts
```

The **create image form** will be placed at http://localhost:4200/admin/images/new, so open **admin.routes.ts**, and add a new route:

```
export const adminRoutes: Routes = [
  { path: '', component: DashboardComponent},
  { path: 'images', component: AdminImageListComponent},
  { path: 'images/create', component: AdminImageCreateComponent}
];
```

Don't forget to import the component:

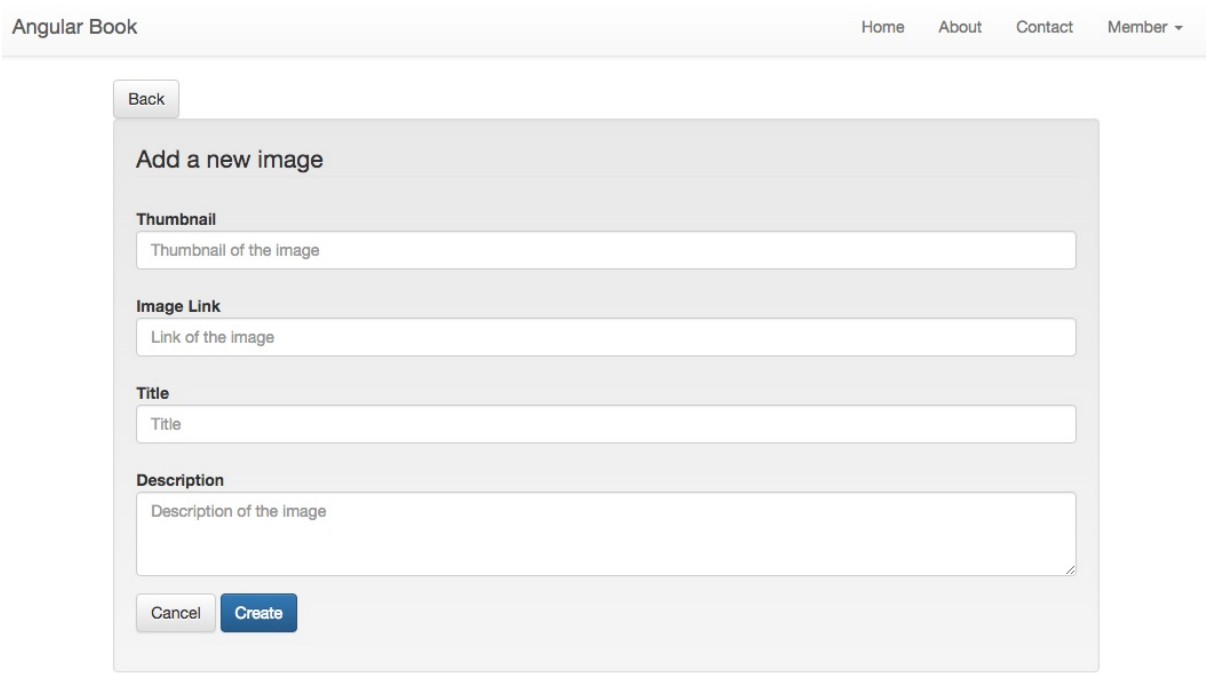
```
import {AdminImageCreateComponent} from './admin-image-create/admin-image-create.compo
nent';
```

Now we can build our form using **Bootstrap** and normal HTML code:

admin-image-create.component.html

```
<div class="col-md-10 col-md-offset-1">
  <div>
    <a [routerLink]="['/admin/']" class="btn btn-default"> Back</a>
  </div>
</div>

<div class="col-md-10 col-md-offset-1">
  <div class="well well bs-component">
    <form class="form-horizontal">
      <fieldset>
        <legend>Add a new image</legend>
        <div class="form-group">
          <label for="thumbnail" class="col-lg-2 control-label">Thumbnail</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="thumbnail" name="thumbnail" pl
aceholder="Thumbnail of the image">
          </div>
        </div>
        <div class="form-group">
            <label for="imagelink" class="col-lg-2 control-label">Image Link</label>
            <div class="col-lg-10">
              <input type="text" class="form-control" id="imageLink" name="imageLink"
placeholder="Link of the image">
            </div>
        </div>
        <div class="form-group">
          <label for="title" class="col-lg-2 control-label">Title</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="title" name="title" placeholder
="Title">
          </div>
        </div>
        <div class="form-group">
          <label for="description" class="col-lg-2 control-label">Description</label>
          <div class="col-lg-10">
            <textarea class="form-control" rows="3" id="description" name="description"
 placeholder="Description of the image"></textarea>
          </div>
        </div>

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

Go to http://localhost:4200/admin/images/create, we should see a nice form there:



So, how do we turn this form into an Angular form?

It's very simple. All you have to do is find:

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

Update to:

```
<form class="form-horizontal" (ngSubmit)="createImage(createImageForm.value)" #createI
mageForm="ngForm">
```

We assign our form data to a custom variable called **createImageForm** using **ngForm**. By doing this, we tell Angular: "This is the form that we want you to handle!".

> **Note**: **createImageForm** is a custom name, you can pick a different name if you want.

We also use **output binding** to response to an event again:

```
(ngSubmit)="createImage(createImageForm.value)"
```

Using **ngSubmit**, when we hit the submit button, it calls the **createImage()** method. The **createImageForm.value** contains our form data.

Next, we have to add **ngModel** directives to all input fields:

Find:

```
<input type="text" class="form-control" id="thumbnail" name="thumbnail" placeholder="T
humbnail of the image">

<input type="text" class="form-control" id="imageLink" name="imageLink" placeholder="L
ink of the image">

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

<textarea class="form-control" rows="3" id="description" name="description" placeholder
="Description of the image"></textarea>
```

Modify to:

```
<input type="text" class="form-control" id="thumbnail" name="thumbnail" placeholder="T
humbnail of the image" ngModel>

<input type="text" class="form-control" id="imageLink" name="imageLink" placeholder="L
ink of the image" ngModel>

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

<textarea class="form-control" rows="3" id="description" name="description" placeholder
="Description of the image" ngModel></textarea>
```

Our form should now look like:

admin-image-create.component.html

```
<div class="col-md-10 col-md-offset-1">
  <div>
    <a [routerLink]="['/admin/']" class="btn btn-default"> Back</a>
  </div>
</div>

<div class="col-md-10 col-md-offset-1">
  <div class="well well bs-component">
    <form class="form-horizontal" (ngSubmit)="createImage(createImageForm.value)" #cre
```

```
ateImageForm="ngForm">
      <fieldset>
        <legend>Add a new image</legend>
        <div class="form-group">
          <label for="thumbnail" class="col-lg-2 control-label">Thumbnail</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="thumbnail" name="thumbnail" pl
aceholder="Thumbnail of the image" ngModel>
          </div>
        </div>
        <div class="form-group">
          <label for="imagelink" class="col-lg-2 control-label">Image Link</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="imageLink" name="imageLink" pl
aceholder="Link of the image" ngModel>
          </div>
        </div>
        <div class="form-group">
          <label for="title" class="col-lg-2 control-label">Title</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="title" name="title" placeholder
="Title" ngModel>
          </div>
        </div>
        <div class="form-group">
          <label for="description" class="col-lg-2 control-label">Description</label>
          <div class="col-lg-10">
            <textarea class="form-control" rows="3" id="description" name="description"
 placeholder="Description of the image" ngModel></textarea>
          </div>
        </div>

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

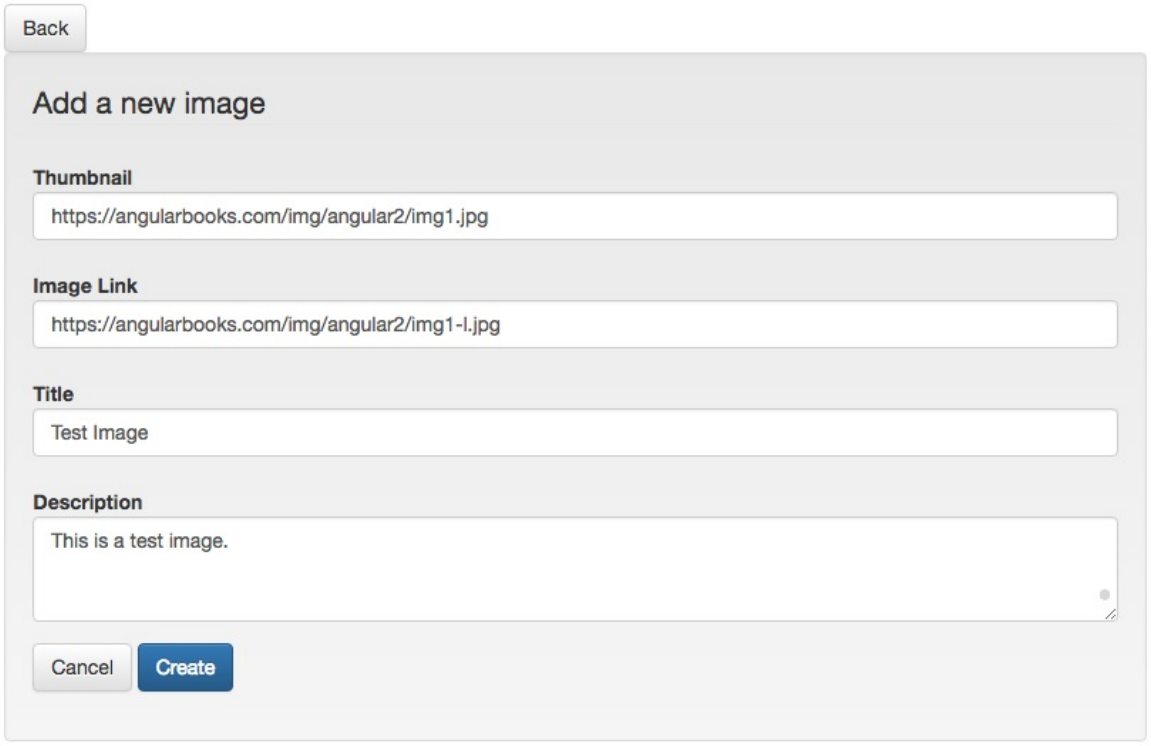
Finally, open **admin-image.create.component.ts**, and create a new **createImage** method:

admin-image.create.component.ts

```
import { Component, OnInit } from '@angular/core';
```

```
@Component({
  selector: 'ng-admin-image-create',
  templateUrl: './admin-image-create.component.html',
  styleUrls: ['./admin-image-create.component.css']
})
export class AdminImageCreateComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

  createImage(image) {
    console.log(image);
  }
}
```

Now if you're trying to submit the form, we should see the form values on the **console** because we're using **console.log** here.



When using **Google Chrome**, right click, and choose **Inspect**. After that, go to the **Console** tab. You should see an object, which contains your form data!

Tip: Using the **Console** tab when building an Angular app is a nice way to test your app.

Sending POST request to the backend using Image Service

Basically, we can submit the form directly using the **createImage** method. But that's not a good way if we want to add images in other components. Using **Image Service** is a better choice.

We have the **Image Service** already, so let's open it, and add a new **addImage** method:

image.service.ts

```
import { Injectable } from '@angular/core';
import {Image} from '../models/image';
import {Http, Response} from '@angular/http';
import 'rxjs/RX';
import {Observable} from 'rxjs/Observable';

@Injectable()
export class ImageService {

  constructor(private http:Http) {
  }

  getImages(): Observable<Image[]> {
    return this.http.get('http://angularbook.app/api/v1/images')
      .map((response: Response) => response.json());
  }
  addImage(image: Object): Observable<Image[]> {
    return this.http.post('http://angularbook.app/api/v1/images', image)
      .map((response: Response) => response.json())
      .catch((error:any) => Observable.throw(error.json().error || {message:"Server Er
ror"} ));
  }
}
```

Just like the **getImages** method, this method will return an **Observable**. As you see, we use **http.post** to send a **POST** request to our server. The second argument is the **image** object.

Additionally, we add one more line:

```
.catch((error:any) => Observable.throw(error.json().error || {message:"Server Error"} )
```

```
);
```

This is how we show errors if something goes wrong. You can add this line to the **getImages** method, too.

Very simple, isn't it?

One more thing, the **http.post** has a third argument, which is used to add extra information, such as request headers. For example:

```
let headers = new Headers({ 'Content-Type': 'application/json' });
let options = new RequestOptions({ headers: headers });

return this.http.post(this.commentsUrl, body, options)
```

Don't forget to import **Headers** and **RequestOptions**:

```
import {Http, Response, Headers, RequestOptions} from "@angular/http";
```

The **Http** class's documentation can be found at:

https://angular.io/docs/ts/latest/api/http/index/Http-class.html

Please note that if you're using the latest version of Angular, you don't have to add the request header and convert the object to a JSON object. The new **http.post** autodetects the type of the object, and handles everything for us. If you read some other tutorials or other books, they may tell that you have to use **JSON.stringify** to convert the object to JSON. That's not necessary anymore.

Back to our **Image Create** component, we can now update the **createImage** method as follows:

admin-image-create.component.ts

```
createImage(image) {
 this.imageService.addImage(image)
   .subscribe(
     image  => console.log(image),
     error =>  console.log(<any>error)
    );
}
```

Don't forget to import ImageService and inject it into the constructor:

```
import {ImageService} from '../../services/image.service';

constructor(private imageService: ImageService) { }
```

As you see, we use **ImageService's addImage** method to add the image. We also **subscribe** to its response. If the image is returned, we use **console.log** to display it. If there is an error, we also use **console.log** to log it!

That's how we send a **POST** request to our backend!

Fixing CORS

When sending a POST request using Angular, you might see this error:

```
Response to preflight request doesn't pass access control check: No 'Access-Control-Al
low-Origin' header is present on the requested resource
```

The reason is that there is a **preflight request** before the **actual request**. And that preflight request doesn't produce the necessary headers. This is not an Angular error, it's a browser/server error. There are many ways to fix this.

> **Note:** If you don't see this error, you may skip this section.

One of the easiest ways to fix the error is to open **index.php** (in your Laravel app), and find:

```
require __DIR__.'/../bootstrap/autoload.php';
```

Add above:

```
$allowedOrigins = array(
    '(http(s)://)?(www\.)?angularbook\.app',
    'http://localhost:4200'
);

if (isset($_SERVER['HTTP_ORIGIN']) && $_SERVER['HTTP_ORIGIN'] != '') {
    foreach ($allowedOrigins as $allowedOrigin) {
        if (preg_match('#' . $allowedOrigin . '#', $_SERVER['HTTP_ORIGIN'])) {
            header('Access-Control-Allow-Origin: ' . $_SERVER['HTTP_ORIGIN']);
            header('Access-Control-Allow-Methods: GET, PUT, POST, DELETE, OPTIONS');
```

```
            header('Access-Control-Max-Age: 1000');
            header('Access-Control-Allow-Headers: Content-Type, Authorization, X-Reque
sted-With');
            break;
        }
    }
}
```

> **Note:** if you use a different domain, you must update or add your domain into the **$allowedOrigins** array.

Lastly, open the **Cors** middleware (if you manually create it, and you're not using the CORS package), update the **handle** function as follows:

```
public function handle($request, Closure $next)
{
    return $next($request)
        ->header('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, OPTIONS')
        ->header('Access-Control-Allow-Headers', 'content-type, withcredentials, Access
-Control-Allow-Headers,
    Origin, Accept, X-Requested-With, Content-Type, Access-Control-Request-Method, Acc
ess-Control-Request-Headers');
}
```

Done. You should be good to go now.

> **Note:** The Laravel CORS package might be updated in the future to fix this bug. Be sure to check out https://github.com/barryvdh/laravel-cors/issues/127 to know more information about this issue.

Creating a new image and redirect users to another route

Our backend doesn't handle the **POST** request yet, so let's open **ImagesController** (in your Laravel app), and update the **store** method as follows:

Http/Controllers/ImagesController.php

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

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

```
            return $response;
        }

        $image = new Image(array(
            'thumbnail' => trim($request->thumbnail),
            'imageLink' => trim($request->imageLink),
            'title' => trim($request->title),
            'description' => trim($request->description),
            'user_id' => 1
        ));
        $image->save();

        $message = 'Your image has been added successfully';

        $response = Response::json([
            'message' => $message,
            'data' => $image,
        ], 201);

        return $response;
    }
```

First, we check the request. If one of the fields is empty, we send a response back, telling that "Please enter all required fields".

if we have enough information, we will create a new image:

```
$image = new Image(array(
    'thumbnail' => trim($request->thumbnail),
    'imageLink' => trim($request->imageLink),
    'title' => trim($request->title),
    'description' => trim($request->description),
    'user_id' => 1
));
$image->save();
```

After that, we send a successful response back, along with the image data.

```
$message = 'Your image has been added successfully';

$response = Response::json([
    'message' => $message,
    'data' => $image,
], 201);

return $response;
```

> **Note:** sending the image data back is optional.

It's time to add a new image! Go to the form and try to add an image.

Check the **Console** tab and you should see:

Great! You can now add an image! Check your database to make sure that a new image has been created.

In real world applications, we usually redirect users to a different page after creating something.

In order to do that, we can use **router.navigate**:

```
createImage(image) {
  this.imageService.addImage(image)
    .subscribe(
      image  => {
        console.log(image);
        this.router.navigate(['/admin/images']);
      },
      error =>  console.log(<any>error));
}
```

Don't forget to import **Router**:

```
import { Router} from '@angular/router';
```

And inject **Router** into the constructor of our component:

```
constructor(private imageService: ImageService, private router: Router) { }
```

Here is the updated **admin-image-create.component.ts**:

admin-image-create.component.ts

```
import { Component, OnInit } from '@angular/core';
import {ImageService} from '../../services/image.service';
import { Router} from '@angular/router';

@Component({
  selector: 'ng-admin-image-create',
  templateUrl: './admin-image-create.component.html',
  styleUrls: ['./admin-image-create.component.css']
})
export class AdminImageCreateComponent implements OnInit {

  constructor(private imageService: ImageService, private router: Router) { }

  ngOnInit() {
  }

  createImage(image) {
    this.imageService.addImage(image)
      .subscribe(
        image  => {
          console.log(image);
          this.router.navigate(['/admin/images']);
        },
        error =>  console.log(<any>error));
  }
}
```

> **Note:** you can remove **console.log** methods if you want.

Try to create a new image again. You will be redirected to http://localhost:4200/admin/images when the image is created successfully.

Form validation with template-driven approach

Angular provides some directives and basic validation rules that we can use to validate our form effectively.

First, we have to add **novalidate** attribute into our form. When present, it specifies that the form data should not be validated by the browser when submitted.

```
<form novalidate class="form-horizontal" (ngSubmit)="createImage(createImageForm.value)
" #createImageForm="ngForm">
```

Here is how we add some validation rules to an input field:

```
<input type="text" class="form-control" id="thumbnail" name="thumbnail" placeholder="T
humbnail of the image" ngModel #thumbnail="ngModel" required minlength="2">
```

As you see, by adding:

```
#thumbnail="ngModel" required minlength="3"
```

We assign the input data to our local variable **thumbnail**. Angular will know that **this field is an Angular Model**, the field **should be required**, and **its minimum length is 3**.

Now we can display errors by adding the following code below the field:

```
<input type="text" class="form-control" id="thumbnail" name="thumbnail" placeholder="T
humbnail of the image" ngModel #thumbnail="ngModel" required minlength="3">
<div *ngIf="thumbnail.errors?.required && thumbnail.touched" class="alert alert-danger"
>
  Thumbnail is required
</div>
<div *ngIf="thumbnail.errors?.minlength && thumbnail.touched" class="alert alert-danger
">
  Minimum of 3 characters
</div>
```

We use **ngIf** here, if the field **has errors** and **it is touched**, the error message will be displayed. Try to enter something wrong, you should see:

Here are some validators that are provided by the framework:

- required: the field should not be empty.
- minlength: the value entered has at least x characters.
- maxlength: the value entered has at most x characters.
- pattern: the value must match the regular expression.

So what is **touched**? It's an Angular form's **state**. We'll learn about it in the next section!

States of an Angular form

Angular form is smart. It can be able to tell us when the user touched the control, when the value changed, or when the value became invalid. You can think about these events as Angular form's states. Angular also automatically adds and removes some **CSS classes** on a field so that we can easily style our form or messages.

For example, a form **will be valid** and has the **ng-valid** class, when the **validators succeed**.

Here are some Angular's CSS state classes:

- valid (ng-valid): the form is valid.
- invalid (ng-invalid): the form is not valid.
- dirty (ng-dirty): the form has been modified.
- pristine (ng-pristine): the form has not been modified.
- touched (ng-touched): the form has been touched.
- untouched (ng-untouched): the form has not been touched.

You may learn more about Angular forms at:

Angular forms

Here is a little tip, using the **invalid** state, we can **disable** the submit button if the form is not valid.

Find:

```
<button type="submit" class="btn btn-primary">Create</button>
```

Modify to:

```
<button type="submit" class="btn btn-primary" [disabled]="createImageForm.invalid">Cre
ate</button>
```

Check the form again!

The submit button is disabled if the form is not valid! Pretty cool, isn't it?

We should have a working form now. Before going to the next section, let's make our **Cancel** button work!

Find:

```
<button class="btn btn-default">Cancel</button>
```

Update to:

```
<a [routerLink]="['/admin/']" class="btn btn-default"> Cancel</a>
```

I also add some validation rules for other fields. Here is our updated **admin-image-create.component.html**:

admin-image-create.component.html

```
<div class="col-md-10 col-md-offset-1">
  <div>
    <a [routerLink]="['/admin/']" class="btn btn-default"> Back</a>
  </div>
</div>

<div class="col-md-10 col-md-offset-1">
  <div class="well well bs-component">
    <form novalidate class="form-horizontal" (ngSubmit)="createImage(createImageForm.v
```

```
alue)" #createImageForm="ngForm">
      <fieldset>
        <legend>Add a new image</legend>
        <div class="form-group">
          <label for="thumbnail" class="col-lg-2 control-label">Thumbnail</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="thumbnail" name="thumbnail" pl
aceholder="Thumbnail of the image" ngModel #thumbnail="ngModel" required minlength="3">

            <div *ngIf="thumbnail.errors?.required && thumbnail.touched" class="alert
alert-danger">
              Thumbnail is required
            </div>
            <div *ngIf="thumbnail.errors?.minlength && thumbnail.touched" class="alert
 alert-danger">
              Minimum of 3 characters
            </div>
          </div>
        </div>
        <div class="form-group">
            <label for="imageLink" class="col-lg-2 control-label">Image Link</label>
            <div class="col-lg-10">
              <input type="text" class="form-control" id="imageLink" name="imageLink"
placeholder="Link of the image" ngModel #imageLink="ngModel" required>
              <div *ngIf="imageLink.errors?.required && imageLink.touched" class="aler
t alert-danger">
                ImageLink is required
              </div>
            </div>
        </div>
        <div class="form-group">
          <label for="title" class="col-lg-2 control-label">Title</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="title" name="title" placeholder
="Title" ngModel #title="ngModel" required>
            <div *ngIf="title.errors?.required && title.touched" class="alert alert-da
nger">
              Title is required
            </div>
          </div>
        </div>
        <div class="form-group">
          <label for="description" class="col-lg-2 control-label">Description</label>
          <div class="col-lg-10">
            <textarea class="form-control" rows="3" id="description" name="description"
 placeholder="Description of the image" ngModel #description="ngModel" required></text
area>
            <div *ngIf="description.errors?.required && description.touched" class="al
ert alert-danger">
              Description is required
            </div>
          </div>
        </div>
```

```
        <div class="form-group">
          <div class="col-lg-10 col-lg-offset-2">
            <a [routerLink]="['/admin/']" class="btn btn-default"> Cancel</a>
            <button type="submit" class="btn btn-primary" [disabled]="createImageForm.
invalid">Create</button>
          </div>
        </div>
      </fieldset>
    </form>
  </div>
</div>
```

Click on the **Cancel** button and we'll go back to http://localhost:4200/admin

Editing an image

In this section, we're going to create a new **Admin Image Edit** Component that takes in a **route parameter**. When you're in the **admin** directory, run this command to create the component:

```
ng g c admin-image-edit
```

```
admin [master] ⚡ng g c admin-image-edit
installing component
  create src/app/admin/admin-image-edit/admin-image-edit.component.css
  create src/app/admin/admin-image-edit/admin-image-edit.component.html
  create src/app/admin/admin-image-edit/admin-image-edit.component.spec.ts
  create src/app/admin/admin-image-edit/admin-image-edit.component.ts
  update src/app/app.module.ts
```

So, what is a **route parameter**?

Route Parameters

Route parameters allow us to **pass values** from our **URL** to our **component** so that we can dynamically change the content of our view. For example, if we want to edit **image 1**, we can visit **/admin/images/edit/1**. If we want to edit **image 2**, we can visit **/admin/images/edit/2**.

As you see, by swapping out any number in our URL, we can display a different view content.

> **Tip:** if you're working with Laravel or other frameworks, this should be a basic concept, right?

In this section, we will learn how to display different pages (view content) based on the URL.

First, open **admin.routes.ts**, and add this route:

```
export const adminRoutes: Routes = [
  { path: '', component: DashboardComponent},
  { path: 'images', component: AdminImageListComponent},
  { path: 'images/create', component: AdminImageCreateComponent},
  { path: 'images/edit/:id', component: AdminImageEditComponent }
];
```

Don't forget to import **AdminImageEditComponent**:

```
import {AdminImageEditComponent} from './admin-image-edit/admin-image-edit.component';
```

As you see, we use **images/edit/:id**. The **':'** denotes that this is a route parameter, and Angular should get this **id** from the URL.

So, how do we get this route parameter (id)?

Open **admin-image-edit.component.ts**, and find:

```
export class AdminImageEditComponent implements OnInit {
```

Add below:

```
id: any;
params: any;
```

We create two variables: **id** and **params**.

> **Note:** you may use different names if you want.

After that, we inject **ActivatedRoute** from **@angular/router** into our component.

```
constructor(private activatedRoute: ActivatedRoute) { }
```

Don't forget to import **ActivateRoute**:

```
import {ActivatedRoute} from '@angular/router';
```

Here is how we get the **id** of the page:

```
ngOnInit() {
  this.params = this.activatedRoute.params.subscribe(params => this.id = params['id']);

}
```

Basically, when using **activatedRoute**, parameters are wrapped in an **Observable**. We **subscribe** to that **Observable** using:

```
this.activatedRoute.params.subscribe()
```

When the route changes, we will know immediately. We can grab the value (which is the **id**) using **params['id']**, and assign it to our **id** variable:

```
params => this.id = params['id']
```

Additionally, to prevent memory leaks, we should **unsubscribe** the **params** when our component is destroyed.

Find:

```
ngOnInit() {
  this.params = this.activatedRoute.params.subscribe(params => this.id = params['id']);

}
```

Add below:

```
ngOnDestroy() {
  this.params.unsubscribe();
}
```

Find:

```
export class AdminImageEditComponent implements OnInit {
```

Update to:

```
export class AdminImageEditComponent implements OnInit, OnDestroy {
```

Importing **OnDestroy** into our component:

```
import {Component, OnInit, OnDestroy} from '@angular/core';
```

Here is the new **admin-image-edit.component.ts**:

admin-image-edit.component.ts

```
import {Component, OnDestroy, OnInit} from '@angular/core';
import {ActivatedRoute} from '@angular/router';

@Component({
  selector: 'ng-admin-image-edit',
  templateUrl: './admin-image-edit.component.html',
  styleUrls: ['./admin-image-edit.component.css']
})
export class AdminImageEditComponent implements OnInit, OnDestroy {
  id: any;
  params: any;

  constructor(private activatedRoute: ActivatedRoute) { }

  ngOnInit() {
    this.params = this.activatedRoute.params.subscribe(params => this.id = params['id']
);
  }

  ngOnDestroy() {
    this.params.unsubscribe();
  }
}
```

Finally, open **admin-image-edit.component.html**, and find:

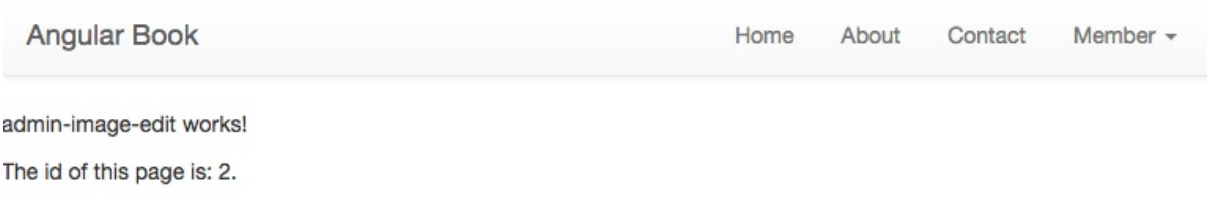
```
<p>
  admin-image-edit works!
</p>
```

Add below:

```
<p>The id of this page is: {{id}}.</p>
```

We use `{{id}}` to display the id of the page.

At this time, when visiting http://localhost:4200/admin/images/edit/2, we should see:



Try to change the id of the URL, you'll see that we can still grab the **id** and display it!

When we have the **id**, we can **send a request** to our API to get the right data that we want.

Getting a single image using Laravel

We don't have an API to get a single image yet, so let's create it.

Open **ImagesController.php**, and update the **show** method as follows:

```
public function show($id)
{
    $image = Image::find($id);

    if(!$image){
        return Response::json([
            'error' => [
                'message' => "Cannot find the image."
            ]
        ], 404);
    }

    return Response::json($image, 200);
}
```

First, we will find the image based on its id:

```
$image = Image::find($id);
```

If we can't find it, we'll send a error message back:

```
if(!$image){
    return Response::json([
        'error' => [
            'message' => "Cannot find the image."
        ]
    ], 404);
}
```

If the image is found, we should see the **200** response and the image data:

```
return Response::json($image, 200);
```

Good job! Visit http://angularbook.app/api/v1/images/2 and you should see:


```
{
    id: 2,
    title: "Nihil tempora nobis dolorum eveniet quo a odio.",
    description: "Est sunt ut cupiditate sapiente numquam aut est officia. Quasi ut quisquam
    voluptas molestias asperiores eaque quae laudantium. Non enim excepturi culpa sint.
    Voluptatem temporibus accusantium cumque veritatis consequuntur. Ut consectetur cum et ullam
    voluptatibus. Beatae ut dolorem recusandae et omnis omnis placeat. Deserunt pariatur nam et
    magnam qui dolor aliquid est. Recusandae quos unde assumenda aut sequi sunt. Omnis dicta
    nobis facere accusantium. Omnis autem sunt harum et. Quis nam illum harum et. Et ad aut nobis
    vel molestiae nobis. Natus cumque adipisci vero provident sit. Eum dolor sequi debitis
    inventore sint non quae similique. Omnis laborum id voluptates est exercitationem et
    corporis. Inventore accusamus sint voluptatum animi. Voluptas eos in omnis possimus nisi
    consequatur. Labore molestias quod in esse numquam officia pariatur debitis. Est illum omnis
    qui sit.",
    thumbnail: "https://angularbooks.com/img/angular4/img2.jpg",
    imageLink: "https://angularbooks.com/img/angular4/img2-l.jpg",
    user_id: 4,
    created_at: "2017-03-26 03:29:23",
    updated_at: "2017-03-26 03:29:23"
}
```


> **Tip:** You may use **Postman** to get the image.

Updating an image using Laravel

As you may have guessed, we also need an API to update an image.

Open **ImagesController.php**, and update the **update** method as follows:

```
public function update(Request $request)
```

```
{
    if ((!$request->title) || (!$request->thumbnail) || (!$request->imageLink)) {

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

    $image = Image::find($request->id);

    if(!$image){
        return Response::json([
            'error' => [
                'message' => "Cannot find the image."
            ]
        ], 404);
    }

    $image->thumbnail = trim($request->thumbnail);
    $image->imageLink = trim($request->imageLink);
    $image->title = trim($request->title);
    $image->description = trim($request->description);
    $image->save();

    $message = 'Your image has been updated successfully';

    $response = Response::json([
        'message' => $message,
        'data' => $image,
    ], 201);

    return $response;
}
```

Just like how we create an image, we check the request first to make sure that all information is correct:

```
if ((!$request->title) || (!$request->thumbnail) || (!$request->imageLink)) {

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

After that, we'll find the image, based on the **requested id**:

```
$image = Image::find($request->id);
```

If we can't find the image, we'll send back an error:

```
if(!$image){
    return Response::json([
        'error' => [
            'message' => "Cannot find the image."
        ]
    ], 404);
}
```

If everything is ok, we'll update the image and return the **201** response:

```
$image->thumbnail = trim($request->thumbnail);
$image->imageLink = trim($request->imageLink);
$image->title = trim($request->title);
$image->description = trim($request->description);
$image->save();

$message = 'Your image has been updated successfully';

$response = Response::json([
    'message' => $message,
    'data' => $image,
], 201);
```

Getting a single image using Angular

When editing an image, we have to send a request to our backend to get the image data, so that we can display information on our edit form.

First, let's add a **getImage** method into our **Image Service**:

services.image.services.ts

```
getImage(id: String): Observable<Image[]> {
  return this.http.get('http://angularbook.app/api/v1/images/' + id)
    .map((response: Response) => response.json());
}
```

This method is simple, it sends a request to http://angularbook.app/api/v1/images/imageID to get the image data.

Next, open **admin-image-edit.component.ts**, and find:

```
this.params = this.activatedRoute.params.subscribe(params => this.id = params['id']);
```

Add below:

```
this.imageService.getImage(this.id).subscribe(
  data => {
    console.log(data);
  },
  error =>  console.log(<any>error));
```

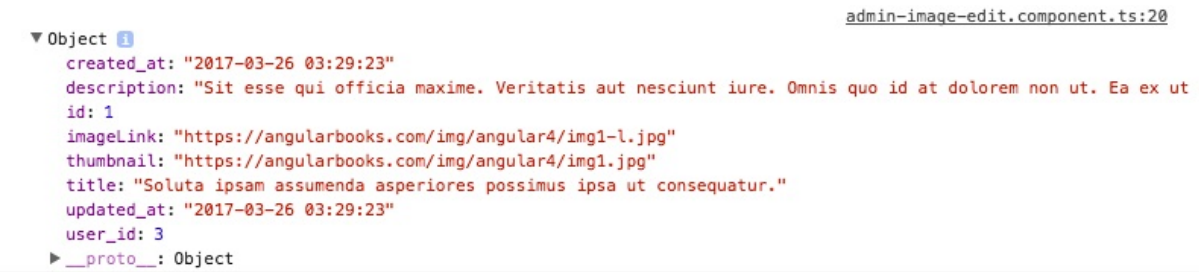
Don't forget to import the **Image Service**:

```
import {ImageService} from '../../services/image.service';
```

Then inject it into our component:

```
constructor(private activatedRoute: ActivatedRoute, private imageService: ImageService)
 { }
```

If you visit http://localhost:4200/admin/images/edit/1, you should see your image data (which is an object) in the **Console** log:



> **Tip:** Instead of showing an **Edit Image** form, we may use this technique to show a single image.

It's time to use the data to display default values of the fields!

You may think that we can do something like this:

```
<input type="text" class="form-control" id="title" name="title" placeholder="Title" ng
Model #title="ngModel" required value="title">
```

It still works. However, this is not a practical way, and we can't use Angular to completely control the form. It's better to use **ngModel** to bind the model (which is the image) to the form control. In this case, we should use **two-way binding** technique!

Introducing two-way binding

What is two-way binding?

According to W3Schools:

> Two-way Binding. Data binding in AngularJS is the synchronization between the model and the view. When data in the model changes, the view reflects the change, and when data in the view changes, the model is updated as well.

Simply put, we can create an **Image** model, and bind it to the form. When we change the data of the **Image** model, our form will be updated, and when we modify the form, our **Image** model will be updated as well.

Don't worry if you don't understand the concept yet. I'll show you how to use two-way binding soon.

First, open **admin-image-edit.component.ts**, and find:

```
constructor(private activatedRoute: ActivatedRoute, private imageService: ImageService)
 { }
```

Add above to create a new image:

```
image = new Image('id', 'title', 'description', 'thumbnail', 'imageLink');
```

Be sure to import **Image** into our component:

```
import { Image } from '../../models/image';
```

This is how we assign our requested data to the image that we've just created:

Find:

```
this.imageService.getImage(this.id).subscribe(
  data => {
    console.log(data);
```

Add below:

```
this.image.description = data['description'];
this.image.title = data['title'];
this.image.imageLink = data['imageLink'];
this.image.thumbnail = data['thumbnail'];
this.image.id = data['id'];
```

Let's proceed to bind our image model to the form. Open **admin-image-edit.component.html**, and update it as follows:

admin-image-edit.component.html

```
<div class="col-md-10 col-md-offset-1">
  <div>
    <a [routerLink]="['/admin/images']" class="btn btn-default"> Back</a>
  </div>
</div>

<div class="col-md-10 col-md-offset-1">
  <div class="well well bs-component">
    <form novalidate class="form-horizontal" (ngSubmit)="updateImage(image)" #editImag
eForm="ngForm">
      <fieldset>
        <legend>Updating: {{image.title}}</legend>
        <div class="form-group">
          <label for="thumbnail" class="col-lg-2 control-label">Thumbnail</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="thumbnail" name="thumbnail" pl
aceholder="Thumbnail of the image" [(ngModel)]="image.thumbnail" #thumbnail="ngModel"
required minlength="3">
            <div *ngIf="thumbnail.errors?.required && thumbnail.dirty" class="alert al
ert-danger">
              Thumbnail is required
            </div>
            <div *ngIf="thumbnail.errors?.minlength && thumbnail.touched" class="alert
 alert-danger">
              Minimum of 3 characters
            </div>
          </div>
        </div>
        <div class="form-group">
          <label for="imageLink" class="col-lg-2 control-label">Image Link</label>
          <div class="col-lg-10">
```

```
            <input type="text" class="form-control" id="imageLink" name="imageLink" pl
aceholder="Link of the image" [(ngModel)]="image.imageLink" #imageLink="ngModel" requi
red>
            <div *ngIf="imageLink.errors?.required && imageLink.dirty" class="alert al
ert-danger">
              ImageLink is required
            </div>
          </div>
        </div>
        <div class="form-group">
          <label for="title" class="col-lg-2 control-label">Title</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="title" name="title" placeholder
="Title" [(ngModel)]="image.title" #title="ngModel" required>
            <div *ngIf="title.errors?.required && title.dirty" class="alert alert-dang
er">
              Title is required
            </div>
          </div>
        </div>
        <div class="form-group">
          <label for="description" class="col-lg-2 control-label">Description</label>
          <div class="col-lg-10">
            <textarea class="form-control" rows="3" id="description" name="description"
 [(ngModel)]="image.description" #description="ngModel" required></textarea>
            <div *ngIf="description.errors?.required && description.dirty" class="aler
t alert-danger">
              Description is required
            </div>
          </div>
        </div>

        <div class="form-group">
          <div class="col-lg-10 col-lg-offset-2">
            <a [routerLink]="['/admin/']" class="btn btn-default"> Cancel</a>
            <button type="submit" class="btn btn-primary" [disabled]="editImageForm.in
valid">Update</button>
          </div>
        </div>
      </fieldset>
    </form>
  </div>
</div>
```

This file is very similar to the **admin-image-create.component.html** file.

I've changed the **createImage** method to **updateImage** method:

```
<form novalidate class="form-horizontal" (ngSubmit)="updateImage(image)" #editImageForm
="ngForm">
```

To bind our image model to the input field, we can do like this:

```
<input type="text" class="form-control" id="thumbnail" name="thumbnail" placeholder="T
humbnail of the image" [(ngModel)]="image.thumbnail"
#thumbnail="ngModel" required minlength="3">
```

As you see, we use a new syntax **[( )]**:

```
[(ngModel)]="image.thumbnail"
```

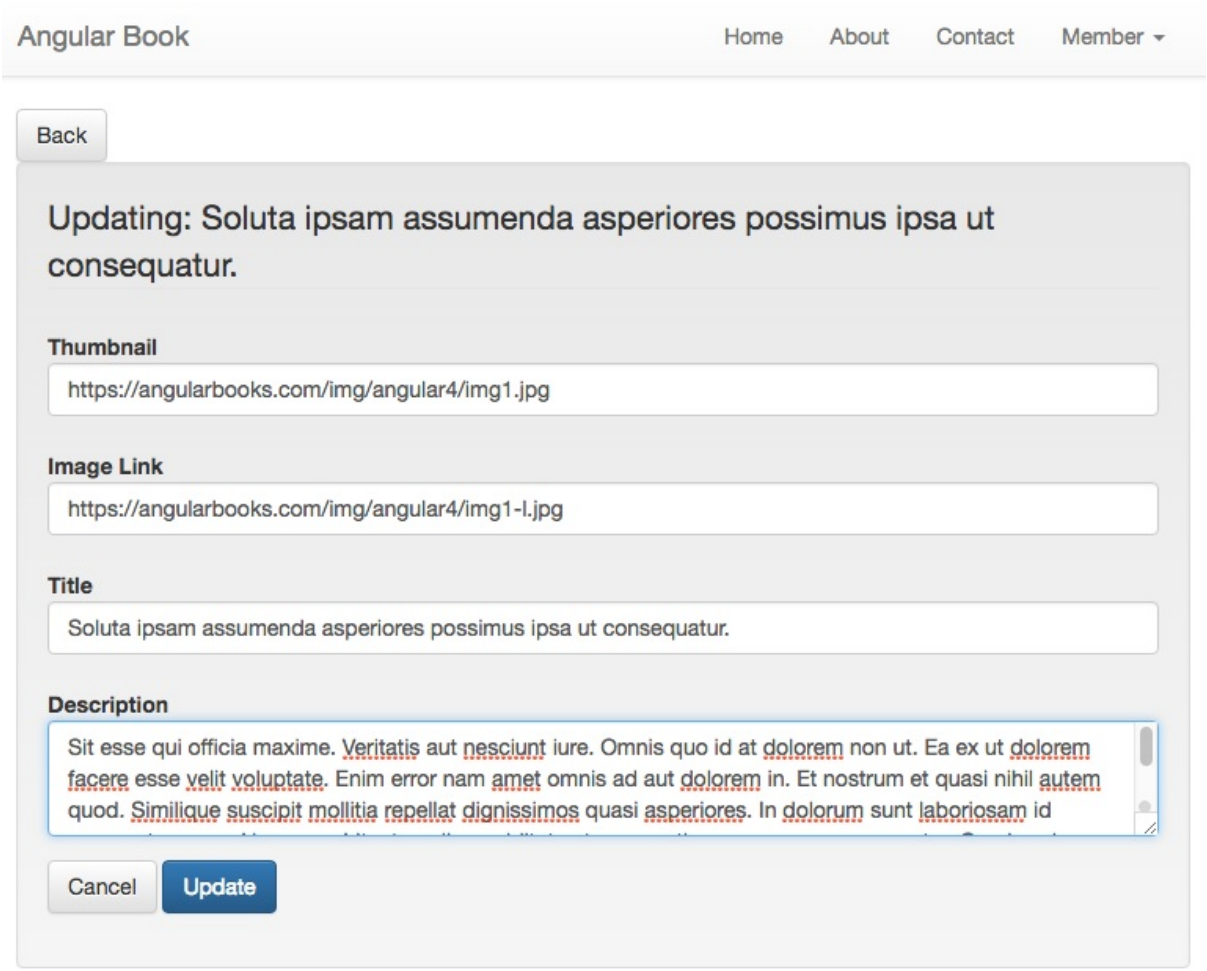
This means we want to use **two-way binding** here.

> **Tip:** Familiar? We've used this **[ ]** syntax in Chapter 2. So, use **[ ]** for one-way binding, and **[( )]** for two-way binding.

We do the same thing for other fields:

```
<input type="text" class="form-control" id="imageLink" name="imageLink" placeholder="L
ink of the image" [(ngModel)]="image.imageLink" #imageLink="ngModel" required>

<input type="text" class="form-control" id="title" name="title" placeholder="Title" [(
ngModel)]="image.title" #title="ngModel" required>

<textarea class="form-control" rows="3" id="description" name="description" [(ngModel)]
="image.description" #description="ngModel" required></textarea>
```

Check the form in your browser:



Great! Our form is filled with data!

You may notice that we use **image.title** as the caption of the form:

```
<legend>Updating: {{image.title}}</legend>
```

When we edit the title, the caption is automatically updated as well!

This is the power of **two-way binding**! By using this technique, you can create interactive forms easily in Angular!

We don't have the **updateImage** method yet, so let's create it. Open **admin-image-edit.component.ts**, and find:

```
ngOnDestroy() {
  this.params.unsubscribe();
}
```

Add below to create the **updateImage** method:

```
updateImage(image) {
  this.imageService.updateImage(image)
    .subscribe(
      image  => {
        console.log(image);
      },
      error =>  console.log(<any>error));
}
```

As you may guess, we should add a new **updateImage** method to the **ImageService**:

image.service.ts

```
updateImage(image: Object): Observable<Image[]> {
  const apiUrl = 'http://angularbook.app/api/v1/images';
  const url = `${apiUrl}/${image["id"]}`;
  return this.http.put(url, image)
    .map((response: Response) => response.json())
    .catch((error:any) => Observable.throw(error.json().error || {message:"Server Erro
r"}));
}
```

To update the image, we use **http.put** to send a **PUT** request to our backend.

Just a little tip, you may also write the URL this way:

```
const apiUrl = 'http://angularbook.app/api/v1/images';
const url = `${apiUrl}/${image["id"]}`;
```

Or this way:

```
const url = 'http://angularbook.app/api/v1/images/' + image["id"]'
```

Visit our form again. Try to modify an image, and then click the **Update** button to update the image.

Check the **Console** tab:

Congratulations! The image has been updated successfully!

Displaying success or error messages

There are many ways to display an error or success message when submitting a form. In this section, we will learn how to use **ngIf** and some custom variables to create status messages, that let us know if our request is successful or not.

Open **admin-image-edit.component.ts**, and add these variables:

Find:

```
export class AdminImageEditComponent implements OnInit {
```

Add below:

```
status: string;
message: string;
```

And then we only need to update the **updateImage** method as follows:

```
updateImage(image) {
  this.imageService.updateImage(image)
    .subscribe(
      image  => {
```

```
          console.log(image);
          this.status = "success";
          this.message = image['message'];
        },
        error =>  {
          console.log(<any>error);
          this.status = "error";
          this.message = error['message'];
        }
        );
  }
```

As you see, to display a **success message**, the **status** variable will be set to **"success"**. To display an error message, the **status** variable will be set to **"error"**.

We can show the messages based on the server's response:

Success message:

```
this.message = image['message'];
```

Error message:

```
this.message = error['message'];
```

Or if we want, we can customize the message like this:

```
this.message = 'Image has been updated successfully';
```

Next step, open **admin-image-edit.component.html**, and find:

```
<div class="well well bs-component">
  <form novalidate class="form-horizontal" (ngSubmit)="updateImage(image)" #editImageF
orm="ngForm">
```

Add above:

```
<div *ngIf="status=='success'" class="alert alert-success" role="alert"> {{ message }}
</div>
<div *ngIf="status=='error'" class="alert alert-danger" role="alert"> {{ message }} </
div>
```

Let's break things down a bit.

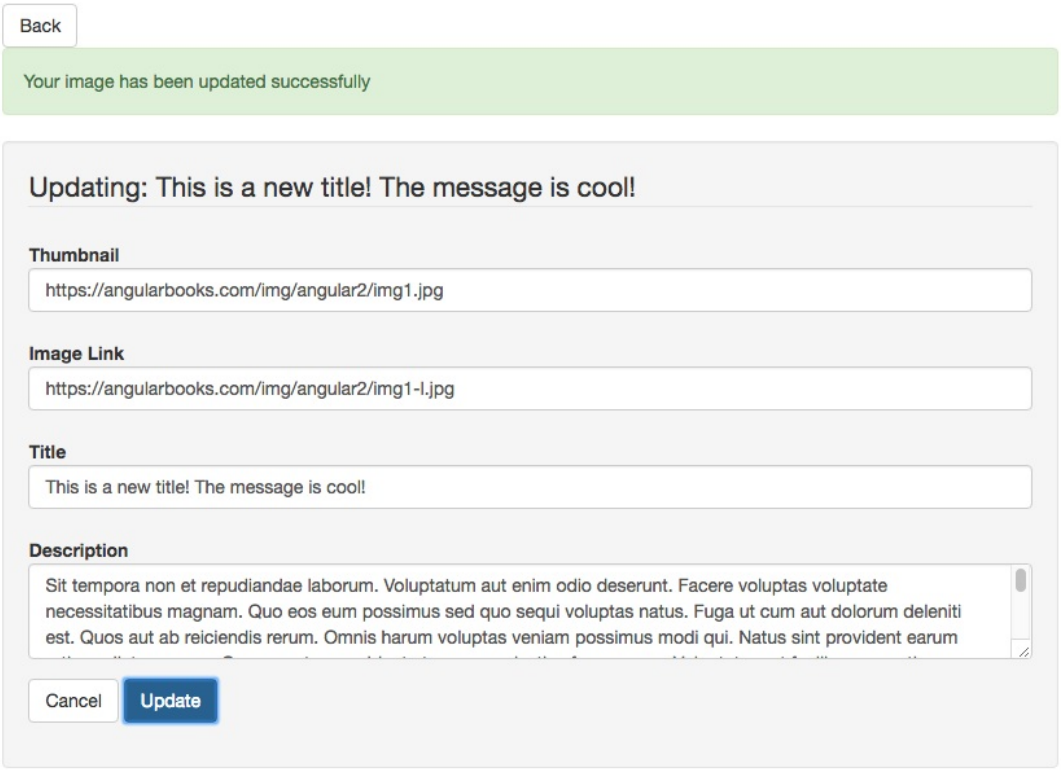
```
<div *ngIf="status=='success'" class="alert alert-success" role="alert"> {{ message }}
</div>
```

We use **ngIf** to check if the **status** variable is set to **"success"**. If it is, we display the success message.

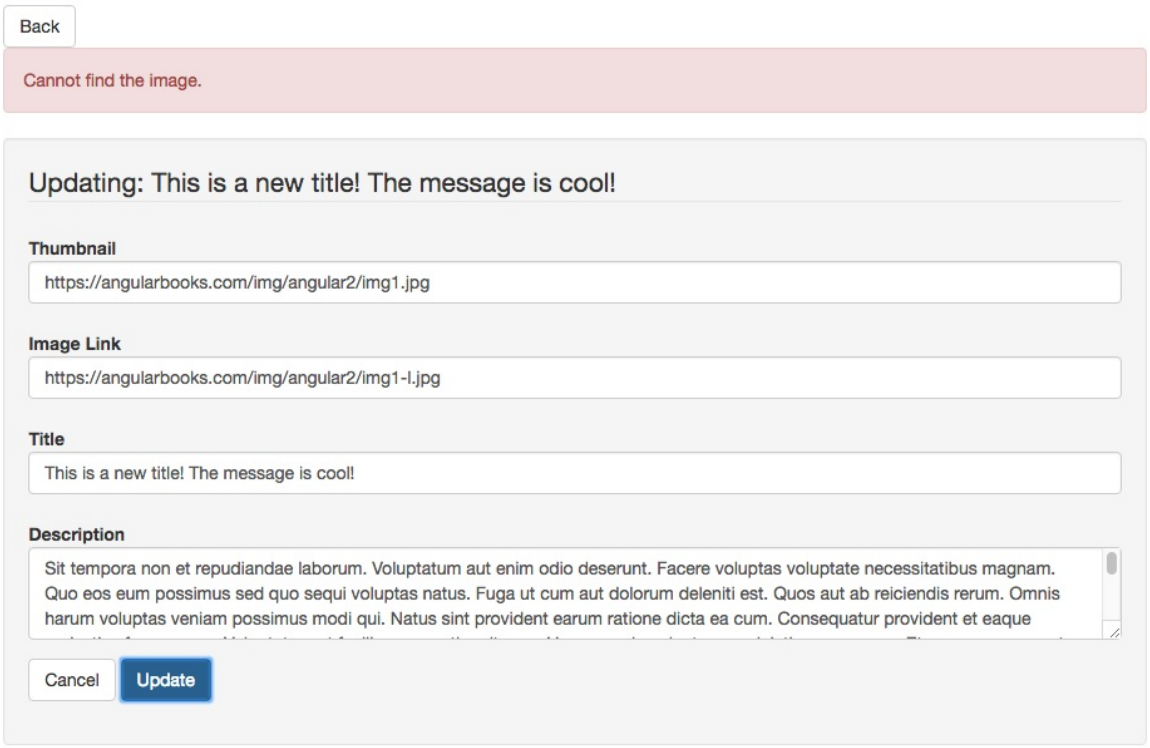
```
<div *ngIf="status=='error'" class="alert alert-danger" role="alert"> {{ message }} </
div>
```

If the value of the **status** variable is **"error"**, we display the error message.

Now when we open our app in the browser and try to update the form, you should see a nice message:



If the server has an error, we should see an error message:



As we've seen above, this technique is simple, but it's very useful to display status messages!

> **Tip:** How about updating the **Create Image form** yourself to display the status messages?

Before going to the next section, open **admin-image-list.component.html**, and find:

```
<a [routerLink]="['/admin/images/edit']" class="btn btn-info"> Edit</a>
<a [routerLink]="['/admin/images/delete']" class="btn btn-danger"> Delete</a>
```

Change to:

```
<a [routerLink]="['/admin/images/edit', image.id]" class="btn btn-info"> Edit</a>
<a [routerLink]="['/admin/images/delete', image.id]" class="btn btn-danger"> Delete</a>
```

Now when we click on the **Edit** or the **Delete** button, we will be able to edit or delete an image.

Currently, the delete button doesn't work yet, so let's make it work!

Deleting an image using Laravel

First things first, let's update the **destroy** action to handle a **DELETE** request.

Open **ImagesController.php**, update the **destroy** action as follows:

```
public function destroy($id)
{
    $image = Image::find($id);

    if(!$image){
        return Response::json([
            'error' => [
                'message' => "Cannot find the image."
            ]
        ], 404);
    }

    $image->delete();

    $message = 'Your image has been deleted successfully';

    $response = Response::json([
        'message' => $message,
    ], 201);

    return $response;
}
```

Again, we will find the image based on the requested **ID**.

If the image is found, we will delete it by using:

```
$image->delete();
```

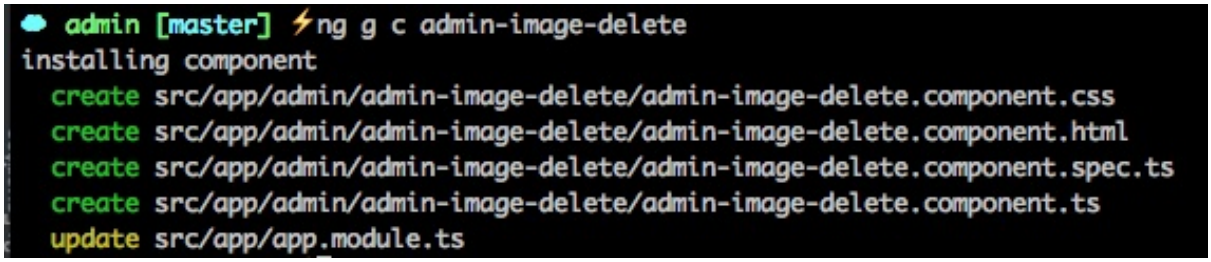
After that, we send back a success response.

```
$message = 'Your image has been deleted successfully';

$response = Response::json([
    'message' => $message,
], 201);

return $response;
```

Deleting an image using Angular

As usual, we will need to create the **Admin Image Delete** component first. While you're in the **admin** directory, run this command:

```
ng g c admin-image-delete
```



Next, open **admin.routes.ts**, and add a new route:

```
export const adminRoutes: Routes = [
  { path: '', component: DashboardComponent},
  { path: 'images', component: AdminImageListComponent},
  { path: 'images/create', component: AdminImageCreateComponent},
  { path: 'images/edit/:id', component: AdminImageEditComponent },
  { path: 'images/delete/:id', component: AdminImageDeleteComponent },
];
```

Remember to import the **AdminImageDeleteComponent**:

```
import {AdminImageDeleteComponent} from './admin-image-delete/admin-image-delete.compo
nent';
```

Open **admin-image-delete.component.ts**. Just like we have done in the previous chapter, we will create two new variables:

```
id: any;
params: any;
```

We also use **ActivatedRoute**, **Image Service** and **Router** in this component:

```
constructor(private activatedRoute: ActivatedRoute, private imageService: ImageService,
 private router: Router) { }
```

Don't forget to import them:

```
import {ActivatedRoute, Router} from '@angular/router';
import {ImageService} from '../../services/image.service';
```

Next, update the **ngOnInit** method as follows:

```
ngOnInit() {
  this.params = this.activatedRoute.params.subscribe(params => this.id = params['id']);

  this.imageService.deleteImage(this.id).subscribe(
    data => {
      console.log(data);
      this.router.navigate(['/admin/images']);
    },
    error =>  console.log(<any>error));
}
```

In the code above, we get the requested **id** by using:

```
this.params = this.activatedRoute.params.subscribe(params => this.id = params['id']);
```

Once we have the id, we use the **Image Service's deleteImage** method to send a **DELETE** request to our backend:

```
this.imageService.deleteImage(this.id).subscribe(
    data => {
      console.log(data);
      this.router.navigate(['/admin/images']);
    },
    error =>  console.log(<any>error));
}
```

If we successfully delete the image, we will be redirected to the **admin/images** route.

Additionally, to prevent memory leaks, we should **unsubscribe** the **params** when our component is destroyed. Add this method below the **ngOnInit()** method:

```
ngOnDestroy() {
  this.params.unsubscribe();
}
```

Find:

```
export class AdminImageDeleteComponent implements OnInit {
```

Update to:

```
export class AdminImageDeleteComponent implements OnInit, OnDestroy {
```

Importing **OnDestroy** into our component:

```
import {Component, OnInit, OnDestroy} from '@angular/core';
```

Here is the updated **admin-image-delete.component.ts**:

admin-image-delete.component.ts

```
import {Component, OnDestroy, OnInit} from '@angular/core';
import {ActivatedRoute, Router} from '@angular/router';
import {ImageService} from '../../services/image.service';

@Component({
  selector: 'ng-admin-image-delete',
  templateUrl: './admin-image-delete.component.html',
  styleUrls: ['./admin-image-delete.component.css']
})
export class AdminImageDeleteComponent implements OnInit, OnDestroy {
  id: any;
  params: any;

  constructor(private activatedRoute: ActivatedRoute, private imageService: ImageServi
ce, private router: Router) { }

  ngOnInit() {
```

```
    this.params = this.activatedRoute.params.subscribe(params => this.id = params['id']
);
    this.imageService.deleteImage(this.id).subscribe(
      data => {
        console.log(data);
        this.router.navigate(['/admin/images']);
      },
      error =>  console.log(<any>error));
  }

  ngOnDestroy() {
    this.params.unsubscribe();
  }

}
```

Last step, open **image.service.ts**, and add a new **deleteImage** method:

```
deleteImage(id: String): Observable<Image[]> {
  let apiUrl = 'http://angularbook.app/api/v1/images';
  let url = `${apiUrl}/${id}`;
  return this.http.delete(url)
    .map((response: Response) => response.json())
    .catch((error:any) => Observable.throw(error.json().error || {message:"Server Erro
r"}));
}
```

Try it out! Go to http://localhost:4200/admin/images, and try to delete an image!

Congratulations! You now have a full working **CRUD** (Create Read Update Delete) application!

Forms in Angular: code-driven approach

As mentioned before, Angular also provides us a **code-driven** way to build forms. In this section, we're going to create a **Create User Form**, that we can use to add a new user.

To use this **code-driven** method, you have to import **ReactiveFormsModule**, instead of **FormsModule**.

Open **app.module.ts**, and make sure that we've imported the **ReactiveFormsModule**:

```
import {FormsModule, ReactiveFormsModule} from '@angular/forms';
```

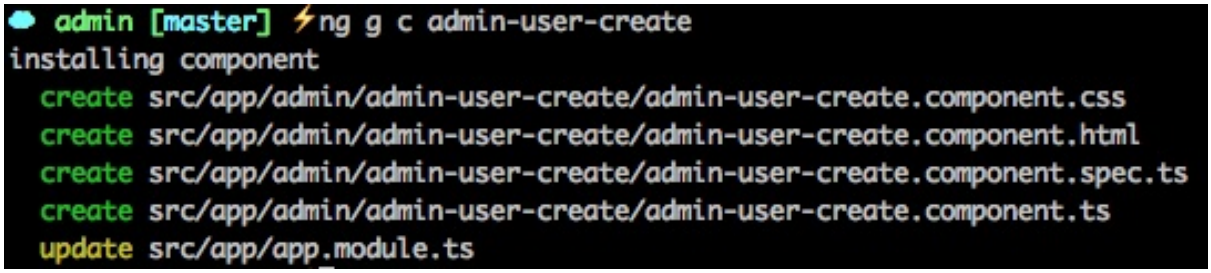
And add it into the **imports** array:

```
imports: [
  routes,
  BrowserModule,
  FormsModule,
  ReactiveFormsModule,
  HttpModule
],
```

Great! Now we can start building forms!

First, let's generate **Admin User Create** component. While you're in the **admin** directory, run this command:

```
ng g c admin-user-create
```



Next, open admin.routes.ts, and add this route:

```
export const adminRoutes: Routes = [
  { path: '', component: DashboardComponent},
  { path: 'images', component: AdminImageListComponent},
  { path: 'images/create', component: AdminImageCreateComponent},
  { path: 'images/edit/:id', component: AdminImageEditComponent },
  { path: 'images/delete/:id', component: AdminImageDeleteComponent },
  { path: 'users/create', component: AdminUserCreateComponent},
];
```

Don't forget to import **Admin User Create Component**:

```
import {AdminUserCreateComponent} from './admin-user-create/admin-user-create.componen
t';
```

In Angular, we can use a helper class called **FormBuilder** to create **Form Group** and **Form Control**. Usually, a form group contains several form controls. We use form controls to **control** input fields of the form.

To make things clearer, let's build a form using the **FormBuilder** class!

Open **admin-user-create.component.ts**, and import the following:

```
import {FormBuilder, FormGroup} from "@angular/forms";
```

Once we have **FormBuilder** and **FormGroup**, we can create a userForm!

Find:

```
export class AdminUserCreateComponent implements OnInit {
```

Add below:

```
userForm: FormGroup;
```

Find:

```
constructor() { }
```

Replace with:

```
constructor(fb: FormBuilder) {
  this.userForm = fb.group({
    email: fb.control(''),
    name:fb.control(''),
    password: fb.control('')
  });
}
```

As you see, we just created a new **userForm** object, which is of type **FormGroup**:

```
userForm: FormGroup;
```

After that, we inject **FormBuilder** into our constructor:

```
constructor(fb: FormBuilder) {
```

Then we use **fb.group** (**FormGroup.group**) to create a group, which contains several controls: email, username, etc.

fb.control('') means the value of the field should be empty.

> **Note: fb** is a custom name. You may use any name that you like.

Finally, we can bind those controls to input fields in our view (**admin-user-create.component.html**) as follows:

Open **admin-user-create.component.html**, and replace everything with:

admin-user-create.component.html

```
<div class="col-md-10 col-md-offset-1">
  <div>
    <a [routerLink]="['/admin/']" class="btn btn-default"> Back</a>
  </div>
</div>

<div class="col-md-10 col-md-offset-1">
  <div class="well well bs-component">
    <form novalidate class="form-horizontal" (ngSubmit)="createUser()" [formGroup]="us
erForm">
      <fieldset>
        <legend>Add a new user</legend>
        <div class="form-group">
          <label for="email" class="col-lg-2 control-label">Email</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="email" name="email" placeholder
="Email of the user" formControlName="email">
          </div>
        </div>
        <div class="form-group">
          <label for="name" class="col-lg-2 control-label">Full name</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="name" name="name" placeholder="
Name of the user" formControlName="name">
          </div>
        </div>
        <div class="form-group">
          <label for="password" class="col-lg-2 control-label">Password</label>
          <div class="col-lg-10">
            <input type="password" class="form-control" id="password" name="password"
placeholder="Password" formControlName="password">
          </div>
        </div>
```

```
        <div class="form-group">
          <div class="col-lg-10 col-lg-offset-2">
            <a [routerLink]="['/admin/']" class="btn btn-default"> Cancel</a>
            <button type="submit" class="btn btn-primary">Create</button>
          </div>
        </div>
      </fieldset>
    </form>
  </div>
</div>
```

Let's break things down again!

We can use **formGroup** directive to bind our form to our **userForm** object:

```
<form novalidate class="form-horizontal" (ngSubmit)="createUser()" [formGroup]="userFo
rm">
```

We also use **ngSubmit** here, so when we hit the **submit button**, the **createUser()** method will be executed.

After that, we can bind a **Form Control** to our input field by using the **formControlName** directive:

```
<input type="text" class="form-control" id="email" name="email" placeholder="Email of
the user" formControlName="email">
```

> **Note:** we don't use the **[ ]** here.

It's that simple!

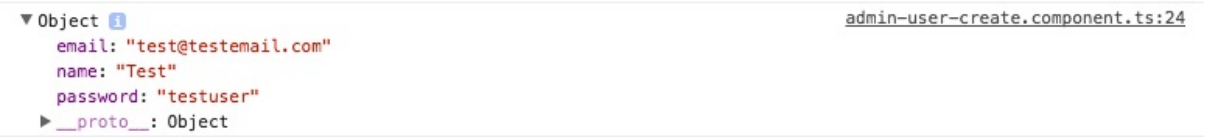
We don't have the **createUser** method yet, so let's create it.

Open **admin-user-crete.component.ts**, and add:

```
createUser() {
  console.log(this.userForm.value);
}
```

Go to http://localhost:4200/admin/users/create and try to enter something:

We should see this object in our **Console** tab:



Form validation with code-driven approach

In the **code-driven** form, we use a **Validator** to validate our form. Here are some validation rules that are provided by default:

- **Validators.required**: the control should not be empty.
- **Validators.minLength(number)**: the control should have at least **number** of characters.
- **Validators.maxLength(number)**: the control should have at most **number** of characters.
- **Validators.pattern(pattern)**: the control must match the regular expression **pattern**.

To use **Validator**, we have to import it first.

Open **admin-user-create.component.ts**, and find:

```
import {FormBuilder, FormGroup} from '@angular/forms';
```

Update to

```
import {FormBuilder, FormGroup, Validators} from '@angular/forms';
```

After that, we can add some rules as follows:

```
constructor(fb: FormBuilder) {
  this.userForm = fb.group({
    email: fb.control('', [Validators.required, Validators.minLength(3)]),
    name:fb.control('', [Validators.required, Validators.minLength(3)]),
    password: fb.control('', [Validators.required, Validators.minLength(6)])
  });
}
```

As you see, all fields should be required. **email** and **name** should have at least 3 characters. The **password** field should have at least 6 characters.

The **code-driven** form also has **form states** (the **valid/invalid** things). Using the **invalid** state, we can disable the submit button if the form is not valid:

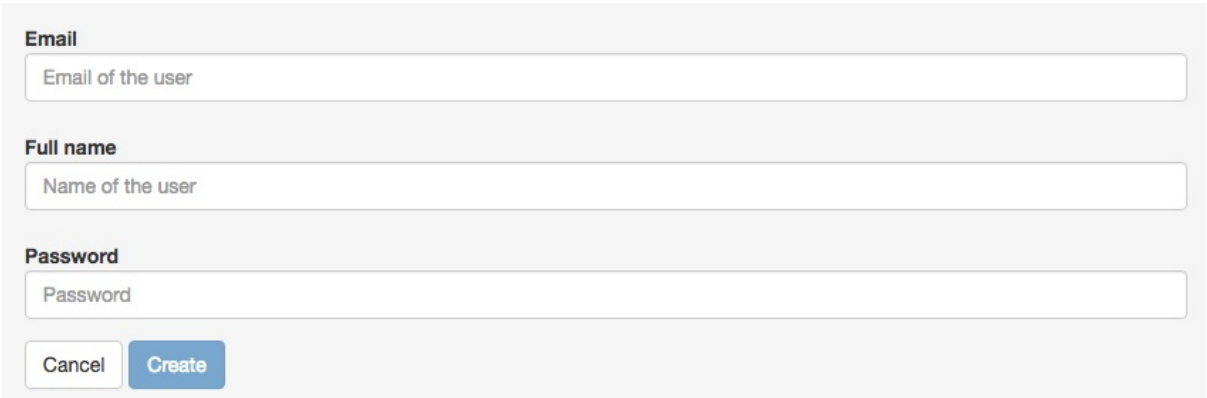
Open **admin-user-create.component.html**, and find:

```
<button type="submit" class="btn btn-primary">Create</button>
```

Modify to:

```
<button type="submit" class="btn btn-primary" [disabled]="userForm.invalid">Create</bu
tton>
```

Check the form in your browser, the **Create** button is greyed out if the form is invalid.



The great thing is, if you don't want to use **Validator**, you can still use the **required** attribute and other attributes in the **code-driven** form, too. For example:

```
<input type="text" class="form-control" id="email" name="email" placeholder="Email of
the user" formControlName="email" required>
```

So, how can we display errors in the **code-driven** form?

A Validator will return a map of errors if a control is not valid. We can display error messages using **ngIf** like this:

```
<div *ngIf="userForm.get('email').dirty && userForm.get('email').hasError('required')"
class="alert alert-danger">Email is required</div>
<div *ngIf="userForm.get('email').dirty && userForm.get('email').hasError('minlength')"
 class="alert alert-danger">Email should have at least 3 characters</div>
```

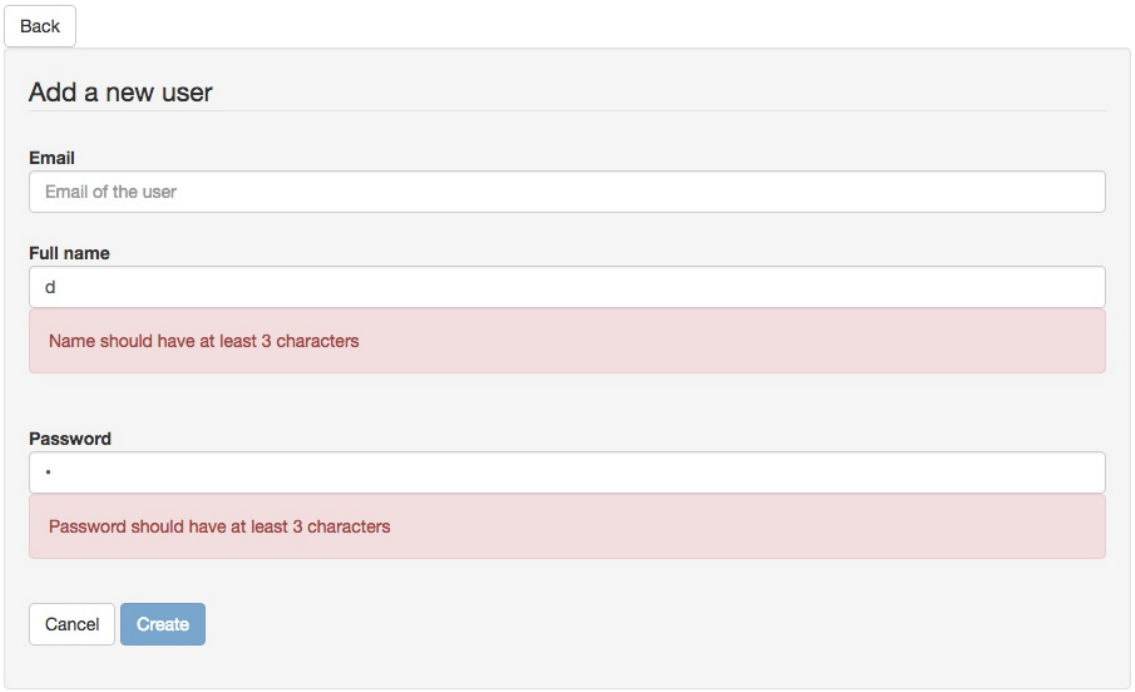
Basically, the above code means: "Ok, if this form is dirty and if it has an error, display an error message".

Here is the updated **Admin User Create** form:

```
<div class="col-md-10 col-md-offset-1">
  <div>
    <a [routerLink]="['/admin/']" class="btn btn-default"> Back</a>
  </div>
</div>

<div class="col-md-10 col-md-offset-1">
  <div class="well well bs-component">
    <form novalidate class="form-horizontal" (ngSubmit)="createUser()" [formGroup]="us
erForm">
      <fieldset>
        <legend>Add a new user</legend>
        <div class="form-group">
          <label for="email" class="col-lg-2 control-label">Email</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="email" name="email" placeholder
="Email of the user" formControlName="email">
            <div *ngIf="userForm.get('email').dirty && userForm.get('email').hasError('
required')" class="alert alert-danger">Email is required</div>
            <div *ngIf="userForm.get('email').dirty && userForm.get('email').hasError('
minlength')" class="alert alert-danger">Email should have at least 3 characters</div>
          </div>
        </div>
        <div class="form-group">
          <label for="name" class="col-lg-2 control-label">Full name</label>
```

```
          <div class="col-lg-10">
            <input type="text" class="form-control" id="name" name="name" placeholder="
Name of the user" formControlName="name">
            <div *ngIf="userForm.get('name').dirty && userForm.get('name').hasError('r
equired')" class="alert alert-danger">Name is required</div>
            <div *ngIf="userForm.get('name').dirty && userForm.get('name').hasError('m
inlength')" class="alert alert-danger">Name should have at least 3 characters</div>
          </div>
        </div>
        <div class="form-group">
          <label for="password" class="col-lg-2 control-label">Password</label>
          <div class="col-lg-10">
            <input type="password" class="form-control" id="password" name="password"
placeholder="Password" formControlName="password">
            <div *ngIf="userForm.get('password').dirty && userForm.get('password').has
Error('required')" class="alert alert-danger">Password is required</div>
            <div *ngIf="userForm.get('password').dirty && userForm.get('password').has
Error('minlength')" class="alert alert-danger">Password should have at least 6 charact
ers</div>
          </div>
        </div>

        <div class="form-group">
          <div class="col-lg-10 col-lg-offset-2">
            <a [routerLink]="['/admin/']" class="btn btn-default"> Cancel</a>
            <button type="submit" class="btn btn-primary" [disabled]="userForm.invalid"
>Create</button>
          </div>
        </div>
      </fieldset>
    </form>
  </div>
</div>
```

Check the form again, you should see some error messages!



Custom validation

With **code-driven** approach, we can create extra validation rules to validate our controls.

Let's say we want to create a new **email** rule to validate the email field, we can do it by following these steps:

First, we have to import **FormControl** into our component:

```
import {FormBuilder, FormGroup, Validators, FormControl} from "@angular/forms";
```

Open **admin-user-create.component.ts**, and find:

```
constructor(fb: FormBuilder) {
```

Add above:

```
static isValidEmail(control: FormControl) {
  let email_regexp = /^[a-z0-9!#$%&'*+\/=?^_`{|}~.-]+@[a-z0-9]([a-z0-9-]*[a-z0-9])?(\.
[a-z0-9]([a-z0-9-]*[a-z0-9])?)*$/i;

  return email_regexp.test(control.value) ? null : {
        invalidEmail: true
```

```
    };
}
```

As you can see, we create a new static method called **isValidEmail**. After that, we will use a regular expression to check if the control value is a valid email:

```
let email_regexp = /^[a-z0-9!#$%&'*+\/=?^_`{|}~.-]+@[a-z0-9]([a-z0-9-]*[a-z0-9])?(\.[a
-z0-9]([a-z0-9-]*[a-z0-9])?)*$/i;
```

To check the value, we can use the **email_regexp.test()** method. The method returns **null** if the control value is a valid email, and returns **{ invalidEmail : true }** if the value is not a valid email.

```
return email_regexp.test(control.value) ? null : {
      invalidEmail: true
  };
```

Next, we can apply the rule to the email control like this:

```
email: fb.control('', [Validators.required, Validators.minLength(3), AdminUserCreateCo
mponent.isValidEmail]),
```

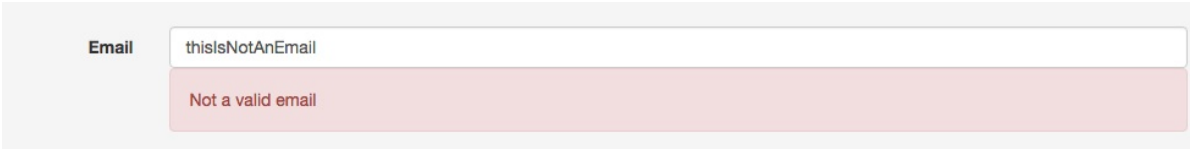
Finally, we can display an error if the email field is not valid by opening **admin-user-create.component.html** and find:

```
<input type="text" class="form-control" id="email" name="email" placeholder="Email of
the user" formControlName="email">
```

Add below:

```
<div *ngIf="userForm.get('email').dirty && userForm.get('email').hasError('invalidEmail
')" class="alert alert-danger">Not a valid email</div>
```

Done! Try to enter an invalid email, you should see an error:



There are many ways to create custom validators, but this is the simplest way. If you want, you can create a custom validator in another class, and then import it into your component when needed.

By the way, there is a good package called ng2-validators that we can use to add more validators to our Angular app. Here is a list of validators:

- Password
- Email
- Universal
- Creditcards

Alternatively, you can use Ng2 Validation Manager, which is based on Laravel Validation method.

Be sure to check them out!

Creating a new user using Laravel

To create a new user, we can use the **Auth/RegisterController**. However, we will create a new controller called **UsersController** to manage users easily, .

Go to the root of your Laravel app, and run this command to generate a new controller:

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

Next, open **api.php** and add this **users** route:

```
Route::group(['prefix' => 'v1', 'middleware' => 'cors'], function(){
    Route::resource('images', 'ImagesController');
    Route::resource('users', 'UsersController');
});
```

To add a new user to our database, we just need to update the **store** action as follows:

```
public function store(Request $request)
{
    if ((!$request->email) || (!$request->name) || (!$request->password)) {
```

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

        $user = new User(array(
            'email' => trim($request->email),
            'name' => trim($request->name),
            'password' => bcrypt($request->password),
        ));
        $user->save();

        $message = 'The user has been created successfully';

        $response = Response::json([
            'message' => $message,
            'data' => $user,
        ], 201);

        return $response;
    }
```

Don't forget to tell Laravel that we're using **User** and **Response** here:

Find:

```
class UsersController extends Controller
```

Add above:

```
use App\User;
use Illuminate\Support\Facades\Response;
```

Done! When we send a POST request to http://angularbook.app/api/v1/users, a new user will be created.

Creating a new user using Angular

In order to create a new user in Angular, we will need to have a **User** model and a **User** service. We can put models and services anywhere in our application; they work just fine. However, I'd like to put all models in a **models** directory, and all services in a **services** directory. By doing this way, I can easily find the files that I need faster, and we would have a better app structure.

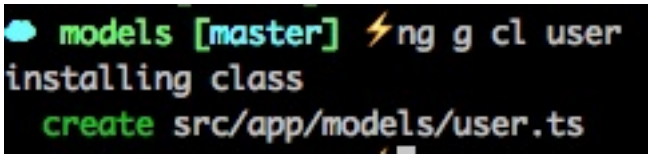
> **Note:** You're free to design app structure. There are a couple of different ways, pick the way that you love.

If you're in the **admin** directory, you can use this command to navigate back to the **app** directory and go to the **models** directory:

```
cd ../models
```

Then create the **User** model:

```
ng g cl user
```



Here is the content of the file:

user.ts

```
export class User {
  constructor(public id:string, public email:string, public name:string, public passwo
rd:string) {
  }
}
```

Similarly, we can create a new **User** service using these commands:

```
cd ../services

ng g s user
```

Once the **User** service is generated, open **user.service.ts**, and update it as follows:

user.service.ts

```
import { Injectable } from '@angular/core';
import {Http, Response, Headers, RequestOptions} from '@angular/http';
import 'rxjs/RX';
import {Observable} from 'rxjs/Observable';
import {User} from '../models/user';
```

```
@Injectable()
export class UserService {

  constructor(private http:Http) {
  }

  addUser(user: Object): Observable<User[]> {
    return this.http.post('http://angularbook.app/api/v1/users', user)
      .map((response: Response) => response.json())
      .catch((error:any) => Observable.throw(error.json().error || {message:"Server Er
ror"}));
  }
}
```

Just like what we've done with the **Image** service, we import everything that we need into our service first:

```
import { Injectable } from '@angular/core';
import {Http, Response, Headers, RequestOptions} from '@angular/http';
import 'rxjs/RX';
import {Observable} from 'rxjs/Observable';
import {User} from '../models/user';
```

After that we create a new **addUser** method to send a **POST** request to our backend:

```
addUser(user: Object): Observable<User[]> {
  return this.http.post('http://angularbook.app/api/v1/users', user)
    .map((response: Response) => response.json())
    .catch((error:any) => Observable.throw(error.json().error || {message:"Server Erro
r"}));
}
```

Next, open **app.module.ts**, then import and add the **UserService** into the **providers** array:

```
import {UserService} from "./services/user.service";

providers: [ImageService, UserService],
```

Finally, open **admin-user-create.component.ts**, and find:

```
createUser() {
  console.log(this.userForm.value);
}
```

Update it:

```
createUser() {
  this.userService.addUser(this.userForm.value)
    .subscribe(
      user  => {
        console.log(user);
        this.router.navigate(['/admin']);
      },
      error =>  console.log(<any>error));
}
```

We use the **User** service here to submit the form.

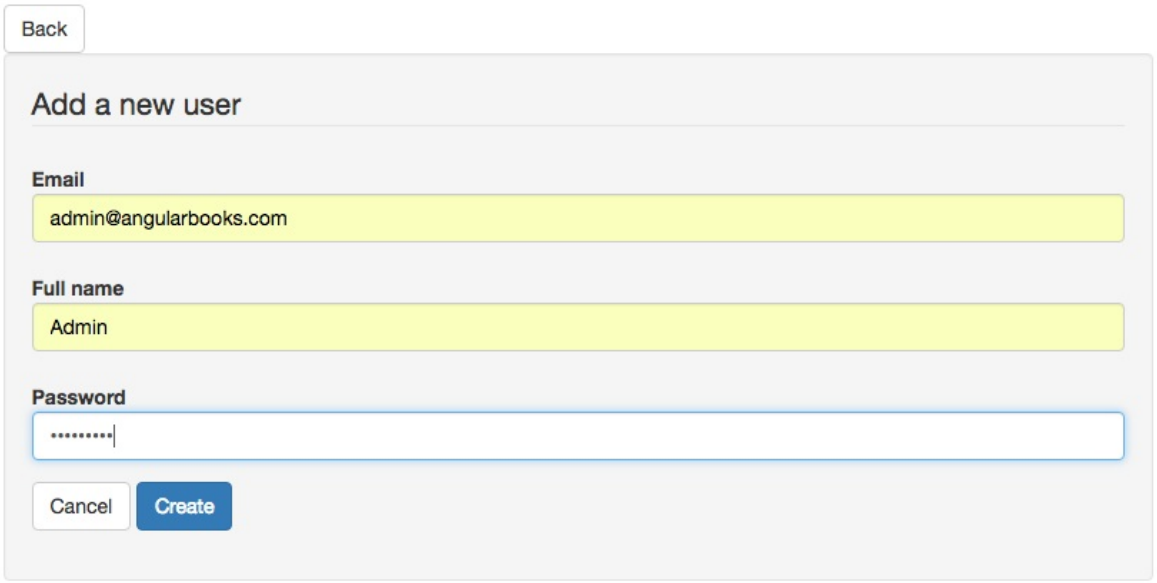
Don't forget to import **Router** and **UserService**:

```
import {UserService} from '../../services/user.service';
import {Router} from '@angular/router';
```

And inject them into our constructor:

```
constructor(fb: FormBuilder, private userService: UserService, private router: Router)
{
```

Visit your browser, and try to create a new user.



We should see this object in our Console tab:

Well done! a new user has been created!

List all users

In this section, we'll create a place to view all users in our app.

> **Tip:** These steps are very similar to what we have done with **Admin Image List** component, so try to do this by yourself to test your knowledge.

List all users using Laravel

Open **UsersController**, and update the **index** action as follows:

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

Now when we send a **GET** request to http://angularbook.app/api/v1/users, we should get all users.

List all users using Angular

As before, we'll create a new component first.

Be sure that you're in the **admin** directory:

```
cd ../admin
```

Then run these commands to generate new components:

```
ng g c admin-user-list
```

```
admin [master] ⚡ng g c admin-user-list
installing component
  create src/app/admin/admin-user-list/admin-user-list.component.css
  create src/app/admin/admin-user-list/admin-user-list.component.html
  create src/app/admin/admin-user-list/admin-user-list.component.spec.ts
  create src/app/admin/admin-user-list/admin-user-list.component.ts
  update src/app/app.module.ts
```

Here is the content of all the files:

admin-user-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import {Observable} from 'rxjs/Observable';
import {User} from '../../models/user';
import {UserService} from '../../services/user.service';

@Component({
  selector: 'ng-admin-user-list',
  templateUrl: './admin-user-list.component.html',
  styleUrls: ['./admin-user-list.component.css']
})
export class AdminUserListComponent implements OnInit {
  users: Observable<User[]>;

  constructor(private userService: UserService) { }

  ngOnInit() {
    this.users = this.userService.getUsers();
  }

}
```

user.service.ts

```
import { Injectable } from '@angular/core';
import {Http, Response, Headers, RequestOptions} from '@angular/http';
import 'rxjs/RX';
import {Observable} from 'rxjs/Observable';
import {User} from '../models/user';

@Injectable()
export class UserService {

  constructor(private http:Http) {
  }
```

```
  addUser(user: Object): Observable<User[]> {
    return this.http.post('http://angularbook.app/api/v1/users', user)
      .map((response: Response) => response.json())
      .catch((error:any) => Observable.throw(error.json().error || {message:"Server Er
ror"}));
  }

  getUsers(): Observable<User[]> {
    return this.http.get('http://angularbook.app/api/v1/users')
      .map((response: Response) => response.json())
      .catch((error:any) => Observable.throw(error.json().error || {message:"Server Er
ror"}));
  }
}
```

admin-user-list.component.html

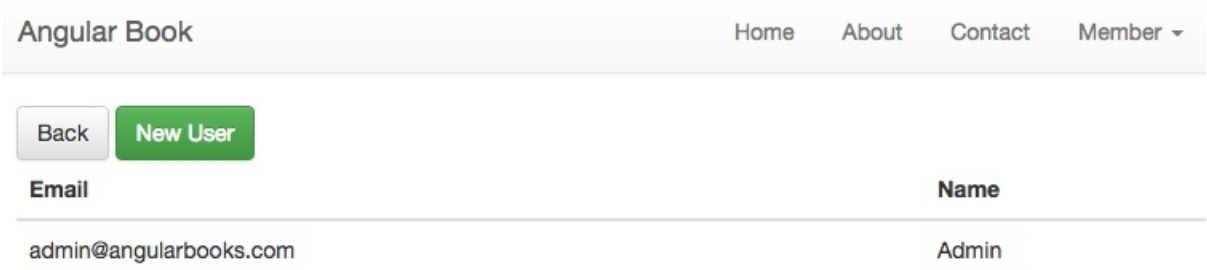
```
<div class="col-md-10 col-md-offset-1">
  <div>
    <a [routerLink]="['/admin/']" class="btn btn-default"> Back</a>
    <a [routerLink]="['/admin/users/create']" class="btn btn-success"> New User</a>
  </div>
</div>

<div class="col-md-10 col-md-offset-1">

  <div class="table-responsive">
    <table class="table">
      <thead>
      <tr>
        <th>Email</th>
        <th>Name</th>
      </tr>
      </thead>
      <tbody>

      <tr *ngFor="let user of users| async">
        <td>{{user.email}} </td>
        <td>{{user.name}}</td>
      </tr>

      </tbody>
    </table>
  </div>
</div>
```

Finally, open **admin.routes.ts**, add a new route:

admin.routes.ts

```
import {Routes} from '@angular/router';
import {AdminImageListComponent} from './admin-image-list/admin-image-list.component';
import {DashboardComponent} from './dashboard.component';
import {AdminImageCreateComponent} from './admin-image-create/admin-image-create.compo
nent';
import {AdminImageEditComponent} from './admin-image-edit/admin-image-edit.component';
import {AdminImageDeleteComponent} from './admin-image-delete/admin-image-delete.compo
nent';
import {AdminUserCreateComponent} from './admin-user-create/admin-user-create.componen
t';
import {AdminUserListComponent} from './admin-user-list/admin-user-list.component';

export const adminRoutes: Routes = [
  { path: '', component: DashboardComponent},
  { path: 'images', component: AdminImageListComponent},
  { path: 'images/create', component: AdminImageCreateComponent},
  { path: 'images/edit/:id', component: AdminImageEditComponent },
  { path: 'images/delete/:id', component: AdminImageDeleteComponent },
  { path: 'users/create', component: AdminUserCreateComponent},
  { path: 'users', component: AdminUserListComponent},
];
```

Visit http://localhost:4200/admin/users, you should see all users.



Angular Authentication

Authentication is one of the most common tasks of an application. We always want to only allow certain types of users to access certain areas. For example, we're building an admin control panel here, so only admins can access the admin area, right?

In this section, we'll learn how to build a registration page and a login page. After that, we'll learn how to restrict access to the admin area.

Register a new user in Laravel

To start off, we will have to build an API that will handle a registration request, sent from our Angular app.

I'd like to put all actions in a new controller called **AuthController**:

> **Note:** You may use **UsersController** or any controller that you want.

vagrant ssh to your **Homestead**, go to your app directory (**angularbook**), and use **PHP Artisan** to create it:

```
php artisan make:controller AuthController
```

Next, open **api.php** and add a new route called **register**:

```
Route::group(['prefix' => 'v1', 'middleware' => 'cors'], function(){
    Route::resource('images', 'ImagesController');
    Route::resource('users', 'UsersController');
    Route::post('register', 'AuthController@postRegister');
});
```

Open **AuthController**, and add a new action called **postRegister**:

```
public function postRegister(Request $request) {

    $validator = Validator::make($request->all(), [
        'email' => 'required|email|unique:users,email',
        'name' => 'required|min:2',
        'passwordForm.password' => 'required|alphaNum|min:6|same:passwordForm.password
_confirmation',
    ]);

    if ($validator->fails()) {
        $message = ['errors' => $validator->messages()->all()];
        $response = Response::json($message,202);
    } else {

        $user = new User(array(
            'email' => trim($request->email),
            'name' => trim($request->name),
            'password' => bcrypt($request->passwordForm['password']),
        ));

        $user->save();
```

```
            $message = 'The user has been created successfully';

            $response = Response::json([
                'message' => $message,
                'user' => $user,
            ], 201);
        }
        return $response;
    }
```

We're using **User**, **Validator** and **Response** here, so let's import them. Find:

```
class AuthController extends Controller
```

Add above:

```
use Response;
use App\User;
use Illuminate\Support\Facades\Validator;
```

You may notice that we've just created some **validation rules**. If you're not familiar with these rules, please take a look at the Laravel documentation or read the Learning Laravel 5 book.

If the **validation fails**, an error response will be generated to notify users. If the **form is valid**, a successful response will be generated, and a new user will be created.

> **Note:** We use **passwordForm.password** here, not just **password**. Because we will submit a **password form** using Angular.

Register a new user in Angular

As you know, we'll need to create a new component called **Register** component, which is very similar to the **Admin Create User** component.

While you're in the **app** directory, run this command to generate the component:

```
ng g c register
```

Open **app.routes.ts**, add a new route:

```
const appRoutes: Routes = [
  { path: '', redirectTo:'/gallery', pathMatch: 'full'},
  { path: 'gallery', component: GalleryComponent},
  { path: 'contact', component: ContactComponent},
  { path: 'about', component: AboutComponent},
  { path: 'admin', component: AdminComponent, children: adminRoutes},
  { path: 'register', component: RegisterComponent},
];
```

And import the **RegisterComponent**:

```
import {RegisterComponent} from '../register/register.component';
```

Because this form will be entered by the normal users, we should add an extra input password field called **password_confirmation** to ensure that the password is correct. Additionally, we'll add a new custom validation rule called **samePassword** to validate the fields.

Here is the **register.component.ts**:

register.component.ts

```
import { Component, OnInit } from '@angular/core';
import {FormGroup, FormControl, FormBuilder, Validators} from '@angular/forms';
import {Router} from '@angular/router';
import {AuthService} from '../services/auth.service';

@Component({
  selector: 'ng-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.css']
})
export class RegisterComponent implements OnInit {
  userForm: FormGroup;
  passwordForm: FormGroup;
  passwordControl: FormControl;
  confirmationControl: FormControl;
  nameControl: FormControl;
  emailControl: FormControl;

  static isValidEmail(control: FormControl) {
    const email_regexp = /^[a-z0-9!#$%&'*+\/=?^_`{|}~.-]+@[a-z0-9]([a-z0-9-]*[a-z0-9])
?(\.[a-z0-9]([a-z0-9-]*[a-z0-9])?)*$/i;

    return email_regexp.test(control.value) ? null : {
      invalidEmail: true
    };
```

```
  }

  static samePassword(group: FormGroup) {
    const password = group.get('password').value;
    const password_confirmation = group.get('password_confirmation').value;
    return password === password_confirmation ? null : {
      invalidPassword: true
    };
  }

  constructor(fb: FormBuilder, private authService: AuthService, private router: Router
) {
    this.passwordControl = fb.control('', [Validators.required, Validators.minLength(6)
]);
    this.confirmationControl = fb.control('', [Validators.required, Validators.minLeng
th(6)]);
    this.nameControl = fb.control('', [Validators.required, Validators.minLength(3)]);
    this.emailControl = fb.control('', [Validators.required, RegisterComponent.isValid
Email]);

    this.passwordForm = fb.group ({
        password: this.passwordControl,
        password_confirmation: this.confirmationControl
      }, {validator: RegisterComponent.samePassword }
    );

    this.userForm = fb.group({
      email: this.emailControl,
      name: this.nameControl,
      passwordForm: this.passwordForm
    });
  }

  ngOnInit() {
  }

  registerUser() {
    this.authService.register(this.userForm.value)
      .subscribe(
        response  => {
          console.log(response);
          this.router.navigate(['/']);
        },
        error =>  console.log(<any>error));
  }

}
```

Well, this component is a bit different, but nothing fancy here. First, we import everything that we need:

```
import { Component, OnInit } from '@angular/core';
import {FormGroup, FormControl, FormBuilder, Validators} from '@angular/forms';
import {Router} from '@angular/router';
import {AuthService} from '../services/auth.service';
```

After that, we create some new objects:

```
userForm: FormGroup;
passwordForm: FormGroup;
passwordControl: FormControl;
confirmationControl: FormControl;
nameControl: FormControl;
emailControl: FormControl;
```

Instead of writing this way:

```
this.userForm = fb.group({
  email: fb.control('', [Validators.required, Validators.minLength(3), RegisterCompone
nt.isValidEmail]),
  name:fb.control('', [Validators.required, Validators.minLength(3)]),
});
```

We can write like this:

```
this.emailControl = fb.control('', [Validators.required, RegisterComponent.isValidEmail
]);
this.nameControl = fb.control('', [Validators.required, Validators.minLength(3)]);

this.userForm = fb.group({
  email: this.emailControl,
  name: this.nameControl,
});
```

We group the **password** and **password_confirmation** control into a group called **passwordForm**. Then we apply a validation rule called **samePassword** to the passwordForm:

```
this.passwordForm = fb.group ({
    password: this.passwordControl,
    password_confirmation: this.confirmationControl
  }, {validator: RegisterComponent.samePassword }
);
```

Here is the **samepassword** custom validation rule:

```
static samePassword(group: FormGroup) {
  const password = group.get('password').value;
  const password_confirmation = group.get('password_confirmation').value;
  return password === password_confirmation ? null : {
      invalidPassword: true
    };
}
```

Next, we put all controls into the **userForm**:

```
this.userForm = fb.group({
  email: this.emailControl,
  name: this.nameControl,
  passwordForm: this.passwordForm
});
```

Finally, we create a new method called **registerUser**. When users submit the form, we will use the **authService** to send a request to our backend.

Here is our **register.component.html** file:

register.component.html

```
<div class="col-md-10 col-md-offset-1">
  <div class="well well bs-component">
    <form novalidate class="form-horizontal" (ngSubmit)="registerUser()" [formGroup]="
userForm">
      <fieldset>
        <legend>Register</legend>
        <div class="form-group">
          <label for="email" class="col-lg-2 control-label">Email</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="email" name="email" placeholder
="Email of the user" formControlName="email">
            <div *ngIf="emailControl.dirty && emailControl.hasError('required')" class=
"alert alert-danger">Email is required</div>
            <div *ngIf="emailControl.dirty && emailControl.hasError('minlength')" class
="alert alert-danger">Email should have at least 3 characters</div>
            <div *ngIf="emailControl.dirty && emailControl.hasError('invalidEmail')" c
lass="alert alert-danger">Not a valid email</div>

          </div>
        </div>
        <div class="form-group">
          <label for="name" class="col-lg-2 control-label">Full name</label>
          <div class="col-lg-10">
```

```
            <input type="text" class="form-control" id="name" name="name" placeholder="
Name of the user" formControlName="name">
            <div *ngIf="nameControl.dirty && nameControl.hasError('required')" class="
alert alert-danger">Name is required</div>
            <div *ngIf="nameControl.dirty && nameControl.hasError('minlength')" class="
alert alert-danger">Name should have at least 3 characters</div>
          </div>
        </div>
        <div formGroupName="passwordForm">

        <div class="form-group">
          <label for="password" class="col-lg-2 control-label">Password</label>
          <div class="col-lg-10">
            <input type="password" class="form-control" id="password" name="password"
placeholder="Password" formControlName="password">
            <div *ngIf="passwordControl.dirty && passwordControl.hasError('required')"
class="alert alert-danger">Password is required</div>
            <div *ngIf="passwordControl.dirty && passwordControl.hasError('minlength')"
 class="alert alert-danger">Password should have at least 6 characters</div>
          </div>
        </div>
        <div class="form-group">
          <label for="password" class="col-lg-2 control-label">Password Confirmation</
label>
          <div class="col-lg-10">
            <input type="password" class="form-control" id="password_confirmation" name
="password_confirmation" placeholder="Password Confirmation" formControlName="password
_confirmation">
            <div *ngIf="confirmationControl.dirty && confirmationControl.hasError('req
uired')" class="alert alert-danger">Password Confirmation is required</div>
            <div *ngIf="confirmationControl.dirty && confirmationControl.hasError('min
length')" class="alert alert-danger">Password Confirmation should have at least 6 char
acters</div>
            <div *ngIf="confirmationControl.dirty && passwordForm.hasError('invalidPas
sword')" class="alert alert-danger">The passwords you enter must match</div>
          </div>
        </div>
        </div>
        <div class="form-group">
          <div class="col-lg-10 col-lg-offset-2">
            <a [routerLink]="['/']" class="btn btn-default"> Cancel</a>
            <button type="submit" class="btn btn-primary" [disabled]="userForm.invalid"
>Register</button>
          </div>
        </div>
      </fieldset>
    </form>
  </div>
</div>
```

As you see, by using **FormControl** objects, we can access the controls directly:

```
<div *ngIf="emailControl.dirty && emailControl.hasError('required')" class="alert aler
t-danger">Email is required</div>
```

Instead of:

```
<div *ngIf="userForm.get('email').dirty && userForm.get('email').hasError('required')"
class="alert alert-danger">Email is required</div>
```

Much cleaner, right?

One important thing to note, we must put password fields into the **passwordForm** div:

```
<div formGroupName="passwordForm">
    (...password fields here)
</div>
```

Next, go to the **services** directory, and create a new **Auth Service**:

```
cd services

ng g s auth
```

Open **app.module.ts**, then import and add the **AuthService** into the **providers** array:

```
import {AuthService} from './services/auth.service';

providers: [ImageService, UserService, AuthService],
```

Last step, open **auth.service.ts**, and add this **register** method:

```
register (user: Object): Observable<any> {
  return this.http.post('http://angularbook.app/api/v1/register', user)
    .map((response: Response) => response.json())
    .catch((error:any) => Observable.throw(error.json().error || {message:"Server Erro
r"}));
}
```

Import **Observable**, **Http**, and **Response** into the service:

```
import {Observable} from 'rxjs/Observable';
import {Http, Response } from '@angular/http';
```

Then inject **Http** into the constructor:

```
constructor(private http:Http) {}
```

> **Tip:** you can use the **User Service** or any service that you like, instead of using the **Auth Service**.

Now, visit http://localhost:4200/register, you should see a nice registration form.

Try to register a new user! If your passwords don't match, you'll see an error.

If everything is ok, you should see this in your **Console** tab.

```
▶ Object {message: "The user has been created successfully", data: Object}    register.component.ts:63
```

Good job! You've successfully registered a new user!

Displaying multiple errors

Our registration form is not perfect yet. It doesn't display errors from the server when the **Laravel Validator** fails. Because of that, if we try to register a new account using a taken email, a new user will not be created, but we won't see any error.

To fix this bug, open **register.component.ts**, and add some new variables:

```
status: string;
message: string;
errors: any;
```

The **errors** variable will contain one or multiple errors, because we return an **array of errors** in Laravel:

AuthController.php (Laravel file)

```
if ($validator->fails()) {
    $message = ['errors' => $validator->messages()->all()];
    $response = Response::json($message,202);
}
```

Once we have all the variables, we may update the **registerUser()** method as follows:

register.component.ts

```
registerUser() {
  this.authService.register(this.userForm.value)
    .subscribe(
      response  => {

        const user = response.user;

        if(user) {
          console.log("A new user has been created");
          this.status = "success";
          this.message = response['message'];
          this.router.navigate(['/admin']);
        } else {
          this.errors = response.errors;
        }
      },
      error =>  console.log(<any>error));
}
```

As you see, if the response has user information, we'll display a success message. Otherwise, we simply **assign the response's errors** to the **errors** variable that we've just created.

To display multiple errors, we'll use **ngFor**:

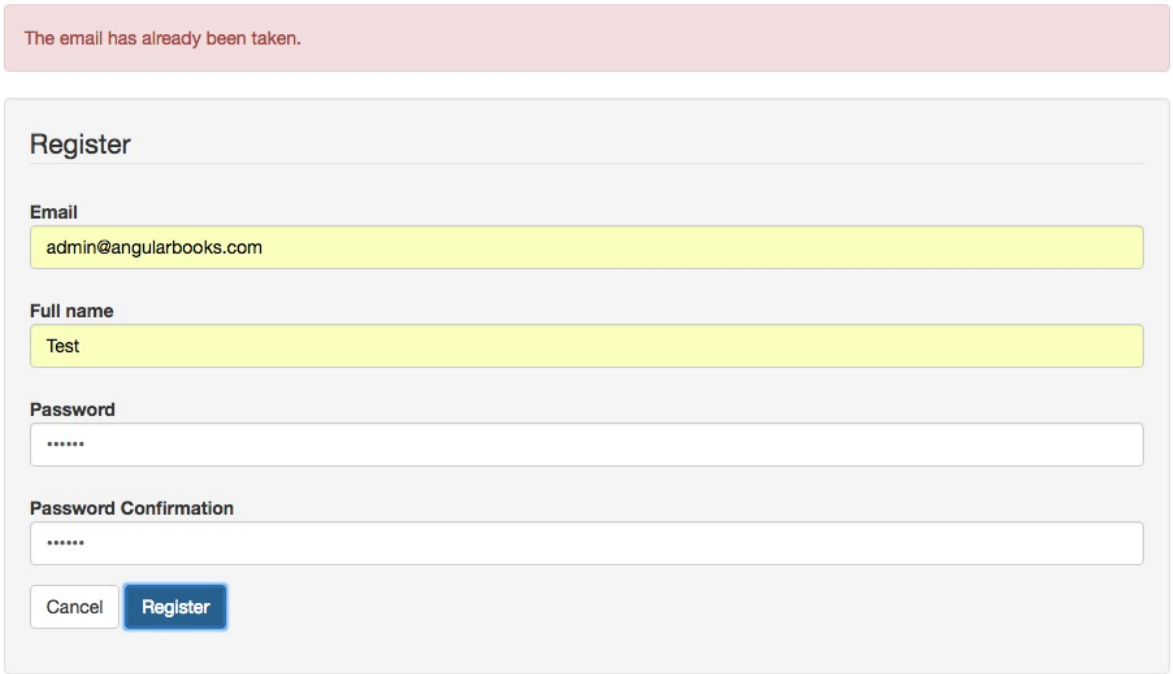
Open **register.component.html**, and find:

```
<div class="well well bs-component">
```

Add above:

```
<div *ngIf="status=='success'" class="alert alert-success" role="alert"> {{ message }}
</div>
<div *ngFor="let error of errors">
  <div class="alert alert-danger" role="alert"> {{ error }} </div>
</div>
```

Now if you try to register a new account using a registered email, you should see an error:



> **Note:** Because we only have one error here, we only see one. If the server returns more errors, we'll see them all.

That's how we display multiple errors!

Creating a login API using Laravel

In this section, we'll build a simple login API endpoint that we will use to retrieve a **JSON Web Token** (**JWT**).

What is **JWT**? **JWT** is an **authentication protocol**, we use it to securely transfer information in JSON format. It's very similar to session or cookie, but it's stateless; which means we don't need to store any session on the server. This kind of authentication is also known as **Token-Based Authentication**.

Using **Token-Based Authentication**, we will have to send a request, which contains user information, to an authentication server. The server will verify the request. If the user information is correct, the server will generate a new **JWT** token, and send it back to us. Once we have the token, we can use it to fetch a secure resource from anywhere.

> **Tip:** Since Laravel 5.3, we can also use Passport library, which is built on top of the **League OAuth2** server, to generate a token. However, OAuth 2 is a framework, and it may take some time to learn. JWT is much easier to learn.

To use **JWT** with Laravel, we first have to install jwt-auth package.

Open **composer.json** and add the following to the require array:

```
"require": {
    "tymon/jwt-auth": "0.5.*"
}
```

After that, **vagrant ssh** into your Homestead, go to the root of your Laravel application and run:

```
composer update
```

Once finished, open **app.php**, and add the following to the **provider** array:

```
'Tymon\JWTAuth\Providers\JWTAuthServiceProvider'
```

Next, find the **aliases** array, and add these facades:

```
'JWTAuth' => 'Tymon\JWTAuth\Facades\JWTAuth'
'JWTFactory' => 'Tymon\JWTAuth\Facades\JWTFactory'
```

Last step, you may publish the config file of this package by running this command:

```
php artisan vendor:publish --provider="Tymon\JWTAuth\Providers\JWTAuthServiceProvider"
```

A new file called **jwt.php** will be generated in the **config** directory. Here are some options that you can configure. You may set the secret key in the file, or you may use this command to generate a new key:

```
php artisan jwt:generate
```

The **jwt-auth** package is now installed!

You may read more information and learn how to install the package here.

Now, let's open **api.php** and create a new route called login:

```
Route::group(['prefix' => 'v1', 'middleware' => 'cors'], function(){
    Route::resource('images', 'ImagesController');
    Route::resource('users', 'UsersController');
    Route::post('register', 'AuthController@postRegister');
    Route::post('login', 'AuthController@authenticate');
});
```

As you see, this route is handled by the **AuthController**'s **authenticate** action:

AuthController.php

```
public function authenticate(Request $request)
{
    $validator = Validator::make($request->all(), [
        'email' => 'required|email',
        'password' => 'required',
    ]);

    if ($validator->fails()) {
        $message = ['errors' => $validator->messages()->all()];
        $response = Response::json($message, 202);
    } else {
```

```
        $credentials = $request->only('email', 'password');

        try {
            $token = JWTAuth::attempt($credentials);

            if ($token) {
                $message = ['success' => $token];
                return $response = Response::json(["token" => $token], 200);
            } else {
                $message = ['errors' => "Invalid credentials"];
                return $response = Response::json($message, 202);
            }
        } catch (JWTException $e) {
            return response()->json(['error' => 'could_not_create_token'], 500);
        }
    }
}
```

Here we also use the **Validator** to check the request. The **JWTAuth::attempt** method is used to **attempt to verify** the user. If the credentials are correct, a new token will be generated.

We can now check the token. If the token is not empty (that means a new key has been created), we will send a **success response with the token** back. Otherwise, we will return an error.

```
if ($token) {
    $message = ['success' => $token];
    return $response = Response::json(["token" => $token], 200);
} else {
    $message = ['errors' => "Invalid credentials"];
    return $response = Response::json($message, 202);
}
```

We also use **try and catch** to catch possible **JWTException** errors.

```
    } catch (JWTException $e) {
        return response()->json(['error' => 'could_not_create_token'], 500);
    }
```

> **Note:** You don't have to use JWTException here. it's optional.

Don't forget to import **JWTAuth** and **JWTException**:

Find:

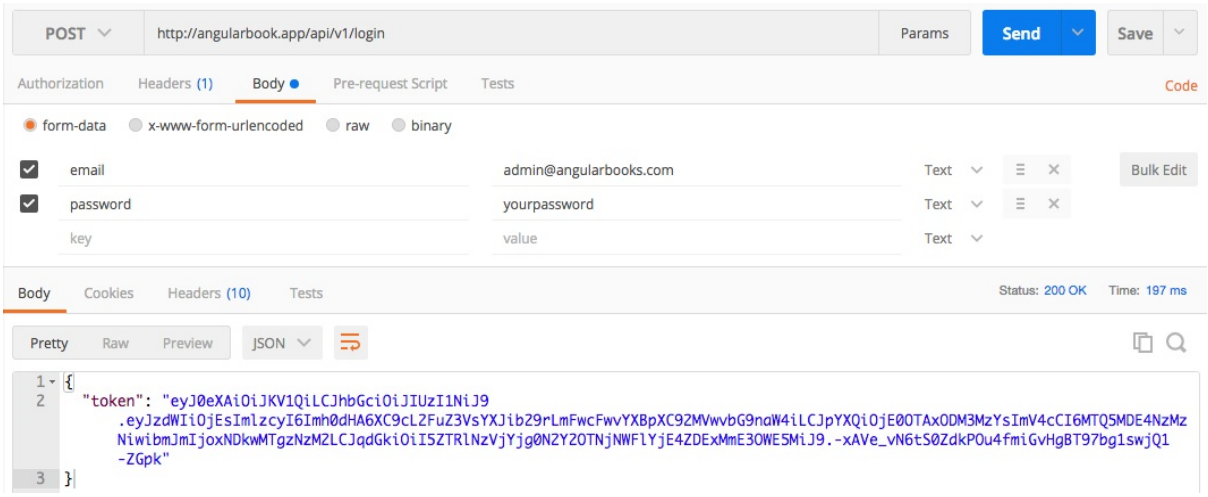
```
class AuthController extends Controller
```

Add above:

```
use Tymon\JWTAuth\Facades\JWTAuth;
use Tymon\JWTAuth\Exceptions\JWTException;
```

Done! Now if we send a request that has correct credentials to http://angularbook.app/api/v1/login, you will receive a token!

You may use **Postman** to test the new API endpoint.



> **Tip:** This technique can be used with Vue.js, ReactJS, or any other frameworks and applications that consume API.

Getting a JSON Web token using Angular

In this section, we're going to build a login form, which is used to send a user's credentials to the Laravel API endpoint and retrieve a JSON Web Token.

We start by generating a new **Login** component. Go to the **app** directory, and run this command:

```
ng g c login
```

A new **LoginComponent** will be created!

Open **app.routes.ts**, and add this new route:

```
{ path: 'login', component: LoginComponent},
```

Remember to import the **LoginComponent**:

```
import {LoginComponent} from './login/login.component';
```

Next, open **auth.service.ts**, and create a new variable called **token**, which is of type **String**:

Find:

```
export class AuthService {
```

Add below:

```
token: string;
```

Then we can create a new **login** function:

```
login (user: Object): Observable<any> {
  return this.http.post('http://angularbook.app/api/v1/login', user)
    .map((response: Response) => {
      let token = response.json().token;

      console.log("Response token:" + token);

      if (token) {
        this.token = token;
        localStorage.setItem('token', JSON.stringify(this.token));
        return true;
      } else {
        return false;
      }

    })
    .catch((error:any) => Observable.throw(error.json().error || {message:"Server Erro
r"}));
}
```

This function is used to make our HTTP request to the login API. If we get a response back, we check if the response contains the token:

```
let token = response.json().token;

console.log("Response token:" + token);

if (token) {
```

If the token exists, we use localStorage to store the token locally. Be sure that we convert the token to **JSON** using **JSON.stringify** to avoid possible errors, because we're using **JSON Web Tokens** here:

```
this.token = token;
localStorage.setItem('token', JSON.stringify(this.token));
```

After that, we return **true**, telling that the login request is successful.

```
return true;
```

If the token doesn't exist, we simply return **false**:

```
} else {
  return false;
}
```

It's time to update the **login.component.ts** to add some form controls and a new **login** method to handle the login form:

login.component.ts:

```
import { Component, OnInit } from '@angular/core';
import {FormBuilder, Validators, FormControl, FormGroup} from '@angular/forms';
import {AuthService} from '../services/auth.service';
import {Router} from '@angular/router';

@Component({
  selector: 'ng-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {
  userForm: FormGroup;
  passwordControl: FormControl;
  emailControl: FormControl;
  token: string;
```

```
  status: string;
  message: string;

  static isValidEmail(control: FormControl) {
    const email_regexp = /^[a-z0-9!#$%&'*+\/=?^_`{|}~.-]+@[a-z0-9]([a-z0-9-]*[a-z0-9])
?(\.[a-z0-9]([a-z0-9-]*[a-z0-9])?)*$/i;

    return email_regexp.test(control.value) ? null : {
      invalidEmail: true
    };
  }

  constructor(fb: FormBuilder, private authService: AuthService, private router: Router
) {
    this.passwordControl = fb.control('', [Validators.required, Validators.minLength(6)
]);
    this.emailControl = fb.control('', [Validators.required, LoginComponent.isValidEma
il]);

    this.userForm = fb.group({
      email: this.emailControl,
      password: this.passwordControl
    });
  }

  ngOnInit() {
  }

  login() {
    this.authService.login(this.userForm.value)
      .subscribe(
        response  => {
          console.log(response);
          if (response === true) {
            this.router.navigate(['/admin']);
          } else {
            this.status = "error";
            this.message = "Username or password is incorrect";
          }
        },
        error =>  {
          console.log(<any>error);
          this.status = "error";
          this.message = error['message'];
        }
      );
  }
}
```

As you see, this component is very similar to the **RegisterComponent**. The login method is also simple. We use **authService.login()** method to submit the form. If the response is **true**, we redirect users to the **admin** area:

```
if (response === true) {
  this.router.navigate(['/admin']);
```

Otherwise, we display an error message:

```
this.status = "error";
this.message = "Username or password is incorrect";
```

Here is the login view:

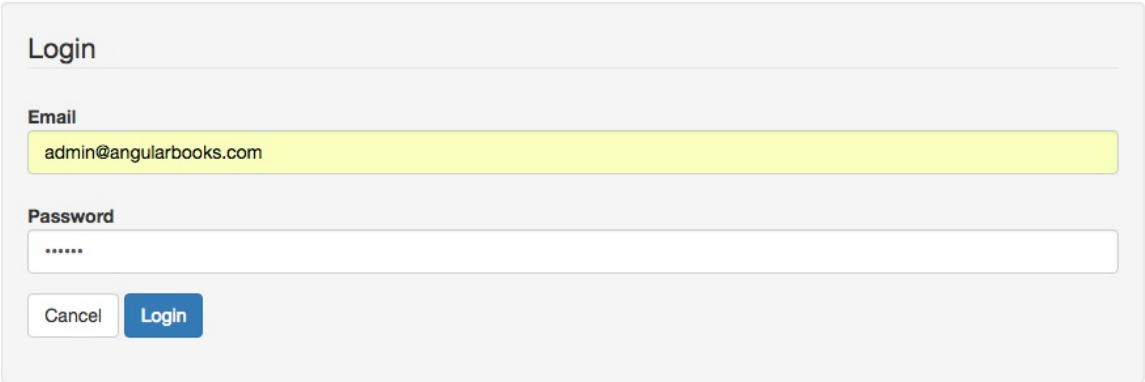
login.component.html

```
<div class="col-md-10 col-md-offset-1">

  <div *ngIf="status=='success'" class="alert alert-success" role="alert"> {{ message
}} </div>
  <div *ngIf="status=='error'" class="alert alert-danger" role="alert"> {{ message }}
</div>

  <div class="well well bs-component">
    <form novalidate class="form-horizontal" (ngSubmit)="login()" [formGroup]="userForm
">
      <fieldset>
        <legend>Login</legend>
        <div class="form-group">
          <label for="email" class="col-lg-2 control-label">Email</label>
          <div class="col-lg-10">
            <input type="text" class="form-control" id="email" name="email" placeholder
="Email of the user" formControlName="email">
            <div *ngIf="emailControl.dirty && emailControl.hasError('required')" class=
"alert alert-danger">Email is required</div>
            <div *ngIf="emailControl.dirty && emailControl.hasError('minlength')" class
="alert alert-danger">Email should have at least 3 characters</div>
            <div *ngIf="emailControl.dirty && emailControl.hasError('invalidEmail')" c
lass="alert alert-danger">Not a valid email</div>

          </div>
        </div>
          <div class="form-group">
            <label for="password" class="col-lg-2 control-label">Password</label>
            <div class="col-lg-10">
              <input type="password" class="form-control" id="password" name="password"
 placeholder="Password" formControlName="password">
```

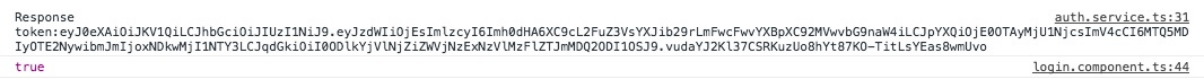
```
              <div *ngIf="passwordControl.dirty && passwordControl.hasError('required')
" class="alert alert-danger">Password is required</div>
              <div *ngIf="passwordControl.dirty && passwordControl.hasError('minlength'
)" class="alert alert-danger">Password should have at least 6 characters</div>
            </div>
          </div>
        <div class="form-group">
          <div class="col-lg-10 col-lg-offset-2">
            <a [routerLink]="['/']" class="btn btn-default"> Cancel</a>
            <button type="submit" class="btn btn-primary" [disabled]="userForm.invalid"
>Login</button>
          </div>
        </div>
      </fieldset>
    </form>
  </div>
</div>
```

Now our form should be working! Let's try it out!

Go to http://localhost:4200/login and try to login with valid credentials:



You'll be redirected to the **admin** route and see this in the **Console** tab:



If you enter a wrong password, you'll see an error message:

Installing Angular JWT package

There is a popular package called angular2-jwt, which is a helper library for working with **JWTs** in our Angular applications. Using this package, we can:

- Attach a **JWT** as an **Authorization** header when making HTTP requests.
- Decode a JWT.
- Check the expiration date of a JWT.
- Allow users access a route, based on the JWT status.

> **Note:** even though the package is called Angular 2 JWT, it still works with Angular 4.

Before going to the next section, let's install this package first!

Go to **the root** of your application (e.g. **nggallery**), and run this command to install the library:

```
npm install angular2-jwt --save
```

> **Tip:** if your application is placed at **Code/nggallery**, you can use this command to go to your app quickly: **cd ~/Code/nggallery**

If you see these errors:

```
UNMET PEER DEPENDENCY
```

Don't worry, the package will still work fine.

If you can't install the package, you may update your **package.json** file as follows:

```
{
  "name": "nggallery",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {
    "@angular/common": "^4.0.0",
    "@angular/compiler": "^4.0.0",
    "@angular/core": "^4.0.0",
    "@angular/forms": "^4.0.0",
    "@angular/http": "^4.0.0",
    "@angular/platform-browser": "^4.0.0",
    "@angular/platform-browser-dynamic": "^4.0.0",
    "@angular/router": "^4.0.0",
    "angular2-jwt": "^0.2.3",
    "core-js": "^2.4.1",
    "rxjs": "^5.1.0",
    "zone.js": "^0.8.4"
  },
  "devDependencies": {
    "@angular/cli": "1.0.0",
    "@angular/compiler-cli": "^4.0.0",
    "@types/jasmine": "2.5.38",
    "@types/node": "~6.0.60",
    "codelyzer": "~2.0.0",
    "jasmine-core": "~2.5.2",
    "jasmine-spec-reporter": "~3.2.0",
    "karma": "~1.4.1",
    "karma-chrome-launcher": "~2.0.0",
    "karma-cli": "~1.0.1",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "karma-coverage-istanbul-reporter": "^0.2.0",
    "protractor": "~5.1.0",
    "ts-node": "~2.0.0",
    "tslint": "~4.5.0",
    "typescript": "~2.2.0"
  }
```

```
}
```

After that, remove the **node_modules** directory (yourApp/node_modules), and run these commands:

```
npm cache clear
npm install --save
```

> **Note:** if you get permission errors, try to run the commands as an administrator. If you're using Mac/Linux, you may try to use: **sudo npm install --save**

The package should now be installed!

Don't forget to read the angular2-jwt's documentation to learn more about the library!

Protect our routes using route guards

In Angular, we have a feature called **Route Guards** (aka **Navigation Guards**), which we can use to protect our application routes. We have these guard types:

- **CanActivate**: A route should be activated or not.
- **CanActivateChild**: A children route should be activated or not.
- **CanDeactivate**: A route should be deactivated or not.
- **CanLoad**: telling Angular if a module can be lazily loaded.
- **Resolve:** ensure that route data is loaded before a route is activated.

You may learn more about Route Guards by reading the official documentation.

If you're working with Laravel or Node.js, you can think **Route Guards** as the **middleware**.

Interesting, right? Ok, so how do we use these guards?

Let's begin by creating a new **AuthGuard** service, which is used to check if the user can access the route or not.

Go to the **services** directory, and run this command to generate a new service:

```
ng g s auth-guard
```

As usual, remember to add this service into the **providers** array of the **app.modules.ts**:

```
providers: [ImageService, UserService, AuthService, AuthGuardService],
```

And import it as well:

```
import {AuthGuardService} from './services/auth-guard.service';
```

Here is the content of **auth-guard.service.ts**:

auth-guard.service.ts

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { CanActivate } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable()
export class AuthGuardService implements CanActivate {

  constructor(private auth: AuthService, private router: Router) {}

  canActivate() {
    if (!this.auth.loggedIn()) {
      this.router.navigate(['/login']);
      return false;
    }
    return true;
  }

}
```

Take a look at the **canActivate** function:

```
canActivate() {
  if (!this.auth.loggedIn()) {
    this.router.navigate(['/login']);
    return false;
  }
  return true;
}
```

We use the **AuthService's loggedIn** function to check if the user has logged in. If the user is not logged in, we redirect the user to the **login** page, and return **false**. Otherwise, we return **true**; which means that the route can be accessed by the user.

Next, we will have to create the **AuthService's loggedIn** function. Open **auth.service.ts**, and add:

```
loggedIn() {
  return tokenNotExpired();
}
```

As you may have guessed, the **tokenNotExpired()** method is from the **angular2-jwt** library!

Be sure to import it:

```
import {tokenNotExpired} from 'angular2-jwt';
```

This helper method is used to check whether there is a **non-expired JWT** in local storage. Simply put, if there is a token and it doesn't expire yet, the method returns **true** (logged in). If there is no token, the method returns false (not logged in).

Just a reminder, in the **login** function, when we receive a token, we add the token into the local storage using:

```
localStorage.setItem('token', JSON.stringify(this.token));
```

One more thing to note, according to **angular2-jwt**'s documentation:

> `tokenNotExpired` will by default assume the token name is `token` unless a token name is passed to it, ex: tokenNotExpired('token_name'). This will be changed in a future release to automatically use the token name that is set in **AuthConfig**.

So the default token name is `token`, and we can set the token name using the config file in the future.

> **Note:** In older versions of **angular2-jwt**, the default token name is `id_token`. Since **angular2-jwt v0.2.3**, the default token name has been changed to `token`.

Our first route guard is now ready to use!

To apply this route guard, open **app.routes.ts**, and find:

```
{ path: 'admin', component: AdminComponent, children: adminRoutes},
```

Modify to:

```
{ path: 'admin', component: AdminComponent, children: adminRoutes, canActivate: [AuthG
uardService]},
```

Finally, be sure to import the **AuthGuardService**:

```
import {AuthGuardService} from "./services/auth-guard.service";
```

Now we can't access the **admin** route if we're not logged in!

Visit http://localhost:4200/admin, and give it a try!

> **Note:** If you can still access the admin route, that means you're logged in! Go to the next section to see how to build a logout functionality.

Creating a logout functionality

We don't have the logout functionality yet, so let's implement it.

Go to the **app** directory, and run this command:

```
ng g c logout
```

Open **app.routes.ts**, and add a new route:

```
const appRoutes: Routes = [
  { path: '', redirectTo:'/gallery', pathMatch: 'full'},
  { path: 'gallery', component: GalleryComponent},
  { path: 'contact', component: ContactComponent},
  { path: 'about', component: AboutComponent},
```

```
  { path: 'admin', component: AdminComponent, children: adminRoutes, canActivate: [Aut
hGuardService]},
  { path: 'register', component: RegisterComponent},
  { path: 'login', component: LoginComponent},
  { path: 'logout', component: LogoutComponent},
];
```

Import **LogoutComponent** into the file:

```
import {LogoutComponent} from './logout/logout.component';
```

Now, of course we need to create a new **AuthService's logout** method:

auth.service.ts

```
logout(): void {
  this.token = null;
  localStorage.removeItem('token');
  console.log("you are logged out!");
  this.router.navigate(['/']);
}
```

We're using **router** here, so let's import it into the service:

```
import {Router} from '@angular/router';
```

and inject it into the constructor:

```
constructor(private http:Http, private router:Router) {}
```

As you can see, we start by setting the token to **null**, and removing the token from the local storage. After that, we redirect users to the home page.

Finally, let's modify the **logout** component. When the logout component is initialized, it should call the **AuthServer's logout** method:

```
import { Component, OnInit } from '@angular/core';
import {AuthService} from '../services/auth.service';

@Component({
  selector: 'ng-logout',
  templateUrl: './logout.component.html',
  styleUrls: ['./logout.component.css']
```

```
})
export class LogoutComponent implements OnInit {

  constructor(private authService: AuthService) {
  }

  ngOnInit() {
    this.authService.logout();
  }
}
```

Super easy! Try to test the functionality!

If you go to http://localhost:4200/logout, you will be logged out!

Protecting our APIs using JWT middleware

At the moment, we only protect the client side (our Angular app). That's a big problem! Any user who knows our APIs can access and manipulate our data. Big security risks, right?

Fortunately, we can easily avoid these risks by telling Laravel that: "Hey, I want you to **check the users' requests**. Some requests **must have a token** to access private data!".

There are many ways that we can do to achieve that, but the easiest way is to use **middlewares** that come with the **jwt-auth** package.

There are two middlewares:

- **GetUserFromToken**: check the header and query string to find the token, and decode it.
- **RefreshToken**: get a token from the request, and generate a new token, based on the old one. If you only want to use a token one time, you may use this middleware.

You may learn more about these middlewares at:

https://github.com/tymondesigns/jwt-auth/wiki/Authentication

To use these middlewares, open **app/Http/Kernel.php**, add **jwt.auth** and **jwt.refresh** to the **$routeMiddleware**:

```
protected $routeMiddleware = [
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'cors' => \App\Http\Middleware\Cors::class,
    'jwt.auth' => 'Tymon\JWTAuth\Middleware\GetUserFromToken',
    'jwt.refresh' => 'Tymon\JWTAuth\Middleware\RefreshToken',
];
```

Now if we want to protect a route, we may add the middleware into a route group:

```
Route::group(['prefix' => 'v1', 'middleware' => ['cors', 'jwt.auth']], function(){
    Route::resource('images', 'ImagesController');
    Route::resource('users', 'UsersController');
    Route::post('register', 'AuthController@postRegister');
    Route::post('login', 'AuthController@authenticate');
});
```

Or we can specify middleware within our controller's constructor.

For example, if we want to protect all **images** routes, we can open **ImagesController**, and find:

```
class ImagesController extends Controller
{
```

Add below:

```
public function __construct()
{
    $this->middleware('jwt.auth');
}
```

Visit http://angularbook.app/api/v1/images, you should see an error because you need to send a token to access this route:

```
{
    error: "token_not_provided"
}
```

Usually, we only want to protect some routes, so we may restrict the middleware to only certain methods on a controller class by using **except**:

```
public function __construct()
{
    $this->middleware('jwt.auth', ['except' => ['index', 'show']]);
}
```

Now we may get a list of images, but we can't update or remove an image if we don't have a valid token.

More information about middleware can be found at https://laravel.com/docs/master/controllers.

One more extra step, I only want an authenticated user to manage users, so open **UsersController.php**, and add:

```
public function __construct()
{
    $this->middleware('jwt.auth');
}
```

Sending a token with every request in Angular

Now we need to send a token from our Angular app to the API endpoints to access private data.

As mentioned earlier, we can pass in the **token that was received** before in our **HTTP header** using the **RequestOptions**:

```
this.http.post(apiUrl, body, options)
```

That should work fine. However, because we're using **angular2-jwt**, we can use its **AuthHttp** class to send requests!

In order to use the class, we have to create a new **Auth module** first.

Creating a module in Angular

Go to **app** directory, and create a new directory called **modules**:

```
mkdir modules
```

You may create a module manually or run this command to generate it:

```
ng g m auth --flat
```

> **Note:** If you don't use the **--flat** (or **-flat**) flag , the module will be put in a new **auth** folder. I don't want to create a new folder, so I use the **--flat** flag here.

Next, open **auth.module.ts**, and update it as follows:

```
import { NgModule } from '@angular/core';
import { Http, RequestOptions } from '@angular/http';
import { AuthHttp, AuthConfig } from 'angular2-jwt';

export function authHttpServiceFactory(http: Http, options: RequestOptions) {
  return new AuthHttp(new AuthConfig(), http, options);
}

@NgModule({
  providers: [
    {
      provide: AuthHttp,
      useFactory: authHttpServiceFactory,
      deps: [Http, RequestOptions]
    }
  ]
})
export class AuthModule {}
```

> **Note:** The code above is got from https://github.com/auth0/angular2-jwt

To use the module, we have to import it into the **app.module.ts**:

app.module.ts

```
import {AuthModule} from './modules/auth.module';
```

Last step, add **AuthModule** into the **imports** array:

```
imports: [
  routes,
  BrowserModule,
  FormsModule,
  ReactiveFormsModule,
  HttpModule,
  AuthModule
],
```

Great! The **AuthModule** is now ready to use!

Using the AuthHttp class

The **AuthHttp** class is actually a wrapper for **Angular's Http**, thus it has all the same **HTTP** methods.

To use the class, open **image.services.ts**, and import it into the service:

```
import {AuthHttp} from 'angular2-jwt';
```

After that, we can inject **AuthHttp** into the constructor:

```
constructor(private http: Http, private authHttp: AuthHttp) { }
```

And use it by changing **this.http** to **this.authHttp**.

```
getImage(id: String): Observable<Image[]> {
  return this.authHttp.get('http://angularbook.app/api/v1/images/' + id)
    .map((response: Response) => response.json());
}
```

Be sure to update any method that needs to send a token! (**addImage**, **updateImage**, **deleteImage**)

Let's open **user.service.ts**, import the **AuthHttp** class, and update the **User Service's constructor** as well:

```
import {AuthHttp} from 'angular2-jwt';
```

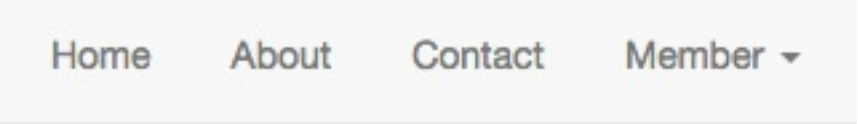
```
@Injectable()
export class UserService {

  constructor(private http:Http, private authHttp: AuthHttp) {

  }
```

Update all methods to send requests using **AuthHttp**.

Check our app again! We can access private data now!

Creating a custom directive to make our navigation bar work

Looks like our navigation bar is not working properly yet. When we click on the **member** button, nothing happens.



As you may know, to make the button work, we may import jQuery and the Bootstrap JS file. However, we're using Angular now, how about doing it in an Angular way?

We will make a custom directive, which uses **@Hostbinding** and **@HostListener** to reference elements (style, classes, properties, attributes) of a host (the navbar button).

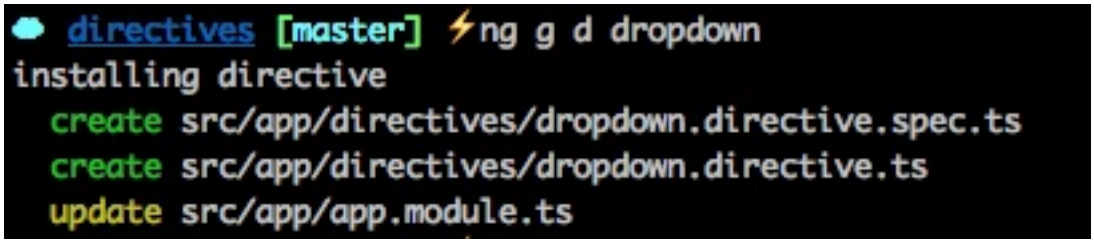
Don't worry if you don't understand it yet, I'll explain everything soon.

Now, let's make a new directive by going to the **app** directory. Create a new directory called **directives**, which is used to store all directives:

```
mkdir directives
```

Go to the **directives** directory, and run this command to generate a new directive called **dropdown**:

```
ng g d dropdown
```



If you see this error:

```
Error locating module for declaration
  SilentError: Multiple module files found:
```

Well, it's a bug of Angular CLI. Please try to update the Angular CLI to the latest version. Or you may create the directive manually by creating a new file called **dropdown.directive.ts**, and put it in the **directives** directory.

Here is its content:

dropdown.directive.ts

```
import {Directive, HostBinding, HostListener} from '@angular/core';

@Directive({
  selector: '[ngDropdown]'
})
export class DropdownDirective {

  private isOpen = false;

  @HostBinding('class.open') get opened() {
    return this.isOpen;
  }

  @HostListener('click') open() {
    this.isOpen = true;
  }

  @HostListener('mouseleave') close() {
    this.isOpen = false;
  }

}
```

Our directive is called **ngDropdown**, you may change to another name if you want:

```
@Directive({
      selector: '[ngDropdown]'
```

```
    })
```

Let's break things down again.

First, we create a boolean called **isOpen**. By default, it's set to **false**.

```
@HostBinding('class.open') get opened() {
  return this.isOpen;
}
```

Here we use **@HostBinding**, which is an **Angular decorator,** to link our directive property (**isOpen**) to a class (**open**) of a host (the **navbar** button). If **isOpen** is **true**, the class (**open**) is added otherwise removed.

After that, we use **@HostListener** to listen to a **click** event. When we click on the navbar button, **isOpen** is set to **true**.

```
@HostListener('click') open() {
  this.isOpen = true;
}
```

Again, we use **@HostListener** to listen to a **mouseleave** event. When we move the cursor out of the button, **isOpen** is set to **false**.

```
@HostListener('mouseleave') close() {
        this.isOpen = false;
      }
```

If you create the directive manually, be sure that the **DropdownDirective** has been imported to **app.module.ts**:

```
import {DropdownDirective} from './directives/dropdown.directive';
```

Then add the directive into the **declarations** array:

```
declarations: [
  AppComponent,
  NavbarComponent,
  GalleryComponent,
  ImageListComponent,
  ImageComponent,
```

```
    ImageDetailComponent,
    ContactComponent,
    AboutComponent,
    AdminComponent,
    AdminImageListComponent,
    DashboardComponent,
    AdminImageCreateComponent,
    AdminImageEditComponent,
    AdminImageDeleteComponent,
    AdminUserCreateComponent,
    AdminUserListComponent,
    RegisterComponent,
    LoginComponent,
    LogoutComponent,
    DropdownDirective
  ],
```

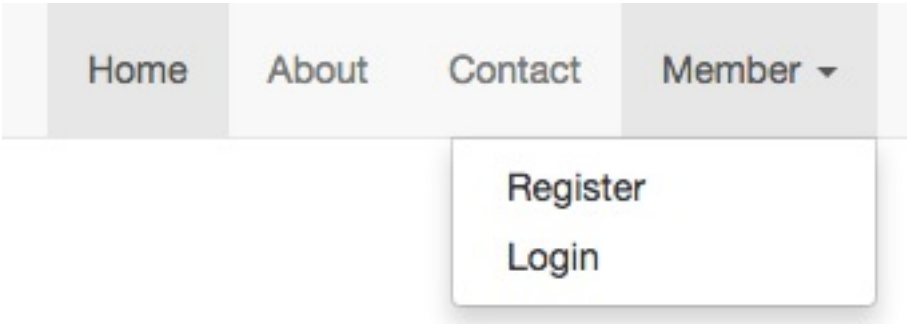
Our directive is now ready!

To use the directive, open **navbar.component.html**, and find:

```
<li class="dropdown">
```

Modify to:

```
<li class="dropdown" ngDropdown>
```



Now our navigation bar should be working as expected!

By adding **ngDropdown** directive to the **li** tag, a class called **open** will be added when we hover over the button.

For example:

```
<li ngdropdown="" class="dropdown">
```

will be changed to:

```
<li ngdropdown="" class="dropdown open">
```

That how it works!

You may read more information about **HostListener/HostBinding** at:

https://angular.io/docs/ts/latest/guide/style-guide.html

Displaying different drop-down menus based on the user state

Currently, our drop-down menu always shows **Register** and **Login**. That's not quite right. If we're logged in, it should show **Logout**, right?

Open **navbar.component.ts**, and add a new boolean called **isLoggedin**:

```
export class NavbarComponent implements OnInit {
  isLoggedIn = false;
```

After that, we can use **AuthService** and **Router** to check whether users **are logged in or not**. Update the **constructor** as follows:

```
constructor(private auth: AuthService, router:Router) {
  router.events.forEach((event) => {
    if(event instanceof NavigationStart) {
      if (this.auth.loggedIn()) {
        this.isLoggedIn = true;
      } else {
        this.isLoggedIn = false;
      }
    }
  });
}
```

Import **AuthService**, **NavigationStart**, **Router** into the component:

```
import {AuthService} from './services/auth.service';
import {NavigationStart, Router} from '@angular/router';
```

What's interesting to note is that we can use **router.events** to check if the route is changed:

```
router.events.forEach((event) => {
  if(event instanceof NavigationStart) {
```

When users move to another route, we re-check the users' state again.

Now we can use **ngIf** to display a different drop-down menu when users are logged in.

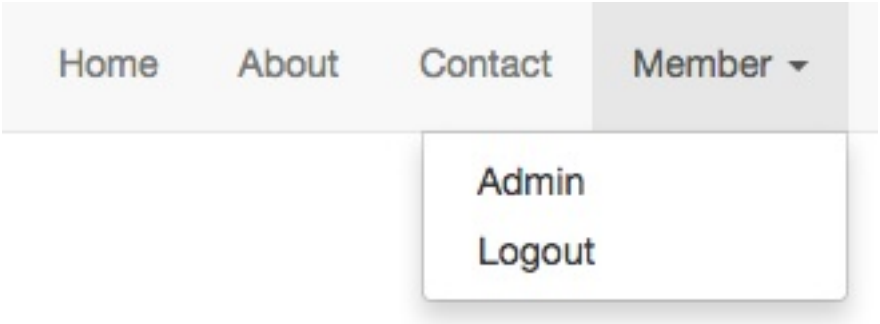
Open **navbar.component.html**, and find:

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

Modify to:

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

<ul *ngIf="isLoggedIn" class="dropdown-menu" role="menu">
  <li><a href="/admin">Admin</a></li>
  <li><a href="/logout">Logout</a></li>
</ul>
```

Testing time! If you're logged in, you should see a different drop-down menu:



Chapter 4 Summary

Congratulations! You've built a nice admin control panel!

Towards the end of this chapter, let's review what we have learned so far:

- You've known how to create child routes!
- You've learned how to use forms in two ways: **template-driven** approach and **code-driven** approach.
- Using **Http**, you can now send requests and get data from any API!
- You also know form states, route parameters, and two-way binding!
- Displaying status messages is easy, right?
- **JWT** is very helpful and powerful. You can use it in many real world applications (including mobile apps).
- **@HostBinding** and **@HostListener** are very useful decorators. Don't forget to use them to build more cool directives!
- After creating custom directives and custom modules, you can import them into other applications!

We hope you enjoyed this book and have learned a lot from it.

The app that we've made is simple, but it could be a good starter template for your amazing apps in the future.

If you have any feedback, please send us an email. We always love to hear feedback, so that we can continually improve the book.

Wish to learn more about other frameworks? Be sure to check our Laravel 5 Cookbook, Learning Laravel 5 book and Vue.js Book out!