# 1. Overview

Welcome to the Up and Running with Angular Workshop. Angular 2 was released in September 2016 and was a full re-write of the framework. With the re-write a lot has changed. In this workshop, you will learn how to build an Angular UI from the ground up.

> This workshop has been written for and tested with Angular 4.1

## 1.1 Goal

The goal of this tutorial is to walk you through creating an application with the common features that a web application would have (headers, footers, forms, calls to API services, authentication, etc).

The tutorial is designed to be gone through from start to finish to build the application as the chapters build on each other.

For each chapter there is a start and complete branch in Github that you can pull down if you need it.

## 1.2 New Naming Convention

The first thing is what to call Angular. Angular 1 is now referred to as AngularJS. Angular 2 and above is referred to as just Angular. From here on out, we will just call it Angular. The only time to refer to Angular with a version number is when you are talking about a feature that it specific to that version. For example, in Angular 4, they add the TitleCase Pipe and an email validator.

## 1.3 Release Schedule

The second thing is that they are now following SemVer for the versioning of Angular and have a release schedule. For example, here is how version 2.3.1 would break down.


Release Schedule:

- Major: Every 6 months
- Minor: Every month
- Patch: Every week

# 1.4 Coming from Angular 1

Third, if you are coming from an Angular 1 background, it is best if you do not try to compare the concepts between Angular 1 and Angular 2+. You will be able to learn Angular Just treat Angular 2 as a whole new framework

# 1.5 Materials

Completed Source Code

# 1.6 Using This Tutorial

### Collapse Chapter List

If you click on the hamburger icon in the header (left icon with the 3 lines), it will toggle the view of the chapters so that you have more room to view the tutorial.

### Copy Code Snippets

Every code snippet has a copy button on the right side of the snippet that will put the code onto the clipboard for you to paste into your code.

## Found An Issue with Tutorial

If we are doing this tutorial in person as part of a class or workshop, please flag me down and let me know.

If you are going through this tutorial online by yourself, click on the github icon in the header (right icon) and log an issue so that I can take a look at it.

# 2. Getting up and running

## 2.1 Overview

To get started, we need to install and configure the following software first.

- Windows, Mac, or Linux computer
- Visual Studio Code (can use any text editor but workshop tested with Visual Studio Code)
- Node 6.9+
- Angular CLI 1.0.0+

## 2.2 Goals

- Under stand how to install the Angular Command Line (CLI)
- Setup your machine for the Workshop

## 2.3 Windows Showing File Extensions

## Windows Only

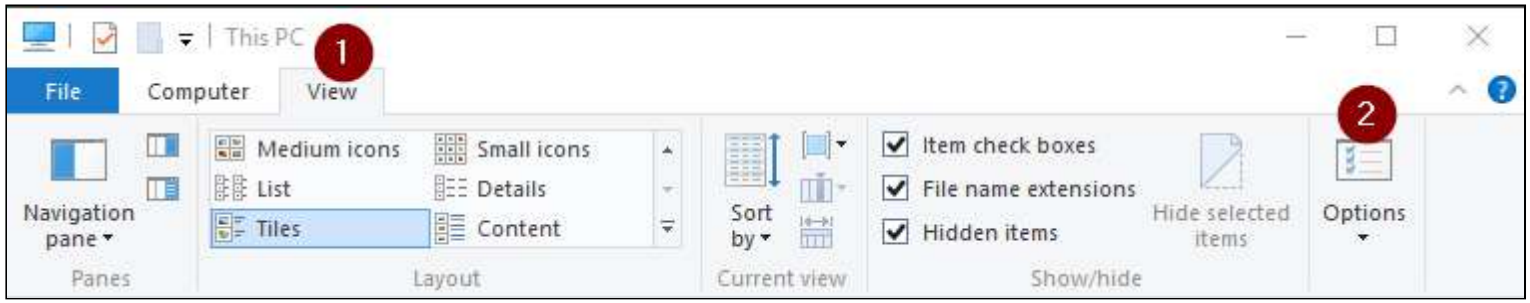
### Non-Windows users can skip to next section

By default Windows is set to not show file extensions for known files which causes files such as .gitconfig and .npmrc to show up as just a period with no file extension which makes it extremely difficult to figure out what the file actually is. To fix this we need to turn set Windows Explorer to show file extensions.

### Exercise: Turn On Windows Showing File Extensions

1. Open Windows Explorer

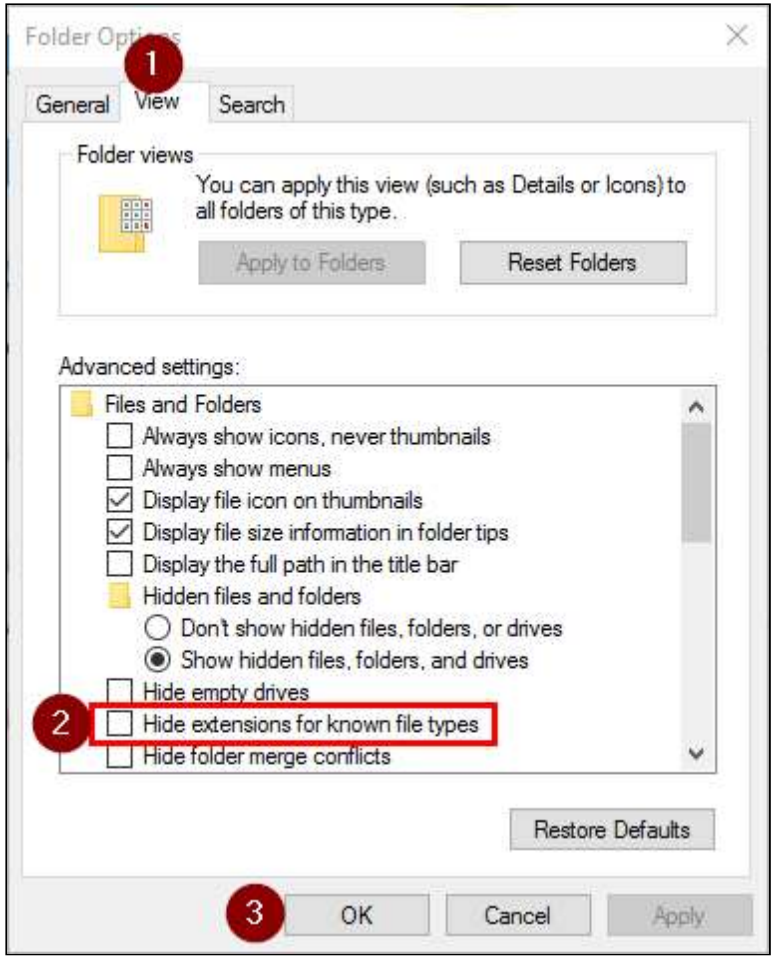
2. Click on the View Tab and select Options




Once the "Folder Options" dialog is open:

1. Click on the View Tab
2. Uncheck the "Hide extensions for known file types"

3. Click Ok




## 2.4 Visual Studio Code

Visual Studio Code is Microsoft lightweight cross platform IDE.

1. Download Visual Studio Code at https://code.visualstudio.com/
2. Once the download finishes, launch the installer except all of the defaults.

## 2.5 Node.js

NodeJS is used to power the Angular CLI as well as install all of our dependencies. The Angular CLI requires Node version 6.9 or greater.

1. Download the latest stable version (LTS) of NodeJS which as of this writing is 6.11.0.
2. Run the installer and accept all defaults.

3. Verify that Node installed. Start a command prompt or terminal window and run:

```
node -v
```

## 2.6 Angular CLI Install

The Angular CLI (Command Line Interface) makes it so that you do not have to worry about the Angular tooling and can focus instead on your code. You can create new projects, components, modules, services, guards, pipes and routes. As well it has commands for linting, testing and running our code.

All of the code that is generated by the Angular CLI following the Angular Style Guide.

While you do not have to use the Angular CLI, it is highly recommended, will increase your productivity, and this workshop only gives the instructions for developing with the Angular CLI.

### Exercise: Install Angular CLI

1. Open a command prompt or terminal and run the following command

Copy

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

2. Verify Angular CLI

Copy

```
ng --version
```

# 3. Create New Project

## 3.1 Overview

In this chapter we will be creating our project with the Angular CLI. We will use this project throughout the workshop.

## 3.2 Goals

- Understand how to create a new project
- Understand how to find help for an Angular CLI command
- Understand the project layout
- Understand how to run the project

## 3.3 Generate New Project

The project that the Angular CLI create for you follows all of the suggested standards and has webpack for bundling built-in to it.

## Exercise: Create Angular Project

1. Open a command prompt

2. Navigate to where you want to store your project files. I use c:\projects on Windows and ~/projects on OSx. You are free to use anywhere that you want.

    - Windows:

```
cd \
```

    - OSx:

```
cd ~/
```

3. Create the projects directory. If you already have a directory that you store your projects in then you can skip this step.

```
mkdir projects
```

4. Navigate into the projects directory

```
cd projects
```

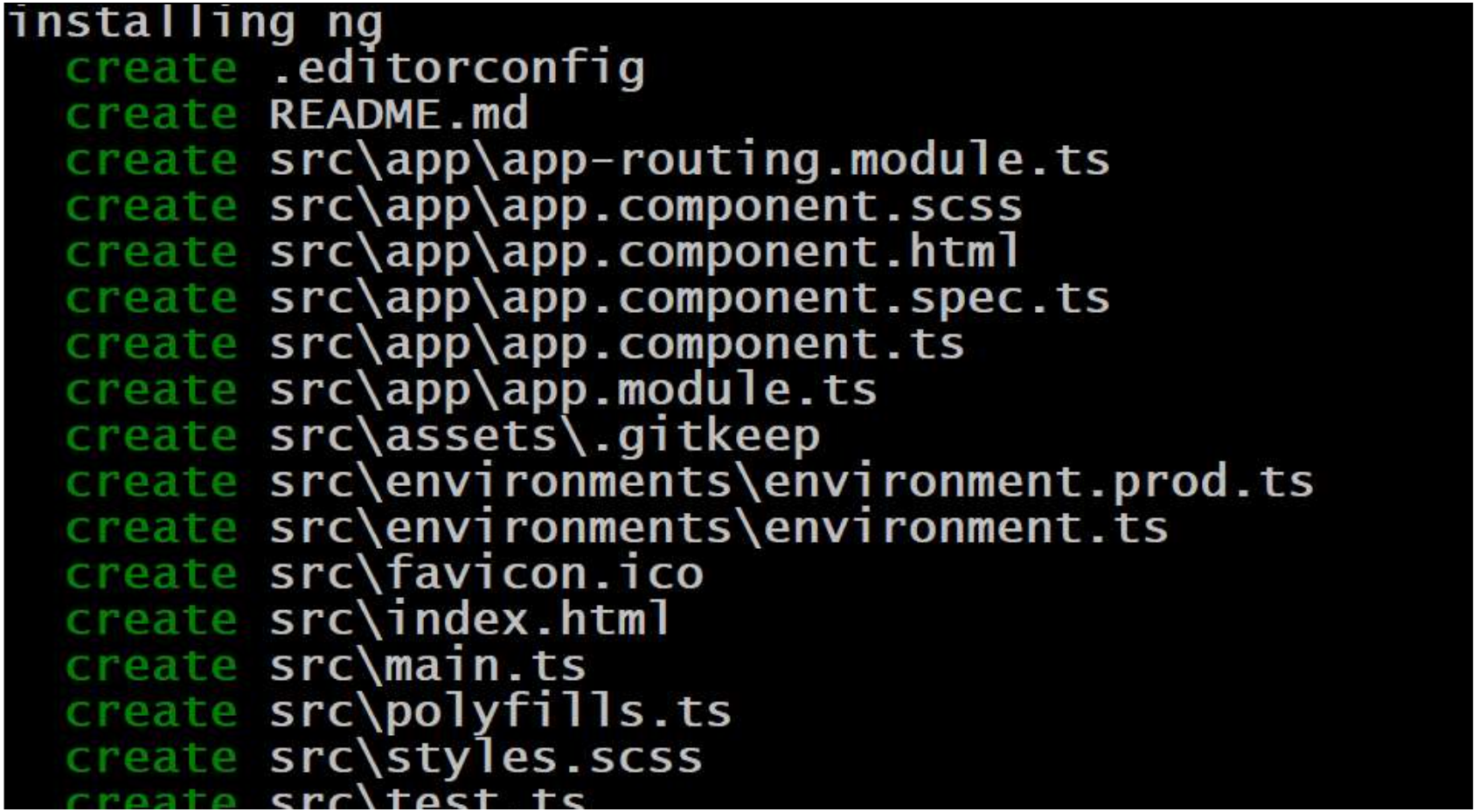
Copy

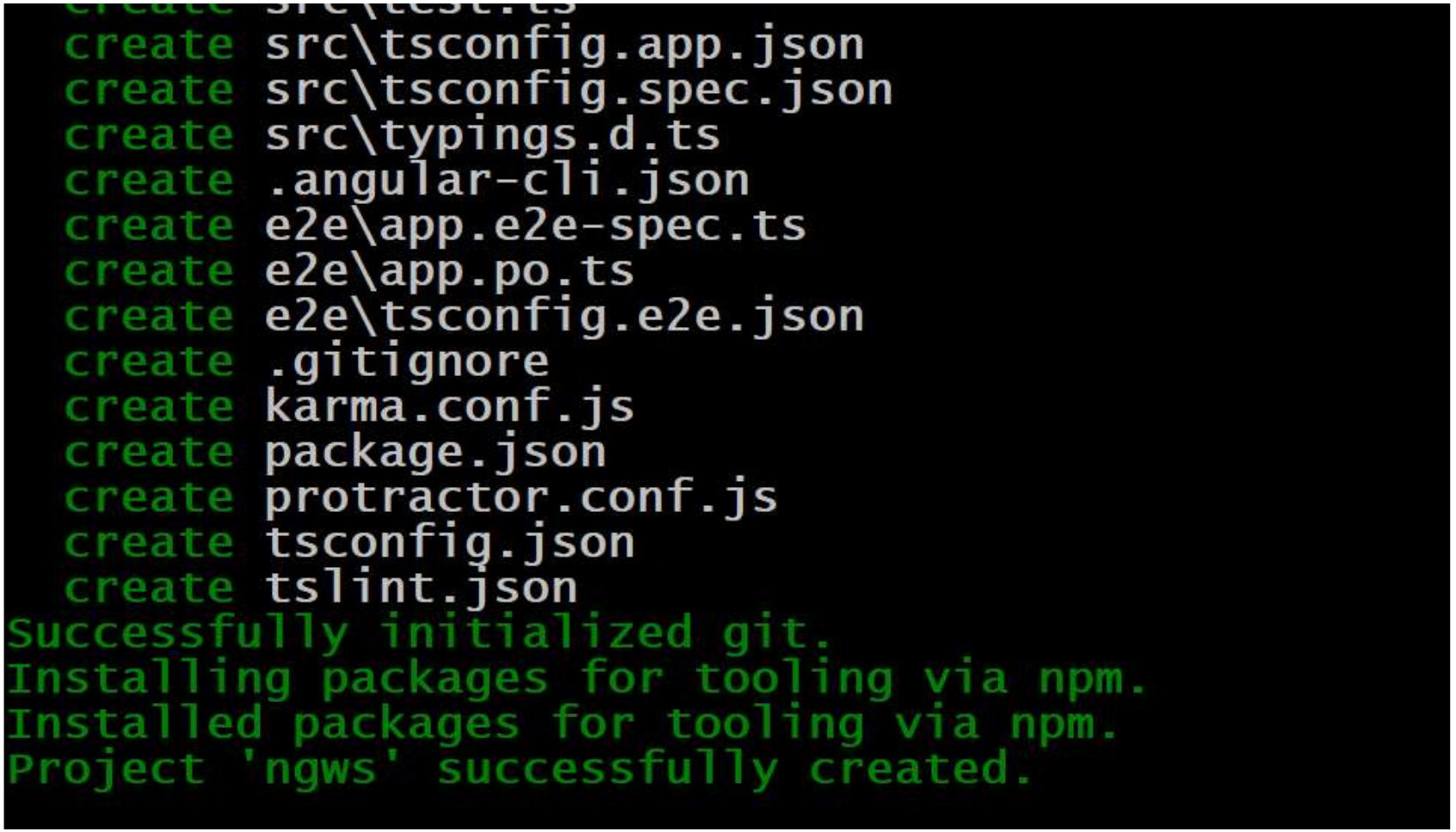
5. Generate a project named ngws that uses scss for styling and includes a routing file by running

Copy

```
ng new ngws --style scss --routing
```

### Sample Output:






6. If you want to see the other ng new options or any Angular CLI command append the --help

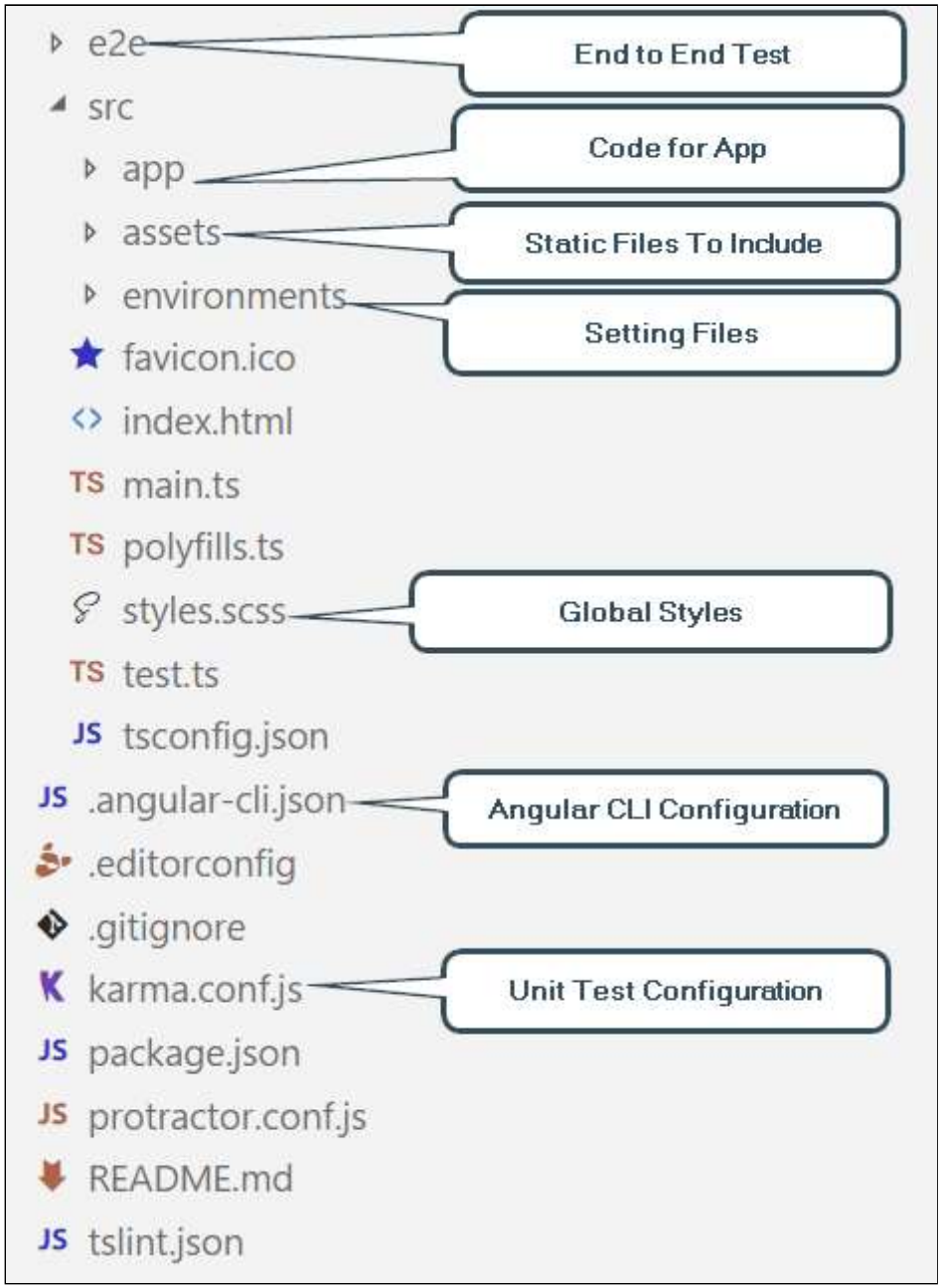
Copy

```
ng new --help
```

# 3.4 Opening Project in Visual Studio Code

1. Open Visual Studio Code
2. Click File -> Open Folder...
3. Navigate to angular-tutorial directory and click Select Folder

4. Your project should now be opened in Visual Studio Code


e2e
End to End Test
src
app
Code for App
assets
Static Files To Include
environments
Setting Files
favicon.ico
index.html
main.ts
polyfills.ts
styles.scss
Global Styles
test.ts
tsconfig.json
.angular-cli.json
Angular CLI Configuration
.editorconfig
.gitignore
karma.conf.js
Unit Test Configuration
package.json
protractor.conf.js
README.md
tslint.json

# 3.5 Running Project

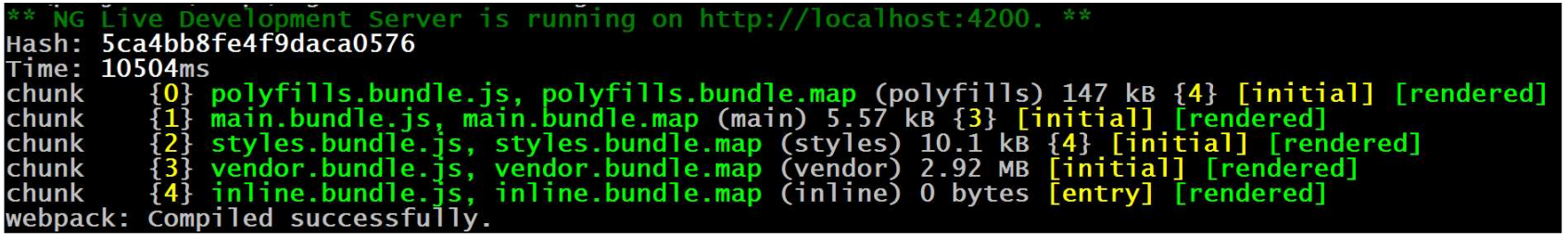
### Exercise: Run the project

The Angular CLI has a built-in command for starting up a web server for your project called `ng serve` which will run webpack to bundle up your code, start the web server, rebuild on file changes (watch) and refresh connected browsers (live reload).

1. Visual Studio Code has a built-in terminal that we can use to run our commands. On Windows, this is a powershell prompt. To open the Integrated Terminal go under the View Menu and click on the Integrate Terminal or press Ctrl+`
    - You are free to use the regular command prompt outside of Visual Studio Code if you would like
2. Run

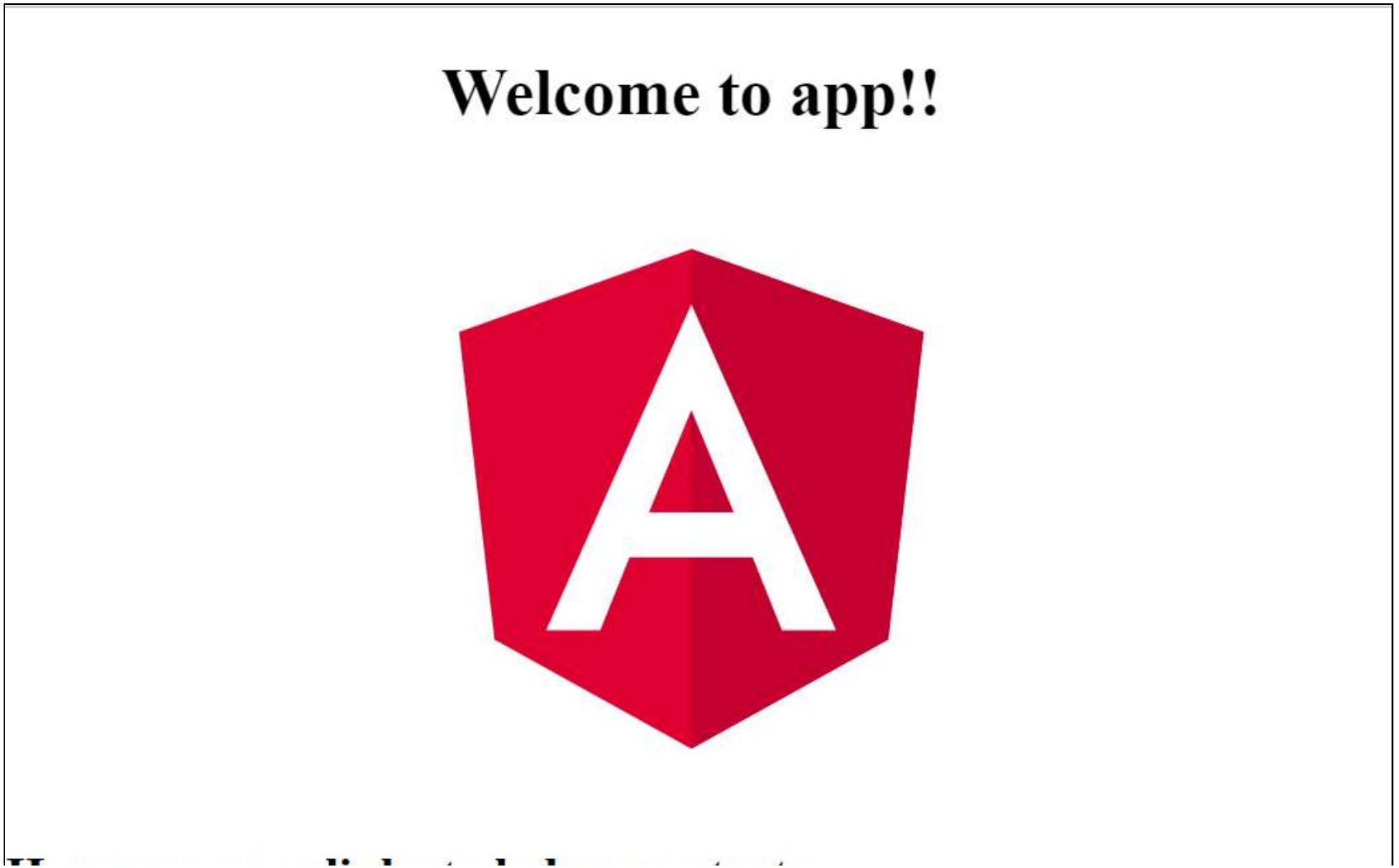
Copy

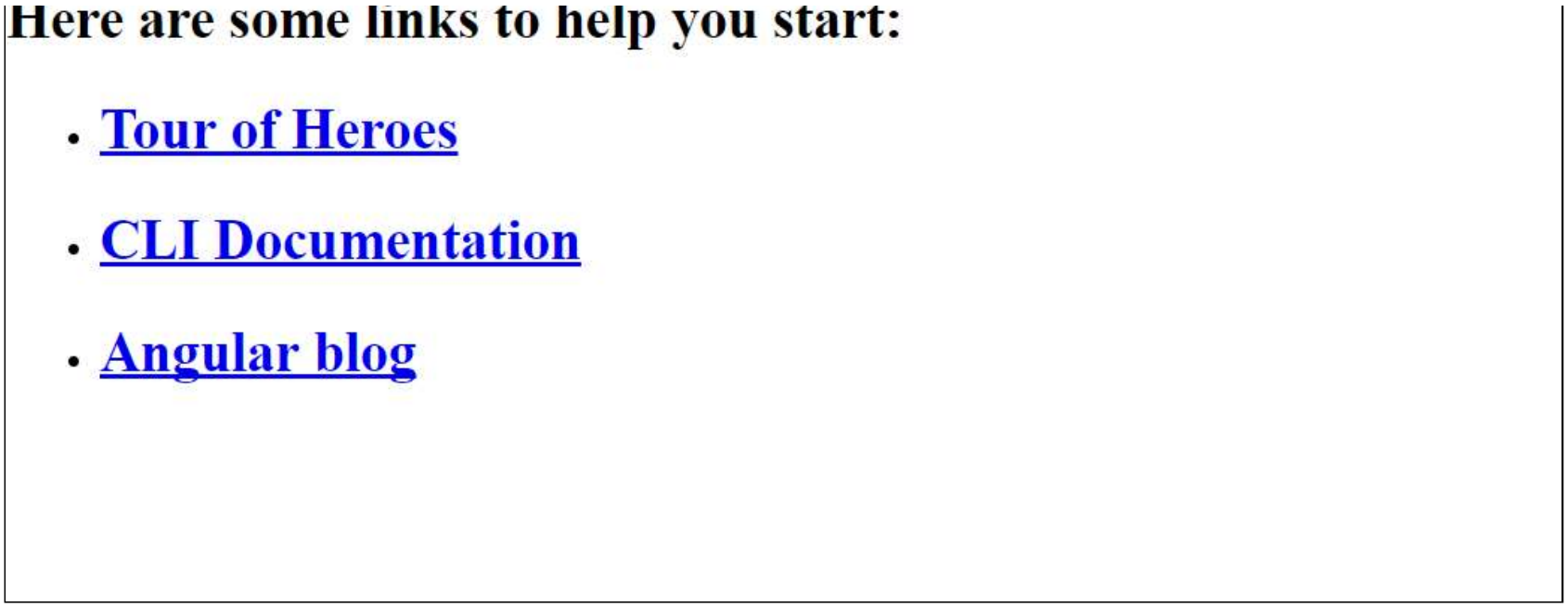
```
ng serve
```



The ng serve command will stay running in order to provide live reloading functionality. If you need to stop ng serve, press ctrl+c

3. If you launch your browser and navigate to http://localhost:4200, you will see a page that looks like






## 3.6 Review

In this chapter we learned 3 things

1. How to create a new projects using the `ng new` command
2. What the project layout looks like for an Angular project generated with the CLI
3. How to run our project with the `ng serve` command and view it at http://localhost:4200

# 4. Bootstrap

## 4.1 Overview

For our styling we are going to use Bootstrap 4 which is still in alpha. The reason for picking Bootstrap 4 and not Bootstrap 3 is so that you will be able to use the Angular UI library if you so choose. We will not be using the Angular UI library in this workshop but it is a great library for component. If you are a Bootstrap 3 developer, you can see all of the change in the Migration Guide.

## 4.2 Goals

- Understand how to include Bootstrap 4 into your project

## 4.3 Install Bootstrap

**Exercise: Install Bootstrap**

1. In the VS Code Integrated Terminal, click the + to open a 2nd terminal

2. Run the npm install command for bootstrap and font-awesome

Copy

```
npm install --save bootstrap@next font-awesome
```

## 4.4 Add Bootstrap to Project

### Exercise: Add Bootstrap to Project

First we need to create our own custom Bootstrap style sheet so that we can override the scss variables if we want to with your own colors and styles.

1. In the src\assets folder, create a new folder named **bootstrap**

You can create folder right in Visual Studio Code by right-click on the src\assets folder

Copy

```
bootstrap
```

2. In the bootstrap folder, create a file called **_variables.scss**

> You can create file right in Visual Studio Code by right-click on the src\assets folder

Copy

```
_variables.scss
```

3. Add the following to the _variables.scss file

Copy

```
// Variables
// Colors
//
// Grayscale and brand colors for use across Bootstrap.
$dark-blue: #003C71;

// Start with assigning color names to specific hex values.
$white:  #fff !default;
$black:  #000 !default;
$red:    #d9534f !default;
$orange: #f0ad4e !default;
$yellow: #ffd500 !default;
$green:  #5cb85c !default;
```

```
$blue:   #0275d8 !default;
$teal:   #5bc0de !default;
$pink:   #ff5b77 !default;
$purple: #613d7c !default;

// Create grayscale
$gray-dark:                #292b2c !default;
$gray:                     #464a4c !default;
$gray-light:               #636c72 !default;
$gray-lighter:             #eceeef !default;
$gray-lightest:            #f7f7f9 !default;

// Reassign color vars to semantic color scheme
$brand-primary:            $blue !default;
$brand-success:            $green !default;
$brand-info:               $teal !default;
$brand-warning:            $orange !default;
$brand-danger:             $red !default;
$brand-inverse:            $gray-dark !default;
```

4. In the bootstrap folder, create a file called **bootstrap.scss**

```
bootstrap.scss
```

Copy

5. Add the following contents to the bootstrap.scss file.

> Note that the list of included files outside of the variables file is the same as the ones in the node_modules\bootstrap\scss\bootstrap.scss file with a ~bootstrap/scss prefix so that the scss compiler is able to find the included file in the node_modules\bootstrap\scss directory.

Copy

```
@import "variables";

@import "~bootstrap/scss/variables";
@import "~bootstrap/scss/mixins";
@import "~bootstrap/scss/custom";

// Reset and dependencies
@import "~bootstrap/scss/normalize";
@import "~bootstrap/scss/print";

// Core CSS
@import "~bootstrap/scss/reboot";
@import "~bootstrap/scss/type";
```

```
@import "~bootstrap/scss/images";
@import "~bootstrap/scss/code";
@import "~bootstrap/scss/grid";
@import "~bootstrap/scss/tables";
@import "~bootstrap/scss/forms";
@import "~bootstrap/scss/buttons";

// Components
@import "~bootstrap/scss/transitions";
@import "~bootstrap/scss/dropdown";
@import "~bootstrap/scss/button-group";
@import "~bootstrap/scss/input-group";
@import "~bootstrap/scss/custom-forms";
@import "~bootstrap/scss/nav";
@import "~bootstrap/scss/navbar";
@import "~bootstrap/scss/card";
@import "~bootstrap/scss/breadcrumb";
@import "~bootstrap/scss/pagination";
@import "~bootstrap/scss/badge";
@import "~bootstrap/scss/jumbotron";
@import "~bootstrap/scss/alert";
@import "~bootstrap/scss/progress";
@import "~bootstrap/scss/media";
```

```
@import "~bootstrap/scss/list-group";
@import "~bootstrap/scss/responsive-embed";
@import "~bootstrap/scss/close";

// Components w/ JavaScript
@import "~bootstrap/scss/modal";
@import "~bootstrap/scss/tooltip";
@import "~bootstrap/scss/popover";
@import "~bootstrap/scss/carousel";

// Utility classes
@import "~bootstrap/scss/utilities";

/*
* Font Awesome 4.x
*/
$fa-font-path: "~font-awesome/fonts";
@import "~font-awesome/scss/font-awesome";
```

Now we need to configure the angular cli to import the bootstrap libraries.

1. Open the **.angular-cli.json** file that is in the root of the project. This file is the configuration file for our project for the Angular CLI

In Visual Studio Code you can quickly open a file by using ctrl+p to open the "Go To File" prompt and typing in the file name

Copy

```
.angular-cli.json
```

2. Find the apps\styles section and replace the section with:

Copy

```
  "styles": [
    "assets/bootstrap/bootstrap.scss",
     "styles.scss"
  ],
```

If the code alignment upon paste is off you can run the "Format Document" command to fix it. Ctrl+Shift+P and search for "Format Document" or use Alt+Shift+F

## 4.5 Add Banner Section to Top

### Exercise: Add Banner To Top of Page

1. Open the src\app\app.compoment.html file and replace the contents with:

Copy

```
<div class="jumbotron">
    <div class="container">
        <h1>{{title}}</h1>
    </div>
</div>
<div class="container">
    <router-outlet></router-outlet>
</div>
```

2. Lets change the title to something better than "App Works!"
    - Open the src\app\app.component.ts file
    - On line 9, change the title variable to

Copy

```
title = 'Our Awesome Todo App!';
```

# 4.6 View Changes

Since we already had ng serve running, it picked up the html changes but you will notice that there is no styling. This is because whenever you change the .angular-cli.json file, you need to restart ng serve before they take effect.

### Exercise: Restart ng serve

1. Go to the integrated terminal in Visual Studio Code that is running the `ng serve` command and do a ctrl+c to stop it.

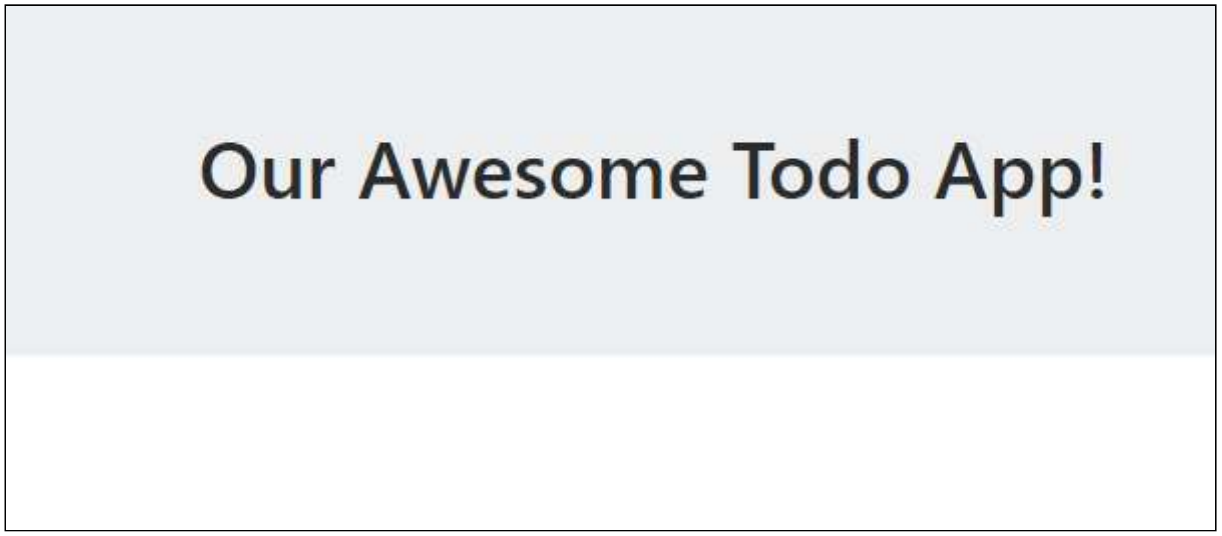
   > There is a dropdown on the right side of the Visual Studio Code integrated terminal that allows you to change to the other terminals that are currently open for your project

2. Run the `ng serve` command again.

   Copy

   ```
   ng serve
   ```

3. The web page should now look like




## 4.7 Review

In this chapter we learned how to use Bootstrap for our project.

Learned:

1. How to install Bootstrap 4 and font-awesome
2. How to integrate it into the Angular CLI in the .angular-cli.json file
3. How to create our own custom Bootstrap SCSS variables to override the built-in styles of Bootstrap with our own colors
4. How to create a banner at the top of the page
5. How to change the text that appears in the banner using our TypeScript title variable

6. Learned that when you modify the .angular-cli.json file that you have to restart `ng serve` for the changes to take effect

# 5. Template Based Forms

## 5.1 Overview

There are 2 options for creating forms within Angular: Template based and Model based. This chapter will cover template based forms.

Template based forms are simple and easy to use. They are great for simple use cases. However, there are some limitations with template based forms: 1.) You end up with a lot of logic in your html code 2.) Cross field validation is more difficult 3.) You can not unit test your form and validation logic.

To demonstrate template based forms, we are going to be build a login component.

## 5.2 Goals

- Understand template based forms
- Create template based form
- Implement input validation
- Submit form values to a service

# 5.3 Create Login Component

The first thing we need to do is to create our login component using the Angular CLI. We will use the `ng generate component` command to do this.

### Exercise: Create Login Component

1. Within VS Code, open up the integrated terminal (ctrl+`) or view menu and then "Integrated Terminal"

2. Run the ng generate command below to create the login component

Copy

```
ng generate component login
```

3. The generate command will create 4 files:

```
$ ng generate component login
installing component
  create src/app/login/login.component.scss
  create src/app/login/login.component.html
  create src/app/login/login.component.spec.ts
  create src/app/login/login.component.ts
  update src/app/app.module.ts
```

- scss (styles)
- html (view)
- spec file (test)
- component (controller)
- add reference to the component in the app.module.ts file.

## 5.4 Add Route

In order to view our Login component we need to tell Angular how to route to it. We are going to add a route at '/login' to the LoginComponent.

**Exercise: Login Routing**

Before we can view our Login component, we need to tell Angular how to route to the component

1. Open the src\app\app-routing.module.ts file

Copy

```
app-routing.module.ts
```

2. You need to import the LoginComponent before we can reference it. On line 3, add the following statement

Copy

```
import { LoginComponent } from './login/login.component';
```

3. In the routes array, we need to add another array item to be able to route to the LoginComponent. Add the following at the end of line 9

Copy

```
,
{ path: 'login', children: [], component: LoginComponent }
```

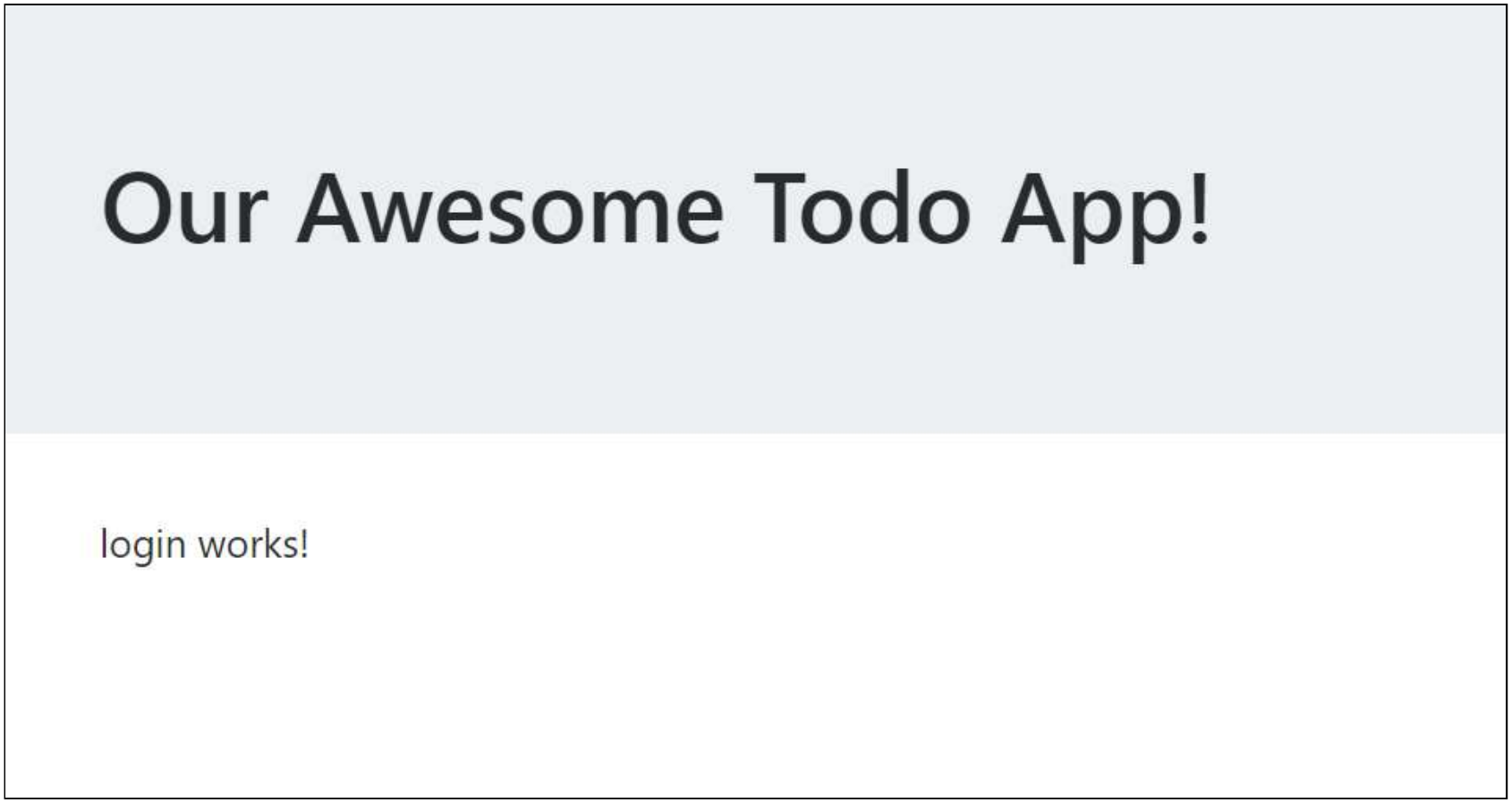
4. Your routes should look like the following

Copy

```
const routes: Routes = [
    {
        path: '',
```

```
        children: []
    },
    { path: 'login', children: [], component: LoginComponent }
];
```

5. The Login page should display when you navigate to http://localhost:4200/login



**Note:** When you navigate to the login route, the LoginComponent is loaded into the `<router-outlet></router-outlet>` in the html in src\app\app.component.html. The router-outlet tag is how Angular knows where to put the rendered content for the route.

## 5.5 Create Form

Next we are going to create the form without any validation logic at all. Our form will have 2 input fields: email and password and 2 buttons: submit and cancel. We will use Bootstrap for the styling and Angular for the validation.

### Exercise: Creating the Form

1. Open src\app\login\login.component.html

Copy

```
login.component.html
```

2. Replace the existing html with the following

Copy

```
<h1>Login</h1>
<hr>
```

```
 <div>
     <form autocomplete="off" novalidate>
         <div class="form-group">
             <label for="email">Email:</label>
             <input name="email" id="email" type="text" class="form-control"
placeholder="Email..." />
         </div>
         <div class="form-group">
             <label for="password">Password:</label>
             <input name="password" id="password" id="password" type="password"
class="form-control" placeholder="Password..." />
         </div>
         <button type="submit" class="btn btn-primary">Login</button>
         <button type="button" class="btn btn-default">Cancel</button>
     </form>
 </div>
```

- This form is using Bootstrap for the styling with the form-group, form-control btn, and btn-* css classes.
- The autocomplete="off" and novalidate directives for the form tag tell the browser to turn off that built-in functionality so that we can do our validation with Angular.

# 5.6 Handle Form Submission

In order to submit the form we need to tell Angular that that login form is an ngForm and use the ngSubmit event to tell Angular what to do when the form submit button is clicked.

### Exercise: Import Angular FormsModule

Before we can interact with our form using Angular, we need to import the FormsModule into our application.

1. Open src\app\app.module.ts

Copy

```
app.module.ts
```

2. Add an import statement for FormsModule from @angular/forms

Copy

```
import { FormsModule } from '@angular/forms';
```

3. Before we can use the imported module, we need to add it to the @NgModule declaration in the imports array. We are going to add it between the BrowserModule and AppRoutingModule.

Copy

```
FormsModule,
```

4. Your @NgModule imports section should now look like:

Copy

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

### Exercise: Add login function

1. Go back to the login.component.html file

Copy

```
login.component.html
```

2. On the `<form>` tag we need to add 2 additional attributes to create a reference variable to the form and handle the form submit.

Copy

```
#loginForm="ngForm" (ngSubmit)="login(loginForm.value)"
```

- The #loginForm creates a variable called loginForm that reference ngForm. This will allow us to reference the loginForm in the login.component.ts file.
- The ngSubmit upon form submit event will call the login function in the login.component.ts file and pass the loginForm.value into it. The loginForm.value holds the values for the form fields.

In order to be able to access all of the form field values by using `loginForm.value` we need tdo add an `(ngModel)="NAME"` attribute onto each field, replacing "NAME" with the name that we want to refer to the field as.

1. On the email input field we need to add an attribute to tell Angular to implement 1 way binding on the field by using the `(ngModel)` attribute

Copy

```
(ngModel)="email"
```

2. On the password field we are going to do the same thing that we just did to the email field.

Copy

```
(ngModel)="password"
```

Now that the basics of the form have been created, we are ready handle the form submit event and implement the login function that `(ngSubmit)` is calling.

1. Open the login.component.ts file

login.component.ts

Copy

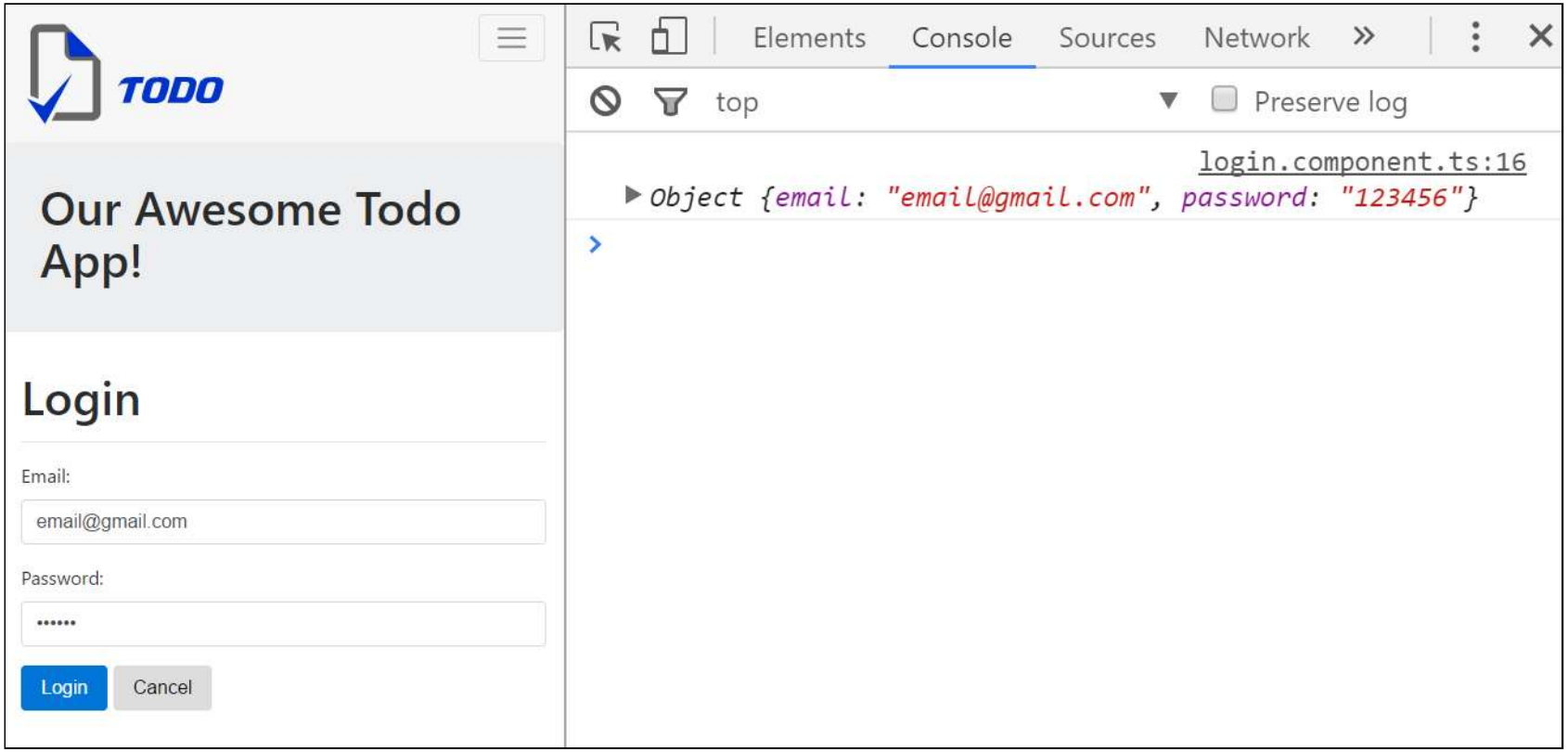
2. Inside of the LoginComponent class after the ngOnInit, add the following function. For now we are just going to output the form values to the Developer Tools console in the browser.

Copy

```
login(formValues) {
    console.log(formValues);
}
```

3. In your browser, while viewing the login form at http://localhost:4200/login open the developer tools and click the console tab so that you can see the console output.

4. Now enter a value into the email and password fields and click the submit button. You will see output similar to the below image where I entered the email of email@gmail.com and password of 123456




## 5.7 Implement Login Service

Now that we have our form done, we are going to implement our login service.

**Exercise: Generate Service**

1. Within VS Code, open up the integrated terminal (ctrl+`) or view menu and then "Integrated Terminal"

2. Run the ng generate command below to create the Authorization service. I like to store my services under a shared\services folder.

Copy

```
ng generate service shared/services/auth --module App
```

3. The generate command will create 2 files and update app.module.ts:

```
$ ng generate service shared/services/auth --module App
installing service
  create src\app\shared\services\auth.service.spec.ts
  create src\app\shared\services\auth.service.ts
  update src\app\App.module.ts
```

- spec file (test)
- typescript (service)
- updated app.module.ts to add LoginService as a provider

Before we can make HTTP calls in our AuthService, we need to import the HttpModule into your AppModule

1. Open the app.module.ts file

app.module.ts

Copy

2. Add an import statement for the HttpModule that comes from @angular/http

Copy

```
import { HttpModule } from '@angular/http';
```

3. In the @NgModule imports array add the HttpModule

Copy

```
HttpModule,
```

## Exercise: Create Class

In the AuthService, in order to hold our user data and get type checking we need to create a TypeScript class with an email and id field. We are going to leave the password field out of the class as we do not want to store this in memory at all.

1. Within VS Code, open up the integrated terminal (ctrl+`) or view menu and then "Integrated Terminal"

2. Run the ng generate command below to create the Authorization service. I like to store my services under a shared\services folder.

```
ng generate class shared/classes/User
```

Copy

3. The generate command will the user.ts file in the shared/classes folder:


1. Open the src\app\shared\classes\User.ts file

Copy

```
user.ts
```

2. Within the User class, add the following fields. Note that the createdAt and updatedAt are automatically added by the API.

Copy

```
email: string;
id: string;
createdAt: Date;
updatedAt: Date;
```

3. Within the User class and before the fields we just added, create a constructor that requires an email and make an id field optional (hint: the `?` makes the parameter optional)

Copy

```
constructor(email: string, id?: string, createdAt?: Date, updatedAt?: Date){
    this.email = email;
    this.id = id;
    if (createdAt) this.createdAt = createdAt;
    if (updatedAt) this.updatedAt = updatedAt;
}
```

## Exercise: Implement Auth Service Login Function

For this tutorial, I have created an API for us to use.

The first thing we are going to do is create our login function

1. Open the src\shared\services\auth.service.ts file

Copy

```
auth.service.ts
```

2. Import the following so that we can make our HTTP calls and get a response back.

```
import { Http, Response, RequestOptions } from '@angular/http';
import { Observable } from 'rxjs/Rx';
```

Copy

1. In order to use the HTTP module, we need to inject it into our constructor

```
constructor(private http: Http) {
}
```

Copy

2. Import the User class that we created earlier

```
import { User } from '../classes/user';
```

Copy

3. We also want to create a public variable within the AuthClass to hold the output from the API call

```
public currentUser: User;
```

Copy

4. For the API that we are using (SailsJS based), it requires that we set the HTTP option to allow credentials so that the session cookie can be passed back and forth, else it will always think you haven't logged in.

```
private options = new RequestOptions({ withCredentials: true });
```

Copy

5. Next we need to create our login function within the AuthService class that will call our API. Place the login function after the constructor.

> For now we are hard coding the API url into the service. In the "Environment Configuration" chapter we will change this to pull from a configuration file

Copy

```
 login(email: string, password: string): Observable<boolean | Response> {
     let loginInfo = { "email": email, "password": password };
     return this.http.put("https://dj-sails-todo.azurewebsites.net/user/login", loginInfo,
this.options)
         .do((res: Response) => {
             if (res){
                 this.currentUser = <User>res.json();
             }
         })
         .catch(error => {
             console.log('login error', error)
             return Observable.of(false);
         });
 }
```

- This code setups the call to the login API, stores the response in the currentUser variable, and returns back an Observable.

- Note that this code is not called until someone subscribes to the login function which we will be doing next.

> The API is setup for username/password validation. Make sure you do not use your real passwords as this is just a test API and not production secured.

### Exercise: Call AuthService from LoginComponent

Now that we have our AuthService completed, we need to call it from our LoginComponent. If we get a user back we will redirect the user to the home page

1. Open the src\login\login.component.ts file

Copy

```
login.component.ts
```

2. Import the AuthService and Router so that we can make call our LoginService and redirect upon successful login.

Copy

```
import { AuthService } from '../shared/services/auth.service';
import { Router } from '@angular/router';
```

3. In order to use the AuthService and Router, we need to inject it into our constructor

Copy

```
constructor(private authService: AuthService, private router: Router) { }
```

4. Next we need to update our login function to call the AuthService and redirect if it finds the user.

Copy

```
  login(formValues) {
this.authService.login(formValues.email, formValues.password)
  .subscribe(result => {
    if (!this.authService.currentUser) {
      console.log('user not found');
    } else {
      this.router.navigate(['/']);
    }
  });
}
```

5. Navigate to http://localhost:4200/login .

- If you enter an email of foo@foo.com with a password 123456 you should be redirected to the home page
- If you enter a bogus email or password, you will see a "user not found" message in the browser developer tools console.

### Exercise: Show Invalid Login Message

Up to this point, we have been using the console to output when the login failed but we can not expect the user to have the dev tools console open. Instead we need to show to the user when there is an error.

1. Open the login.component.ts file

Copy

```
login.component.ts
```

2. Create a new variable inside of the LoginComponent class called invalidLogin, is of type boolean and the default value is false.

Copy

```
invalidLogin: boolean = false;
```

3. Now in the login method replace the console.log line and set the invalidLogin variable to true. Make sure in the else statement that you set the invalidLogin to false.

```
this.invalidLogin = true;
```

Copy

- We have to use this. to access the variable due to scoping.

Now we are ready to implement the UI to show the error message.

1. Open the login.component.html file

Copy

```
login.component.html
```

2. We want to put our message after the cancel button but inside the `</form>` tag

3. Add the following alert message

Copy

```
<div *ngIf="invalidLogin" class="alert alert-danger">
    Invalid Login
</div>
```

# 5.8 Required Validation

**Exercise: Add Required Validation**

A standard requirement for html forms is to have client side validation. In this section we are going to implement the required field validation for both email and password.

Adding required validation is as easy as adding a required attribute to the input field

1. Open the login.component.html

Copy

```
login.component.html
```

2. On the email and password fields add the `required` attribute

Copy

```
required
```

## Exercise: Display Validation Errors

Next we want to display a message to the user when they have invalid entries in the form fields.

1. In order to refer to the email field by name when checking errors, we need to add the `#email="ngModel"` attribute to the email input field to tell Angular to create a variable for the control and make the value the

ngModel value for the email field.

Copy

```
#email="ngModel"
```

2. For the email field, inside of the form-group div tag and after the input field add the following code. This code will show when the required errors triggers. To trigger the error message, click on the email field and then click on the password field.

Copy

```
<div *ngIf="email.errors && (email.dirty || email.touched)" class="alert alert-danger">
    <div [hidden]="!email.errors.required">
    Email is required
    </div>
</div>
```

3. Just like the email field, we need to add the `#password="ngModel"` attribute to the input control to name the field

Copy

```
#password="ngModel"
```

4. For the password field, inside of the form-group div tag and after the input field add the following code. This code will show when the required errors triggers. To trigger the error message, click on the password field and then click on the email field.

```
 <div *ngIf="password.errors && (password.dirty || password.touched)" class="alert alert-
danger">
     <div [hidden]="!password.errors.required">
     Password is required
     </div>
 </div>
```

Copy

## 5.9 Email Validation

### Exercise: Add Email Validation

With the Angular 4 release, they added an email validator. To add the validator to our email field, we just need to add an `email` attribute like we did for the `required` attribute.

Copy

```
email
```

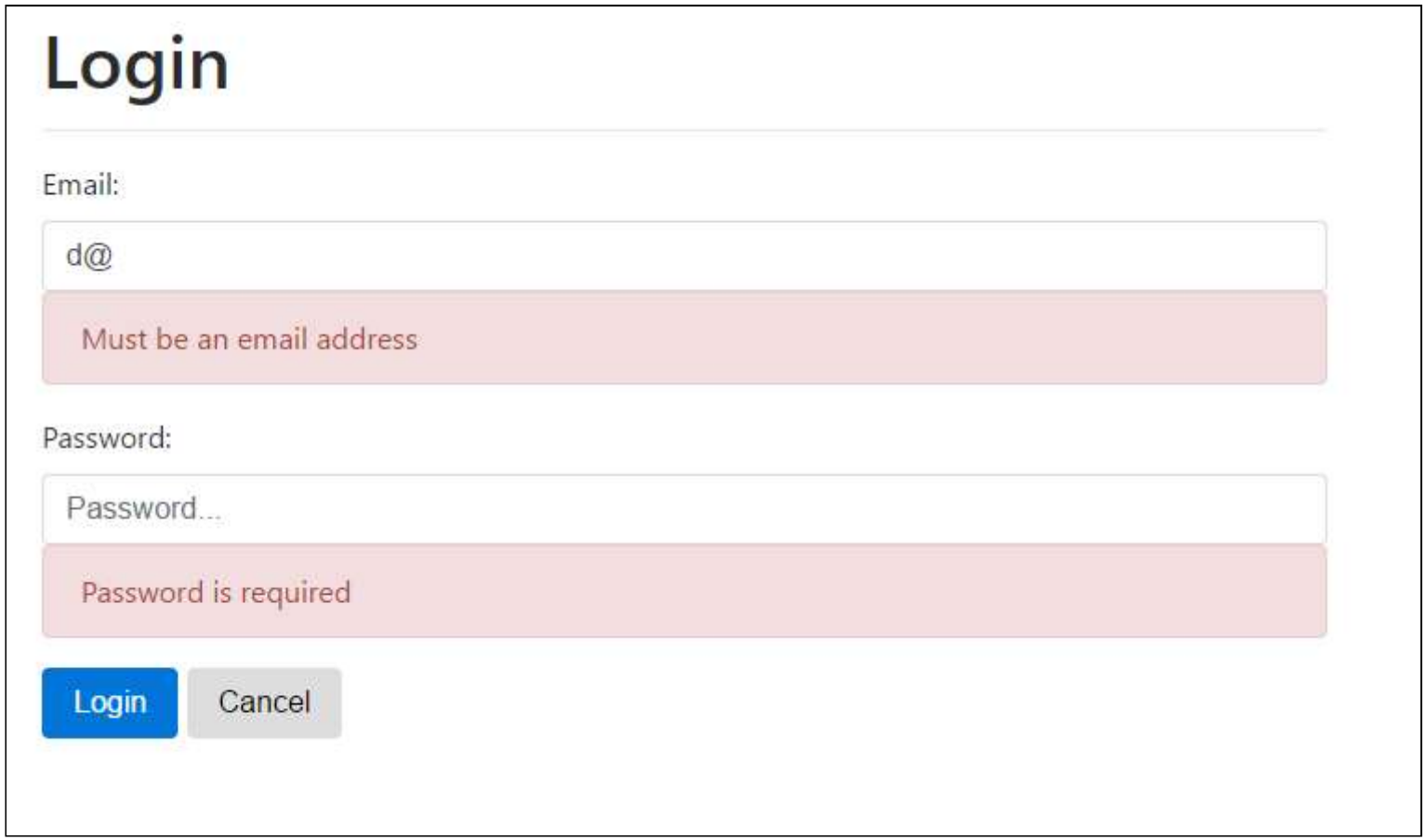
Next we need to add a message like we did for the required validator for when the email validation triggers on an invalid email. Add the following code below the required message but within the `<div>` that checks if there are errors.

Copy

```
<div [hidden]="email.errors.required || !email.errors.email">
    Must be an email address
</div>
```

Now we are ready to test the email validation.

- The validation will trigger if you input text into the email field that is not a valid email address.



# 5.10 Disable Login Button

### Exercise: Disable Login Button Until Valid

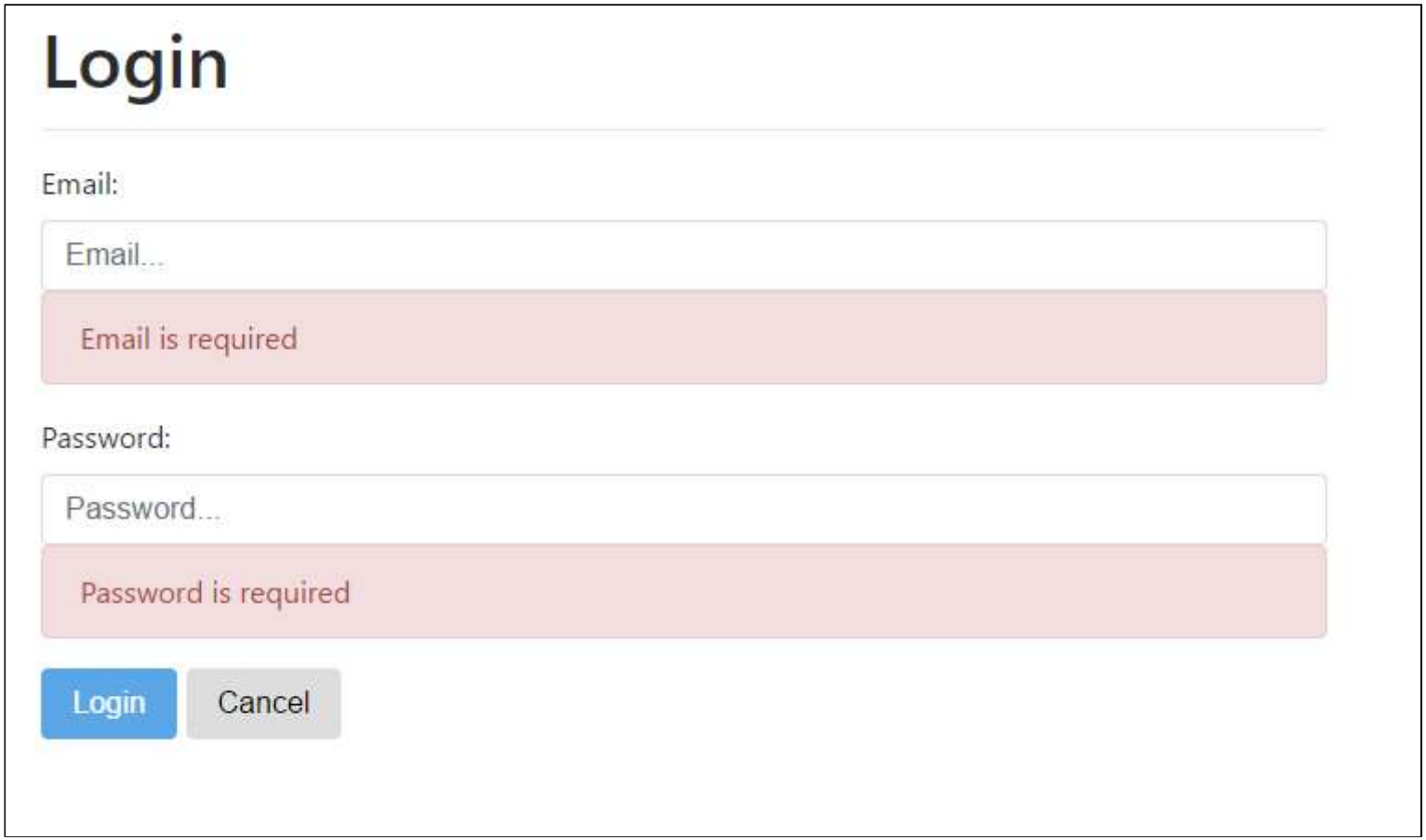
The last thing we want to do is to disable the login button until the form is valid.

1. Find the submit button and add the disabled attribute that makes sure that the loginForm is valid before enabling the button

Copy

```
[disabled]="loginForm.invalid"
```

2. Now when either field is invalid the login button will be disabled and a lighter share of blue




## 5.11 Create Create Account Component

Up to this point, we have only been able to login to an existing account. Now we are going to create the signup page.

Creating the signup component is just like the rest of the component that we have created. We will have an html, scss, spec, and ts file. We will have a form that calls to a function in the component that calls to a service to process the data.

Since the signup component creation is exactly like the login component we are just going a quick here is the code walk through

## Exercise: AuthService Signup Function

We are first going to create the signup function in the AuthService.

1. Add the following method to allow an account to be created

Copy

```
signup(email: string, password: string) {
 let loginInfo = { "email": email, "password": password };
 return this.http.post("https://dj-sails-todo.azurewebsites.net/user/", loginInfo,
this.options)
   .do((res: Response) => {
     if (res) {
       this.currentUser = <User>res.json();
     }
   })
   .catch(error => {
     console.log('signup error', error)
     return Observable.of(false);
```

```
    });
}
```

REMINDER: Since we have to pass the password to the API in order to create the account and we are communicating over a non-secure channel, make sure you do not use your real passwords.

### Exercise: Create Signup Component

1. Open terminal and run

Copy

```
ng generate component signup
```

### Exercise: Add Signup Route

1. Open the src\app\app-routing.module.ts file

Copy

```
app-routing.module.ts
```

2. Import the signup component

Copy

```
import { SignupComponent } from './signup/signup.component';
```

3. Add a new route to get to the signup page

Copy

```
{ path: 'signup', component: SignupComponent},
```

4. You routes should look like

Copy

```
const routes: Routes = [
    {
        path: '',
        children: [],
    },
    { path: 'login', children: [], component: LoginComponent },
    { path: 'signup', component: SignupComponent},
];
```

## Exercise: Create Signup UI

1. Open the src\app\signup\signup.html file

Copy

```
signup.html
```

2. Replace the contents with the following UI and template based form to allow a user to create an account

Copy

```
 <h1>Sign Up</h1>
 <hr>
 <div>
 <form #signupForm="ngForm" (ngSubmit)="signup(signupForm.value)" autocomplete="off"
novalidate>
     <div class="form-group">
     <label for="userName">Email:</label>
     <input #email="ngModel" (ngModel)="email" name="email" id="email" required email
id="email" type="text" class="form-control" placeholder="Email..." />
     <div *ngIf="email.errors && (email.dirty || email.touched)" class="alert alert-
danger">
         <div [hidden]="!email.errors.required">
             Email is required
         </div>
         <div [hidden]="email.errors.required || !email.errors.email">
```

```
            Must be an email address
        </div>
    </div>
    </div>
    <div class="form-group">
    <label for="password">Password:</label>
    <input #password="ngModel" (ngModel)="password" name="password" id="password" required
minlength="6" id="password" type="password" class="form-control" placeholder="Password..."
/>
    <div *ngIf="password.errors && (password.dirty || password.touched)" class="alert
alert-danger">
        <div [hidden]="!password.errors.required">
            Password is required
        </div>
        <div [hidden]="!password.errors.minlength">
            Password must be at least 6 characters long.
        </div>
    </div>
    </div>

    <span (mouseenter)="mouseoverLogin=true" (mouseleave)="mouseoverLogin=false">
        <button type="submit" [disabled]="signupForm.invalid" class="btn btn-primary">Sign
Up</button>
```

```
    </span>
    <button type="button" (click)="cancel()" class="btn btn-default">Cancel</button>
    <span><a [routerLink]="['/login']">login to existing account</a></span>

</form>
<br />
<div *ngIf="loginInvalid" class="alert alert-danger">Invalid Login Info</div>
</div>
```

3. Open the src\app\signup\signup.component.ts file

Copy

```
signup.component.ts
```

4. Replace the contents of the file with the following

Copy

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from '../shared/services/auth.service';
import { Router } from '@angular/router';

@Component({
    selector: 'app-signup',
```

```
    templateUrl: './signup.component.html',
    styleUrls: ['./signup.component.scss']
})
export class SignupComponent implements OnInit {
    loginInvalid: boolean = false;
    constructor(private authService: AuthService, private router: Router) { }

    ngOnInit() {
    }

    signup(formValues) {
        this.authService.signup(formValues.email, formValues.password)
        .subscribe(result => {
            if (!result) {
            this.loginInvalid = true;
            } else {
            this.router.navigate(['/']);
            }
        });
    }
}
```

## Exercise: Add Link Between Login and Signup

1. Open the login.component.html file

Copy

```
login.component.html
```

2. Next to the cancel button add the following HTML to give a link to the create page

Copy

```
<span><a [routerLink]="['/signup']">create account</a></span>
```

3. If you go to http://localhost:4200/signup you should now to be able to signup and navigate between the signup and login pages. Once signed up, you will be redirected to the home page and shown the todo items.

# 6. Reactive Forms

## 6.1 Overview

In the previous chapter, we took a look at template based forms. Templates are great for very simple forms that you do not want to unit test at all. However, if you want to unit test your forms you will need to reuse reactive based forms.

Reactive forms allow you to define the form fields and validation in the component instead of the template. You can easily test the form fields and validation logic. You can also dynamically build the form and validation in the component.

We are going to build the form to enter our Todo items using Reactive forms.

## 6.2 Goals

- Understand reactive forms
- Create a reactive form
- Implement input validation
- Submit form values to a service

# 6.3 Create Todo Component

### Exercise: Create Todo Component

1. Within VS Code, open up the integrated terminal (ctrl+`) or view menu and then "Integrated Terminal"

2. Run the ng generate command below to create the todo component

Copy

```
ng generate component todo
```

3. The generate command will create 4 files:


   - scss (styles)

- html (view)
- spec file (test)
- component (controller)
- add reference to the component in the app.module.ts file.

# 6.4 Add Route

### Exercise: Todo Routing

Before we can view our todo component, we need to tell Angular how to route to the page

1. Open the src\app\app-routing.module.ts file

Copy

```
app-routing.module.ts
```

2. Add the Import statement for the todo component on line 3

Copy

```
import { TodoComponent } from './todo/todo.component';
```

3. We want to make the Todo compoment the home page. We can do this by adding a compoment field to the `path: ''` route

Copy

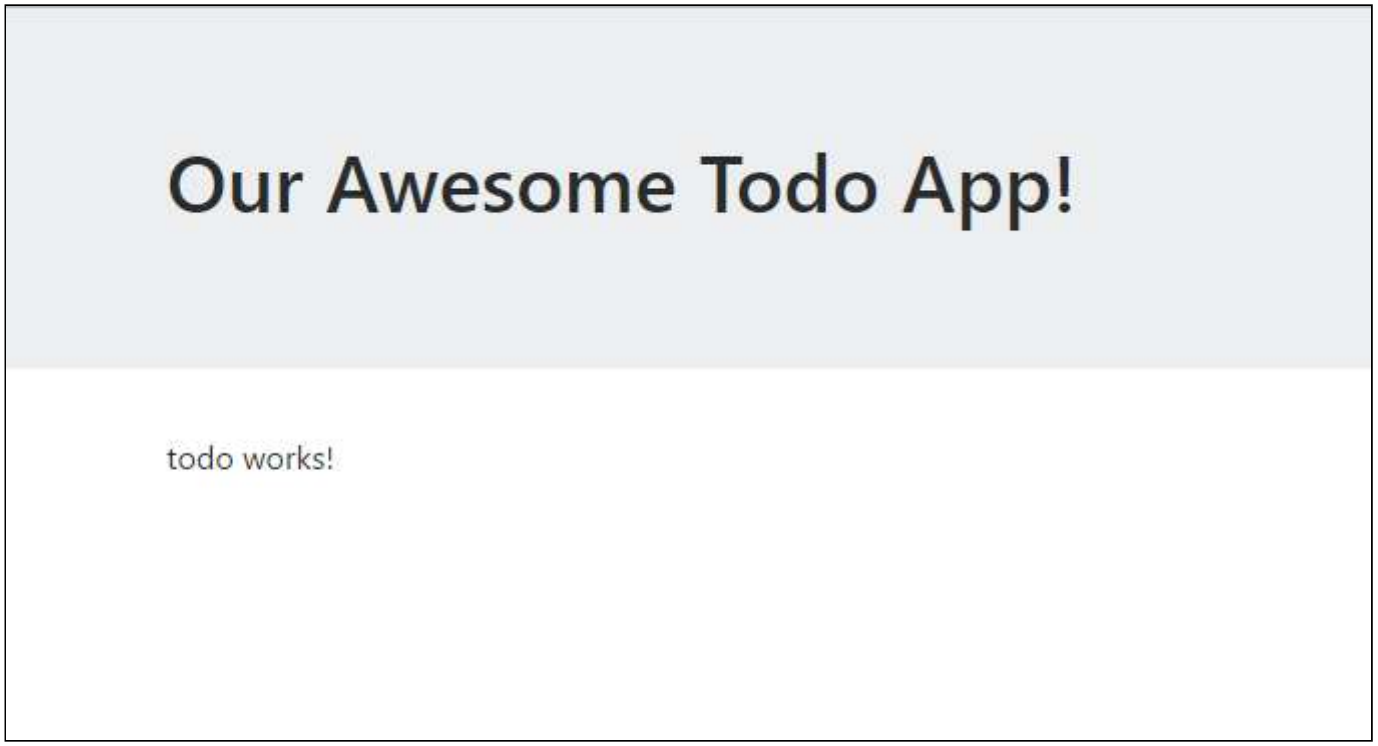
```
{
    path: '',
    children: [],
    component: TodoComponent
}
```

4. You routes should look like

Copy

```
const routes: Routes = [
    {
        path: '',
        children: [],
        component: TodoComponent
    },
    { path: 'login', children: [], component: LoginComponent },
    { path: 'signup', component: SignupComponent},
];
```

5. The todo page should display when you to the home, http://http://localhost:4200/




**Note:** When you navigate to the home page, the TodoCompoment is loaded into the `<router-outlet>` `</router-outlet>` in the html in src\app\app.component.html. The router-outlet tag is how Angular knows where to put the rendered content for the route.

## 6.5 Create Form

Next we are going to create the form without any validation logic at all. Our form will have 1 input field for the todo item and an add button.

## Exercise: Import ReactiveFormsModule

With reactive forms, we are going to setup the form in your components TypeScript file.

We need to import the ReactiveFormsModule into our AppModule before we can use it.

1. Open the src\app\app.module.ts

Copy

```
app.module.ts
```

2. Add the ReactiveFormsModule to the existing @angular/forms import so that it looks like

Copy

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
```

3. In the @NgModule imports you need to add ReactiveFormsModule to the list

Copy

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

```
    AppRoutingModule
],
```

## Exercise: Form setup

Now we are ready to add all of the functionality to the component for the UI to interface with. For the todo component it is the src\app\todo\todo.component.ts file.  We need to add the Form.

1. Open the todo\todo.component.ts file

Copy

```
todo.component.ts
```

2. When creating Reactive Forms, you will need to import the FormGroup and FormBuilder modules from @angular/forms

Copy

```
import { FormGroup, FormBuilder } from '@angular/forms';
```

3. In order to use the FormBuilder we need to inject it into the TodoComponent's constructor

Copy

```
constructor(private formBuilder: FormBuilder) {}
```

4. Before creating our form, we need to define the addForm variable to hold the configuration. Inside of the TodoComponent class above the constructor, add the following variable

Copy

```
addForm: FormGroup;
```

5. To configure the form we will use the ngOnInit lifecycle event. The ngOnInit lifecycle event will run before the component has rendered. Inside the ngOnit function we need to tell Angular that the addForm is a formBuilder group with 1 field called item with an empty default value.

Copy

```
this.addForm = this.formBuilder.group({
    'item': ''
});
```

## Exercise: Creating the Todo UI

We are now ready to create our UI.

1. Open src\app\todo\todo.component.html

Copy

```
todo.component.html
```

2. Replace the existing html with:

Copy

```
 <div class="container">
     <div class="page-header">
         <h1 align="center">Todo List</h1>
     </div>
     <form role="form" [formGroup]="addForm" class="text-center">
         <div class="form-group row">
             <div class="col-sm-10">
                 <input type="text" class="form-control form-control-lg"
formControlName="item" placeholder="Todo!">
             </div>
             <div class="col-sm-2">
                 <button type="submit" [disabled]="addForm.invalid" class="btn btn-primary
btn-lg btn-block">Add</button>
             </div>
         </div>
     </form>
     <div class="container">
         <h3>Form Status Info</h3>
         <table class="table table-striped table-bordered">
         <thead>
```

```
            <tr>
                <th>Status</th>
                <th>Form</th>
                <th>Item Field</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>Dirty</td>
                <td>{{ addForm.dirty }}</td>
                <td>{{ addForm.get('item').dirty }}</td>
            </tr>
            <tr>
                <td>Touched</td>
                <td>{{ addForm.touched }}</td>
                <td>{{ addForm.get('item').touched }}</td>
            </tr>
            <tr>
                <td>Valid</td>
                <td>{{ addForm.valid }}</td>
                <td>{{ addForm.get('item').valid }}</td>
            </tr>
            <tr>
```

```
                <td>Errors</td>
                <td>N/A</td>
                <td><pre>{{ addForm.get('item').errors | json }}</pre></td>
            </tr>
            <tr>
                <td>Form Field Values</td>
                <td colspan="2"><pre>{{ addForm.value | json }}</pre></td>
            </tr>
        </tbody>
        </table>
    </div>
</div>
```

- The [formGroup] binds the addForm in the todo.component.ts to the form.
- On the input field, the formControlName corresponds to the `item` field in the `addForm` FormGroup
- The form is using Bootstrap for the styling with the form-group, form-control, btn, and btn* css classes.
- The form status info section, is debugging information for the form and the item field for us so that we can see the current state, validation errors, and the form field values.
    - As you type into the item field, the form status info value section will update
- The add button is set as disabled until validation passes.

# 6.6 Add Submit Method

In order to submit the form we need to add an `(ngSubmit)=""`

### Exercise: Add ngSubmit

1. In the src\app\todo\todo.component.html file and on the `<form>` tag add the `(ngSubmit)=""` attribute

Copy

```
(ngSubmit)="save()"
```

2. Open the src\app\todo\todo.component.ts file

Copy

```
todo.component.ts
```

3. Add a function called save that returns a void. For now inside of the save function we are going to console log the addForm field values.

Copy

```
save() : void {
    console.log('form values: ', this.addForm.value);
}
```

If you completed the previous chapter on Template Forms, you will notice that we did not setup any ngModel tags or pass in the form to the save method. With Reactive Forms, the formGroup provides the data binding for us.

4. If you do not already have the Chrome Developer Tools open, open them up and click on the console tab

5. Enter text into the input box on the home page http://localhost:4200 and click submit.

6. You should see the form values output to the Chrome Developer Tools console

## 6.7 Validators

Right now we do not have any validation being done on the form. To setup validation, we need to modify our formBuilder item field setup.

### Exercise: Add Validators for required and min length

1. Open the src\app\todo\todo.component.ts file

Copy

```
todo.component.ts
```

2. To use the Angular form validators, we need to add the Validators module to the @angular/forms import statement like so (note that order of the modules in not important)

Copy

```
import { FormGroup, FormBuilder, Validators } from '@angular/forms';
```

3. In the ngOnInit function we need add the Validators required and minLength to the item field that we defined earlier. In order to add the validators, we need to turn the item field value into an array with the 1st position being the default value and the 2nd position as an array of validators. For the minLength, we are going to require at least 3 characters.

Copy

```
'item': ['', [Validators.required, Validators.minLength(3)]]
```

Now that we have validators setup, we need to output an error message to user when the validation fails. For now, we are going to do it in the html but later we will make it more generic and add it to the component file instead.

1. Open the todo.component.html file

Copy

```
todo.component.html
```

2. After the closing form tag in the todo.component.html file add the following

Copy

```
 <div class="alert alert-danger" *ngIf="addForm.get('item').errors &&
(addForm.get('item').dirty || addForm.get('item').touched)">
     <span *ngIf="addForm.get('item').errors.required">
         Item is required
     </span>
     <span *ngIf="addForm.get('item').errors.minlength">
         Item must be at least 3 characters
     </span>
 </div>
```

- By looking at the dirty and touched status, we can ensure that we don't display the error message before the user has had a chance to click on the input field.
- By looking at the errors status, we can ensure we only show the messages when there are errors
- For each of the messages, we can look at the individual validators to see which one failed and only display that message.

3. If you go to http://localhost:4200/ click on the field and then click off it the required validator will fire. If you enter less than 3 characters the minLength validator will fire.

## 6.8 Easier Validation Messages

Having the validation messages in the html template gets old really fast. It is a lot of code to maintain. With the Reactive Forms, we can create a generic error checker for the whole form and set a value for each of the form fields.

### Exercise: Watching for Changes

We are going to create a function that will be called each time the form values change. Within the function we will loop through all of the fields and check if they are dirty and not valid. Then we will look up the validation message to use for the form field and validator that failed.

1. Open the src\app\todo\todo.component.ts file

Copy

```
todo.component.ts
```

2. We need to create a variable to hold the error message for each of the form fields. In this case we only have 1 form field called item. We are going to call the variable formErrors. The default value for the error message is a blank string. This list will also be used to determine which form fields to inspect for validation errors.

Copy

```
formErrors = {
    'item': ''
};
```

3. Next we need to create a variable to hold the validation messages for each of the form fields. The name of the validation message must match the name of the validation.

Copy

```
validationMessages = {
    'item': {
    'required': 'Item is required.',
    'minlength': 'Item must be at least 3 characters'
    }
};
```

4. Now that we the look up for the validation error messages and a place to store the form field error we are ready to create our generic function to determine the actual error message.

Copy

```
onValueChanged(data?: any) {
    if (!this.addForm) { return }
    const form = this.addForm;

    for (const field in this.formErrors) {
        // clear previous error message (if any)
        this.formErrors[field] = '';
        const control = form.get(field);

        if (control && control.dirty && !control.valid) {
            const messages = this.validationMessages[field];
            for (const key in control.errors) {
```

```
                this.formErrors[field] += messages[key] + ' ';
            }
        }
    }
}
```

- The 1st thing the method does is make sure that our addForm actually has a value.
- Then we loop through the formError variable, get the field and check if the form field is invalid
- If the form field is invalid, then we look up the validation message for the form field and validator that failed and set the formError for that field.

5. Next we need to subscribe to the addForm valueChanges event and call the onValueChanged function we just created. We are going to setup the subscribe in the ngOnInit function

Copy

```
this.addForm.valueChanges.subscribe(data => this.onValueChanged(data));
```

6. The last thing we are going to is call the onValueChanged function in the ngOnInit function to reset any formErrors back to blank

Copy

```
this.onValueChanged();
```

## Exercise: Show Error Message in UI

The last thing we need to do is up the UI to display the form error messages.

1. Open the todo.component.html file

2. Replace the validation messages that you added in the previous section with the one below.

Copy

```
<div *ngIf="formErrors.item" class="alert alert-danger">
    {{ formErrors.item }}
</div>
```

3. If you navigate to http://localhost:4200, click on the item form field and enter 1 character it will trigger the minLength validator and will show the minLength validation message. If you then blank out the field you will see the required message.

## Exercise: Add Border On Invalid

You can also add a border around the Bootstrap form-group for the item form field by adding the has-danger css class when the formErrors.item has a value.

1. Open the todo.component.html file

Copy

```
todo.component.html
```

2. To the form-group div tag, add an `[ngClass]` attribute that checks that formErrors.item has a value and if so then adds the has-danger css to the div tag

Copy

```
[ngClass]="{'has-danger': formErrors.item}"
```

## 6.9 Wait Before Validation Messages

You might have notice after implementing the previous logic to check the field values in the TypeScript file, that the validation errors are immediately shown which can be annoying to users while they type. Instead it would be better to wait for a given amount of time after the last keystroke before checking. This is called debounce.

Angular makes it very easy to implement what they call debounce to wait for the user to stop typing before running validation on our item input field.

**Exercise: Implement Debounce**

1. Open todo.component.ts file

todo.component.ts

Copy

2. Import the rxjs debounceTime

Copy

```
import 'rxjs/add/operator/debounceTime';
```

3. On the line that you added the `itemControl.valueChanges.subscribe` add the `debounceTime` statement between valueChanges and subscribe like so

Copy

```
this.addForm.valueChanges.debounceTime(1000).subscribe(data => this.onValueChanged(data));
```

4. Now if you test the UI at http://localhost:4200, it will wait 1 second after the last keystroke before checking the input field validation. You can change the time it waits by increasing or decreasing the value that is passed into the debounceTime function.

## 6.10 Saving Todo Items

We are at the point, where we are ready to create a service to save the todo item. Right now, we are just output the add form values to the console.

### Exercise: Class to Hold Todo item

Since TypeScript is a strongly typed language it is best practice to create a class to hold our Todo items. This way we can get the type support for the different fields.

1. Within VS Code, open up the integrated terminal (ctrl+`) or view menu and then "Integrated Terminal"

2. Run the ng generate command below to create the todo component

Copy

```
ng generate class shared/classes/Todo
```

3. This will create 1 files:


1. Open src/app/shared/classes/todo.ts

Copy

```
todo.ts
```

2. Add the following 4 fields to the Todo class to hold the information about our todo item

| Field | Data Type | Purpose |
| --- | --- | --- |
| id | string | unique identifier to the item |
| item | string | todo item text |
| createdAt | Date | date added |
| updatedAt | Date | date last updated |
| completed | boolean | completion state |
| user | string | id of the user that created the todo item |

Copy

```
id: string;
item: string;
completed: boolean;
createdAt: Date;
updatedAt: Date;
user: string;
```

3. To make it easier to create new todo Items and implement unit testing, we are going to add a constructor to initialize the fields. Item will required while id, completed and createdAt will be optional. The optional fields must be after all of the required fields.

Copy

```
constructor(
    item: string,
    id?: string,
    completed?: boolean,
    createdAt?: Date,
    updatedAt?: Date) {
    id = id ? id : '';
    this.item = item;
    this.completed = completed ? completed: false;
    this.createdAt = createdAt ? createdAt: new Date();
    this.updatedAt = updatedAt ? updatedAt: new Date();
}
```

### Exercise: Create Todo Service

1. Within VS Code, open up the integrated terminal (ctrl+`) or view menu and then "Integrated Terminal"

2. Run the ng generate command below to create the todo component

Copy

```
ng generate service shared/services/Todo --module App
```

3. This will create 2 files and update the app.module to add the TodoService into the providers list

```
installing service
  create src/app/shared/services/todo.service.spec.ts
  create src/app/shared/services/todo.service.ts
  WARNING Service is generated but not provided, it must be provided to be used
```

   - todo.service.ts
   - todo.service.spec.ts

### Exercise: Add Todo Service Save

Now that we have the Todo service file created, we need to add our save method that calls our json-server api server and then update the Todo component to call the service.

1. Open src\app\shared\services\todo.service.ts

Copy

```
todo.service.ts
```

2. Import the following so that we can make our HTTP calls and get a response back.

```
import { Http, Response, RequestOptions } from '@angular/http';
import { Observable } from 'rxjs/Rx';
```

Copy

3. In order to use the HTTP module, we need to inject it into our constructor

```
constructor(private http: Http) { }
```

Copy

4. For the API that we are using (SailsJS based), it requires that we set the HTTP option to allow credentials so that the session cookie can be passed back and forth, else it will always think you haven't logged in. Add this varaible to the TodoService class.

```
private options = new RequestOptions({ withCredentials: true });
```

Copy

You will need to pass in this.options as the last parameter for all of our http calls.

5. Before we create our save method we need to import the Todo class so that our data is typed when we pass it from the service the component.

```
import { Todo } from '../classes/todo';
```

Copy

6. Next we need to create our save function that will call our API, pass in our TodoItem, and return back the results to the component.

Copy

```
 save(item: string): Observable<Todo> {
     return this.http.post('https://dj-sails-todo.azurewebsites.net/todo', new Todo(item),
this.options)
     .map((res: Response) => {
         return <Todo>res.json();
     })
     .catch(error => {
         console.log('save error', error)
         return error;
     });
 }
```

### Exercise: Call TodoService Save from TodoCompoment

Now that we have our Todo service save function created, we need to call it from our Todo component so that we can save our data.

1. Open the src\app\todo\todo.component.ts file

todo.component.ts Copy

2. Before we can call the `TodoService.save` function we have to import the TodoService

```
import { TodoService } from '../shared/services/todo.service';
```
Copy

3. Now that we have the TodoService import we need to inject it into the constructor to make it avaiable to the component.

```
constructor(private formBuilder: FormBuilder, private todoService: TodoService) { }
```
Copy

4. Update the save method with the following code to call the `TodoService.save` function and output the result to the console

```
this.todoService.save(this.addForm.value.item)
.subscribe(result => {
    console.log('save result', result);
},
error => {
    this.errorMessage = <any>error;
});
```
Copy

5. We now need to create the errorMessage variable that is of type string in the Todocomponent class

Copy

```
errorMessage: string;
```

6. Open the todo.component.html file so that we can add the display of the save error message.

Copy

```
todo.component.html
```

7. Now we need to add an alert section to our todo.component.html to display the error message. After the `</form>` tag, add the following code

Copy

```
<div *ngIf="errorMessage" class="alert alert-danger" role="alert">
    <h3>Error Saving</h3>
    {{ errorMessage }}
</div>
```

8. Testing the error message display requires that we temporary set a value for the errorMessage. We are going to do this in the ngOnInit just to verify that the error message will display:

Copy

```
this.errorMessage = 'testing'
```

9. Now if you go to http://localhost:4200 you will see the following display


10. We can remove the temporary value that we set for the errorMessage.

## 6.11 Displaying Items

Now that we have the ability to save our items, we need to be able to display the current list with options to complete or delete a todo item.

### Exercise: Add TodoService Get All

Next we need to add a getAll function to our TodoService that does an http get to our API to get all of the todo items.

1. Open the src\app\shared\services\todo.service.ts file

Copy

```
todo.service.ts
```

2. Add the following function to make an http get call to our Todo API

Copy

```
getAll(): Observable<Array<Todo>>{
let url = "https://dj-sails-todo.azurewebsites.net/todo";
return this.http.get(url, this.options)
  .map((res: Response) => {
    return <Array<Todo>>res.json();
  })
  .catch(error => {
    console.log('get error', error);
    return error;
  });
}
```

### Exercise: Call TodoService Get All from Component

Now that we have the TodoService.getAll function created, we are ready to create the getTodoListAll function that will call the TodoService. We will wire up the getTodoListAll function to be called in ngOnInit so that it will populate the todo list on component load.

1. Open the src\app\todo\todo.component.ts file

Copy

```
todo.component.ts
```

2. Import the todo class

Copy

```
import { Todo } from '../shared/classes/todo';
```

3. Create a variable in the TodoComponent class called todoList that is an array of Todo and intialize to an empty array

Copy

```
todoList: Array<Todo> = [];
```

4. Create the getTodoListAll function that will return void, call the TodoService.getAll function and set a todoList variable at the Todocomponent class level.

Copy

```
getTodoListAll(): void {
this.todoService.getAll()
  .subscribe(
  data => {
    this.todoList = data;
```

```
    },
    error => {
      this.errorMessage = <any>error;
    }
    );
  }
```

5. Now we are ready to call the getTodoListAll function in ngOnInit

Copy

```
this.getTodoListAll();
```

## Exercise: Html to Display Items

1. Open the src\app\todo\todo.component.html

Copy

```
todo.component.html
```

2. Since we have our form working, remove the form status table.

3. After the error message alert, add the following html to display the list of todo items

```
<div class="row" *ngFor="let todoItem of todoList">
  <div class="col-12 done-{{todoItem.completed}}">
    {{todoItem.item}} <small>created: {{todoItem.createdAt | date:'short'}}</small>
  </div>
</div>
```

Copy

- the *ngFor will loop through all of the items in the todoList variable that we will be creating next.
- For the date, we are using the built-in date pipe to convert it to a short date that strips out the time part of the date
- We are also setup to have a different style when an item is completed. We will add the styling in a bit.

## Exercise: Updating Todo list on save

Now that we have the Todo list being stored in the todoList variable, when we save a new todo item, we can add it to the todoList array and the todo list will automatically update with the change.

1. Open todo.component.ts

Copy

```
todo.component.ts
```

2. In the todo.component.ts file, we need to update the save function to push the save result into the todoList array. We need to add this code into the subscribe of the `TodoService.save` call.

Copy

```
this.todoList.push(result);
```

3. When you add a new todo item, the list will now update itself.

### Exercise: Complete Todo Item

Right now the todo list is just a read only view but we need to be able to complete the todo items. We need to add an icon to the todo list that will toggle the completed status and save the new todo item state to the API.

1. Open the src\app\shared\services\todo.service.ts file

Copy

```
todo.service.ts
```

2. We are going to create an update method that will take in a Todo item and make an http put call to our API to update the one record with the new completion state.

Copy

```
updateTodo(todo: Todo): Observable<Todo> {
    let url = `https://dj-sails-todo.azurewebsites.net/todo/${todo.id}`;
```

```
    return this.http.put(url, todo, this.options)
    .map((res: Response) => <Todo>res.json())
    .catch(error => {
        console.log('update error', error);
        return error;
    });
}
```

- For the url we are using string interpolation to create the url. This is done with the `` tags and the ${}

Now we need to call the updateTodo service function in our component.

1. Open the todo.component.ts file

Copy

```
todo.component.ts
```

2. Create the completeTodo method that the UI will call

Copy

```
completeTodo(todo: Todo): void {
    todo.completed = !todo.completed;
    this.todoService.updateTodo(todo)
```

```
    .subscribe(
        data => {
            // do nothing
        },
        error => {
            todo.completed = !todo.completed;
            this.errorMessage = <any>error;
            console.log('complete error', this.errorMessage);
        });
}
```

- For now we are not going to do anything with the returned result. In the future you could call a sort function or update an open todo item counter.

The last thing we need to do it do update the UI to have a checkbox icon that will be clicked on to toggle the completion state.

1. Open the todo.component.html file

Copy

```
todo.component.html
```

2. Inside the ngFor loop, above the existing div that is displaying the individual item, add the following icon that uses the Font Awesome library for the icon and is set to take up 1 column of space

```
   <div class="col-1" (click)="completeTodo(todoItem)"><i [className]="todoItem.completed
== true ? 'fa fa-check-square-o' : 'fa fa-square-o'"></i></div>
```

Copy

- We are passing in the todo item that we are wanting to update to the completedTodo function. This will pass in the whole object so we have access to all of the fields.
- We are updating the icon used based on the completed field state. If completed we are using fa-check-square-o. If not completed, we are using fa-square-o

3. With Bootstrap it is a 12 column grid, so we need to reduce the size of the existing div from col-12 to col-11

## Exercise: Delete Todo Item

In addition to being able to complete a todo item, we also need to be able to delete one. We need to add an icon to the todo list that will call a delete function in the component and delete the todo item from our database.

1. Open the src\app\shared\services\todo.service.ts file

2. We are going to create delete method

```
deleteTodo(todo: Todo): Observable<Response> {
    let url = `https://dj-sails-todo.azurewebsites.net/todo/${todo.id}`;
```

Copy

```
    return this.http.delete(url, this.options)
    .catch(error => {
        console.log('delete error', error);
        return error;
    });
}
```

- We are passing in the todo item that we are wanting to update to the completedTodo function. This will pass in the whole object so we have access to all of the fields.
- We are not doing any kind of mapping of the return results since there is none. It is either successful or not.

Next we need to create the deleteTodo item function in the component that will call the TodoService.delete function

1. Open the todo.component.ts file

2. Create the deleteTodo function that takes in a todo item and calls the TodoService.delete function

Copy

```
deleteTodo(todo: Todo): void {
    this.todoService.deleteTodo(todo)
    .subscribe(
    data => {
        let index = this.todoList.indexOf(todo);
        this.todoList.splice(index, 1);
```

```
        },
        error => {
            todo.completed = !todo.completed;
            this.errorMessage = <any>error;
            console.log('complete error', this.errorMessage);
        });
    }
```

- In the results of the deleteTodo service, we are going to remove the todo item from the displayed list since it no longer exist. We could have also call the TodoService.getAll function but since we already have all of the items and the items are specific to a single user, there is no need to make the extra database call.

The last thing that we need to do is to add the delete icon to the todo list.

1. Open the todo.component.html file

2. Add a div after the div displays the todo item text but still inside of the ngFor div. This new div will hold the delete icon which we will use the fa-trash icon and when clicked it will call the deleteTodo function. The icon is going to take up 1 column of space in the grid.

Copy

```
<div class="col-1" (click)="deleteTodo(todoItem)"><i class="fa fa-trash"></i></div>
```

3. Since the Bootstrap grid is 12 columns wide, we need to reduce the text div from col-11 to col-10.

The reason that we used the Bootstrap grid is so that everything wrapped correctly with longer todo items and when the screen was smaller. The Bootstrap grid provides this functionality automatically for you.

The html for the display of the Todo list should look like the following:

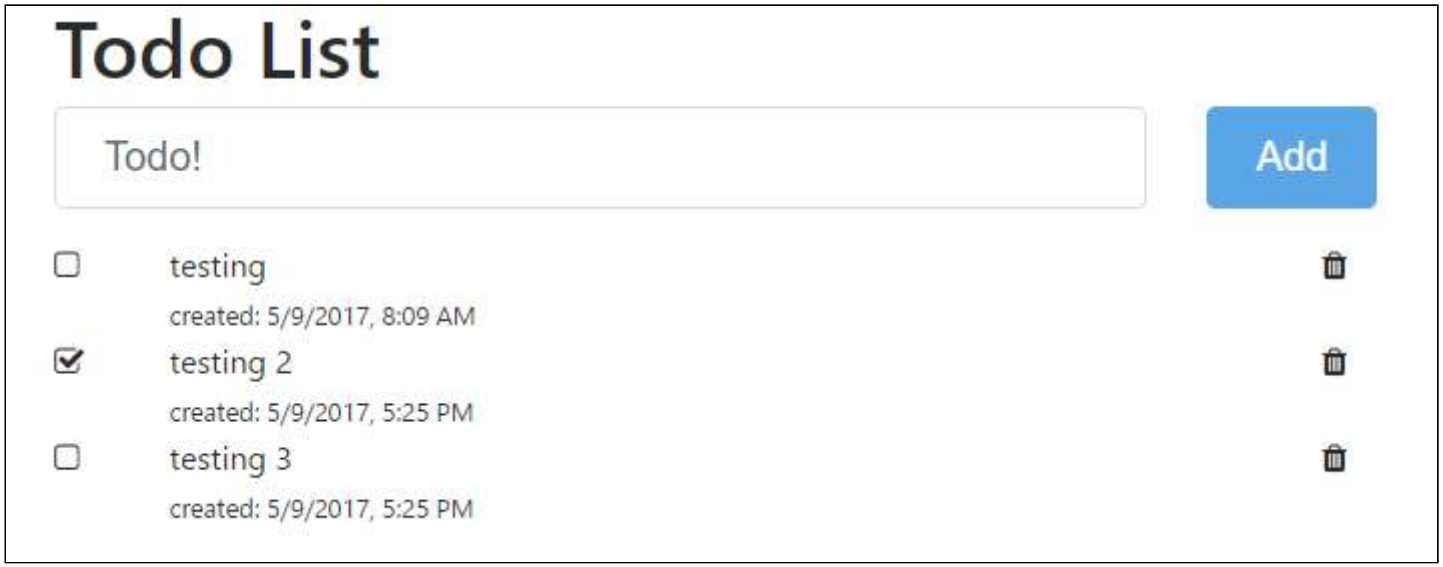
Copy

```
<div class="row todo" *ngFor="let todoItem of todoList">
    <div class="col-1" (click)="completeTodo(todoItem)"><i [className]="todoItem.completed ==
true ? 'fa fa-check-square-o' : 'fa fa-square-o'"></i></div>

    <div class="col-10 done-{{todoItem.completed}}">{{todoItem.item}} <br /><small>created:
{{todoItem.createdAt | date:'short'}}</small></div>

    <div class="col-1" (click)="deleteTodo(todoItem)"><i class="fa fa-trash"></i></div>
</div>
```

## 6.12 Adding Style

### Exercise: Making the Todo list look nicer

Right now the UI looks decent but with a few tweaks it could look much better.




If we added some padding around each row, a bottom border, made the date smaller and gray, increased the size of each icon and made the completed items gray with a strike-through, the UI would pop.

The first thing we need to do is add in our styles to the Todo component. Since these styles are strictly for the Todo component we are going to add them into the todo.component.scss instead of the app's style.scss file.

1. Open the src\app\todo\todo.component.scss

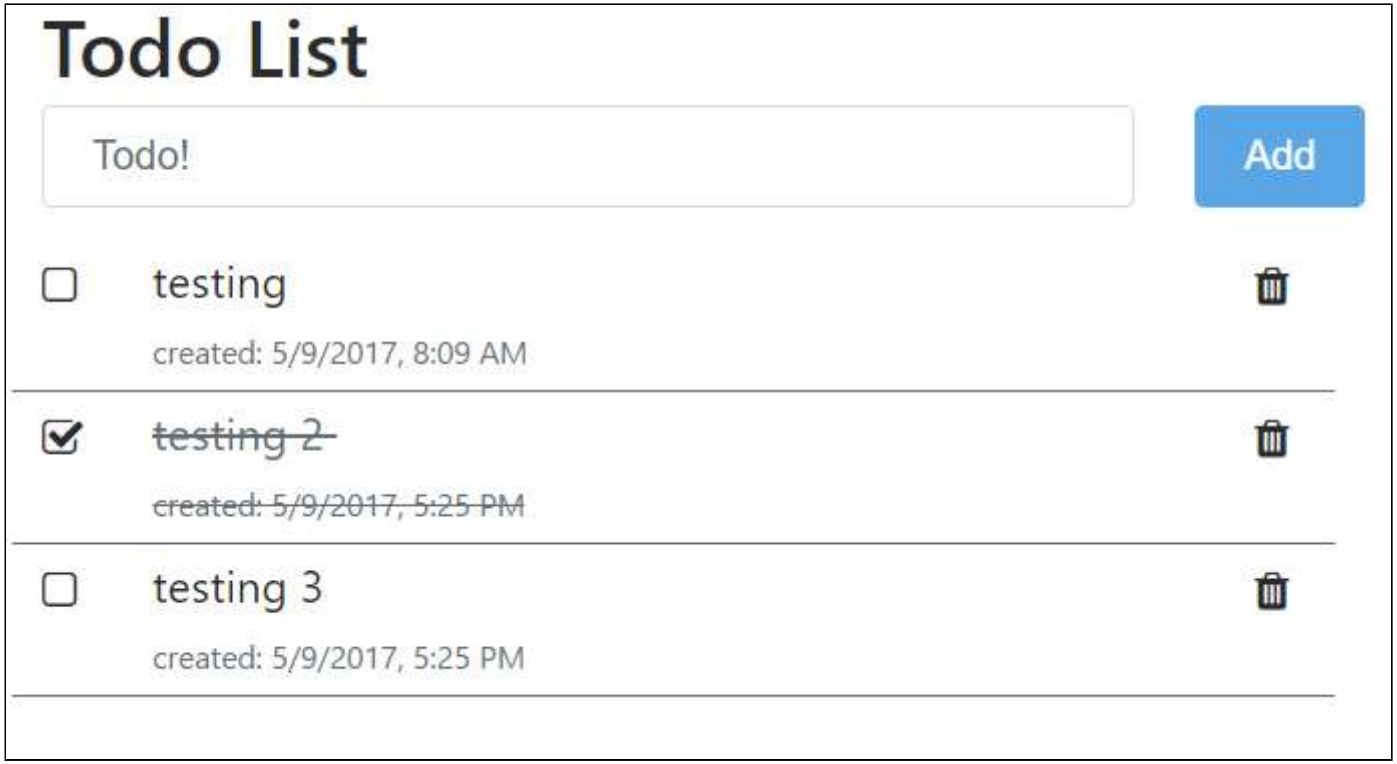
2. Add the following contents to the file. To ensure we are following our branding, we are importing our scss color variables.

Copy

```
@import "../../assets/bootstrap/variables";
div.todo {
    width: 100%;
```

```
    padding-bottom: .2em;
    padding-top: .2em;
    border-bottom: 1px solid $gray-light;
    font-size: 1.4em;

    small {
        font-size: .7em;
        color: $gray-light;
    }

    i {
        width: 40px;
        padding-right: 10px;
        vertical-align: middle
    }

    .done-true {
        text-decoration: line-through;
        color: $gray-light;
    }
}
```

3. Now if you view the UI it should look like below.




## 6.13 Review

We did a lot in this chapter implementing our Reactive form

1. Created a form using the FormBuilder
2. Added validation to the FormBuilder
3. Created a value change observable to set the field validation message

4. Added debounce to wait until the user is done typing before checking the field validation
5. Show the list of todo list with icons to complete and delete todo items

6. You used several Angular directives to implement functionality in the UI:

    - *ngIf - replacement for ng-if. Only show section if condition is true
    - *ngFor - replacement for ng-repeat. Loop through a list and do something
    - [(ngModel)] - two-way data binding
    - (click) - binds to the click event
    - [className] - replacement for ngClass and set the css class for the element
    - (ngModelChange) - runs method when the [(ngModel)] value changes
    - [hidden] - hides the element when condition is true
    - (ngSubmit) - submits a form
    - [formGroup] - used for reactive forms. basically dynamic forms that you can control in the controller
    - [disabled] - set the element to disabled when condition is true

# 7. Environment Configuration

## 7.1 Overview

One of the requirements that you commonly have it to be able to change configurations for your application based on the environment it is running in. A common configuration is to change the API url between development and production. In order to change the configurations for different environments, Angular uses configurations files that are stored in the src\environments folder.

## 7.2 Goals

- Learn how to change configurations per environment
- Implement environment configurations

## 7.3 Default Configuration

The src\environments\environment.ts file is the default configurations if no environment is specified when running `ng serve`

In the environment.ts we need to add the environmentName and apiBaseUrl values. The apiBaseUrl is how to get to your service layer, if you have one. I have created an API for us to use at https://dj-sails-todo.azurewebsites.net

### Exercise: Setup Default Configuration

1. Open the src\environments\environment.ts file

2. Add the environmentName and apiBaseUrl values.

Copy

```
export const environment = {
    production: false,
    environmentName: 'Development',
    apiBaseUrl: 'https://dj-sails-todo.azurewebsites.net'
};
```

3. Go to the terminal that is running the `ng serve` command and do a ctrl+c to stop it.

4. Run the `ng serve` command again.

Copy

```
ng serve
```

5. Everything should still start as normal. You will not see any changes at this point since nothing is using those settings. We will use them in a bit in our footer and our services.

## 7.4 Local Development

The other environment that is typically created is for local development on your machine. For this tutorial the environment.ts and local.ts have the same values but once you go from development to production they will differ.

### Exercise: Create Local Configuration

1. Create the file src\environments\environment.local.ts file

2. Add the following to the environment.local.ts file

Copy

```
export const environment = {
production: false,
environmentName: 'Local',
apiBaseUrl: 'https://dj-sails-todo.azurewebsites.net'
};
```

3. Open the .angular-cli.json that is in the root of the project.

4. Find the apps\environments section and add the local configuration to the at the top of the list of environments

Copy

```
"local": "environments/environment.local.ts",
```

5. If you want to run the local configuration we need to pass in the environment command line argument. The environment argument value is the name of the environment name in the angular-cli.json file. Note that you can only have 1 ng serve running at a time.

Copy

```
ng serve -e local
```

6. If the Angular compile was successful, you will now be using the local environment configuration. You will not see any changes at this point since nothing is using those settings. We will use them in the next exercise in our footer.

## Exercise: Add Environment Name to Footer

1. Open the src\app\shared\footer\footer.component.ts file and import the environment into the file on line 2 right below the Angular core import

Copy

```
import { environment } from '../../../environments/environment';
```

2. Inside the FooterComponent class we to add a variable to capture the environment name that Angular is running in

Copy

```
public env = environment.environmentName;
```

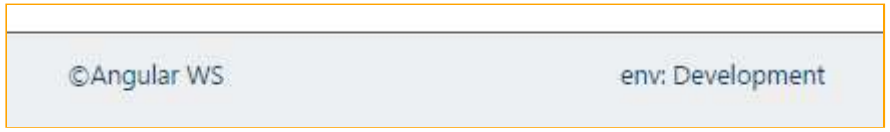
3. Open the src\app\shared\footer\footer.component.html and replace the contents with

Copy

```
<footer>
    <div class="pull-left">
        &copy;Angular WS
    </div>
    <div class="pull-right">
        env: {{ env }}
    </div>
</footer>
```

4. If you view the web page you should see the footer

Notice that the env:Local in the footer is coming from the environment.local.ts file. If you stop `ng serve` and run it without using the -e argument, the env value will change to Development




## Exercise: Updating Services With Environment Url

Now that we have the API url in the environments file, we should update the TodoService and AuthService to get the base url from the environments file instead of having it hard coded. While we are at it, we should also create a variable in the services to hold the base url so that we don't have it all over the place in the services.

1. Open the src\app\shared\services\todo.service.ts file

2. Import the environment file

Copy

```
import { environment } from '../../../environments/environment';
```

3. Create a private variable called url inside the TodoService class that gets the apiBaseUrl from the environment file and append on the /todo

```
  private url: string = `${environment.apiBaseUrl}/todo`;
```

Copy

4. For the save and getAll functions update the hard coded url to use the class level url variable. You will need to change url to this.url in order to access the class level variable.

5. For the updateTodo and deleteTodo we need to replace the hard coded url with the class level url variable. Since we are already using string interpolation to create the string, we need to replace the hard coded value with `${this.url}`

Copy

```
let url = `${this.url}/${todo.id}`;
```

6. Everything should still work to list, insert, update and delete todo items as before but now the url is no longer hard coded and can easily be changed as you move to different environments.

7. Now repeat the same process with the AuthService

# 8. Locking Down Routes

## 8.1 Overview

A very common requirement with applications is to do authentication or authorization check before allowing a user to perform action. Common usage cases are:

- Role checks such as admin before allowing the user into the admin section of the application
- Is the user logged in check before allowing them into the page the pulls or updates data
- Is the user logged out before allowing them to create an account

In this chapter we are going to implement a check for if the user is logged in.

## 8.2 Goals

- Understand how to protect a route
- Understand how to use a guard

# 8.3 Create Guard

## Exercise: Check If User Is Logged In

1. Open terminal and generate the guard

Copy

```
ng generate guard shared/guards/IsLoggedIn
```

```
$ng generate guard shared/guards/IsLoggedIn
installing guard
  create src/app/shared/guards/is-logged-in.guard.spec.ts
  create src/app/shared/guards/is-logged-in.guard.ts
  WARNING Guard is generated but not provided, it must be provided to be used
```

## Exercise: Add Guard to Module

Before you can use the guard, you need to add it to the providers list in the module

1. Open the app.module.ts

2. Import the AdminGuard

```
import { IsLoggedInGuard } from './shared/guards/is-logged-in.guard';
```

Copy

3. Add the AdminGuard to the providers list

Copy

```
providers: [AuthService, TodoService, IsLoggedInGuard],
```

## 8.4 Add Logic to Guard

### Exercise: Add Logic

We need to do add logic in the AuthService to verify that the currentUser variable has a value.

1. Open the src\app\shared\services\auth.service.ts file

2. Add an isAuthenticated function to the AuthService that checks the API to make sure that the user is still logged in

Copy

```
isAuthenticated(): Observable<boolean> {
 return this.http.get(this.url + '/identity', this.options)
   .map((res: Response) => {
```

```
      if (res) {
        return Observable.of(true);
      }

      return Observable.of(false);
    })
    .catch((error: Response) => {
      if (error.status !== 403) {
        console.log('isAuthenticated error', error)
        this.clearUser();
      }

      return Observable.of(false);
    });
}
```

Next we need to add logic to the guard's canActivate function to call the AuthService.isAuthenticated function

1. Open src\app\shared\guards\is-logged-in.guard.ts

2. Import the AuthService

Copy

```
import { AuthService } from '../services/auth.service';
```

3. Import the Router as part of the @angular/route import statement

1. create a constructor and inject the AuthService and Router

Copy

```
constructor(private authService: AuthService, private router: Router) { }
```

2. In the canActivate function we want to add the following logic instead of just returning true. This code will call the AuthService.isAuthenticated function and then return true or false depending on if the user is logged in or not. If the user is not logged in or there is an error validating if they are logged in then we will navigate them to the login route

Copy

```
canActivate(  next: ActivatedRouteSnapshot,
 state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
 let isLoggedIn = new Observable<boolean>(observer => {
 this.authService.isAuthenticated()
   .subscribe((res: boolean) => {
   if (res) {
     observer.next(true);
     observer.complete();
   } else {
     this.router.navigate(['/login']);
```

```
      observer.next(false);
      observer.complete();
    }
    }, error => {
      observer.next(false);
      observer.complete();
    });
  });

  return isLoggedIn;
}
```

## 8.5 Add Guard to Route

In order to use the Guard we need to add it to the route. Each route has a canActivate attribute that takes an array of guards as the 3rd parameter.

**Exercise: Add Guard to Route**

1. Open the app-routing.module.ts file

2. Import the IsLoggedInGuard

Copy

```
import { IsLoggedInGuard} from './shared/guards/is-logged-in.guard'
```

3. To the default route, add the canActivate attribute with the value being an array that contains the IsLoggedInGuard

Copy

```
, canActivate: [IsLoggedInGuard]
```

4. Now stop ng serve and restart it. When you try to go to http://localhost:4200 it will redirect you to the login page since you are not logged in yet.

# 9. Implementing Default Route

## 9.1 Overview

We have not yet implemented a catch all route which will be the route that is used when Angular does not match any other configured routes. .


There are 2 ways to fix this.

1. Make the TodoComponent the default so that it will redirect the user to that page. This is not as nice of a user experience since it does not alert the user that the url they were trying to go to does not exist.

2. Create a "Not Found" component and redirect the user to that component if none of the other routes match. This is the preferred method.

## 9.2 Goals

- Understand how to deal with an unknown route

## 9.3 Add Default Route

**Exercise: Adding Default Route**

**Note:** Not the preferred method.

1. Open the app.routing.ts file

2. Add an additional route to the Routes list with a ** for the path and set the component to TodoComponent. This will tell Angular to use the TodoComponent when it can not determine the route

Copy

```
, { path: '**', component: TodoComponent }
```

Make sure that you make the default route the last route. Any routes below the ** route will be ignored.

# 9.4 Create Not Found Component

### Exercise: Redirecting User to "Not Found" component

**Note:** This is the preferred method.

From a user experience perspective, it is a much experience to redirect the user to a "not found" page instead of redirecting them back to the home page. Create the route not found page is just like we did with the todo component.

1. Run the Angular Cli generate command to create the notFound component

Copy

```
ng generate component notFound
```

2. 4 files are created just like when we created the TodoComponent earlier. You will notice this time though that the Angular Cli put a dash between not and found. It will automatically do that when it encounters an uppercase character.

```
create src\app\not-found\not-found.component.scss
create src\app\not-found\not-found.component.html
create src\app\not-found\not-found.component.spec.ts
create src\app\not-found\not-found.component.ts
```

3. In the src\app\app-routing.module.ts file, add an import statement for the NotFoundComponent

Copy

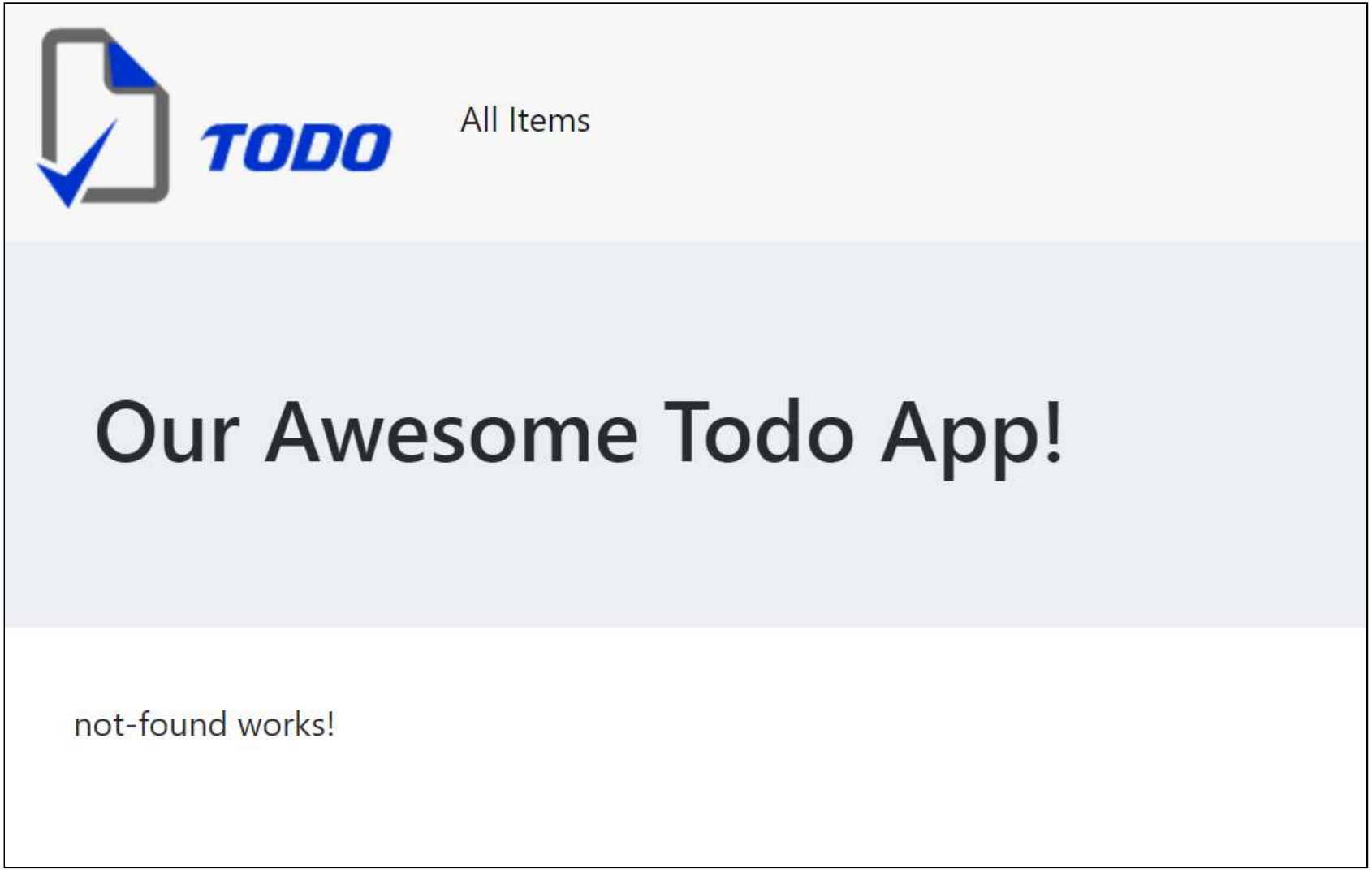
```
import { NotFoundComponent } from './not-found/not-found.component';
```

4. In the app-routing.module.ts file, change the ** route to use the NotFoundComponent

Copy

```
, { path: '**', component: NotFoundComponent }
```

5. If you now try to navigate to http://localhost:4200/unknown you will be shown the NotFoundComponent


TODO
All Items
Our Awesome Todo App!
not-found works!

# 10. Header and footer

## 10.1 Overview

In most web sites we have a header and footer at the top and bottom of the page respectively with our logo, navigation, and important links. The header will contain our logo and navigation menu while the footer will contain our copyright info.

## 10.2 Goals

- Understand how to create a new component
- Understand how to include components inside of other components
- Understand how to utilize Bootstrap

## 10.3 Create Header Component

**Exercise: Creating the header component**

We can leave ng serve running while we make these changes and open up another Integrate Terminal for the commands below.

1. In the VS Code Integrated Terminal, click the + to open a 2nd terminal

2. Run the ng generate command to create the header component

Copy

```
ng generate component shared/header
```

3. The generate command will create 4 files:

```
PS C:\projects\angular-tutorial>  ng generate component shared\header
installing component
  create src\app\shared\header\header.component.scss
  create src\app\shared\header\header.component.html
  create src\app\shared\header\header.component.spec.ts
  create src\app\shared\header\header.component.ts
  update src\app\app.module.ts
```

- scss (styles)
- html (view)
- spec file (test)
- component (controller)
- add reference to the component in the app.module.ts file.

## Exercise: Add the Menu

This is a sample menu that includes a few links and a sub-menu dropdown. Note that none of the links will work since we have not created any of those other pages or routes.

1. Open the src\app\shared\header.component.html file and replace the contents with the following.

Copy

```
 <header>
     <nav class="navbar navbar-toggleable-md navbar-light bg-faded">
         <button class="navbar-toggler navbar-toggler-right" type="button" data-
toggle="collapse" data-target="#navbarSupportedContent"
         aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle
navigation">
         <span class="navbar-toggler-icon"></span>
     </button>
         <a class="navbar-brand" [routerLink]="['/']"><img class="navbar-logo"
src="./assets/todo_logo.png" alt="Logo"></a>
         <div class="collapse navbar-collapse" id="navbarSupportedContent">
         <ul class="nav navbar-nav mr-auto">
             <li class="nav-item active">
             <a class="nav-link" [routerLink]="['/']">All Items</a>
             </li>
```

```
          </ul>
          </div>
      </nav>
  </header>
```

2. Right-click on the logo below and save it to the src/assets folder of our project


## 10.4 Display Header

**Exercise: Update Main page**

Now that we have created the menu, we need to add it to our rendered view.

1. Open the src\app\app.component.ts file

2. Import the header component. Add the import statement on line 2 right below the existing import statement.

```
import { HeaderComponent } from './shared/header/header.component';
```

Copy

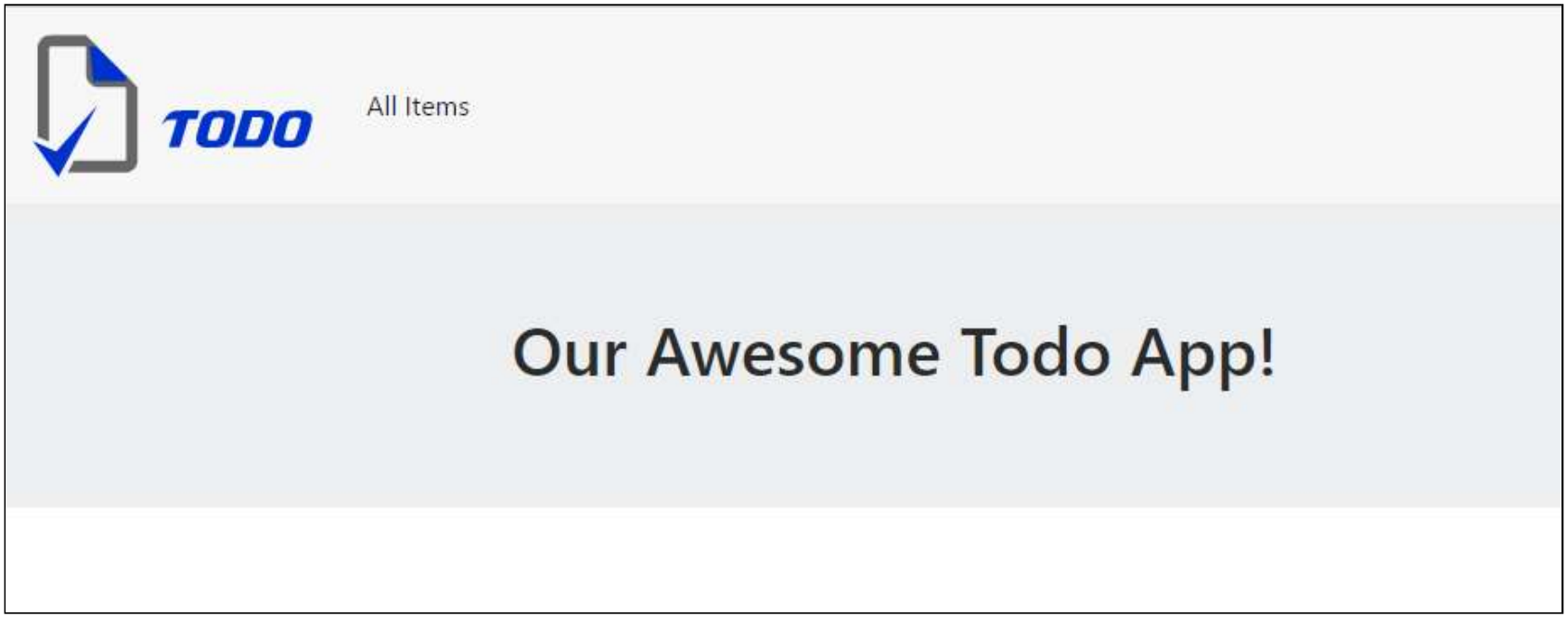
3. Open the src\app\app.component.html file and replace the contents of this file with the following html.

Copy

```
<app-header></app-header>
<div class="jumbotron">
    <div class="container">
        <h1>{{title}}</h1>
    </div>
</div>
<div class="container clearfix">
    <router-outlet></router-outlet>
</div>
```

**Note:** that the router-outlet tag is how Angular knows where to put the content of each of the views.

4. If you view the web page you should see the header with logo, "All Items" menu and our banner.




## 10.5 Create Footer

Creating the footer is very similar to creating the header. The biggest difference is that we have some css styling that we will apply to position the footer at the bottom of the page.

The really awesome part of the css we are going to apply is that Angular has the concept of CSS encapsulation so the styling will only apply to the footer component and not to the rest of the site. This gets automatically implemented for us, just by putting the styling in the footer components scss file instead of using th global style.scss file.

**Exercise: Creating the footer component**

1. In the Integrated Terminal, run the ng generate command to create the header component

Copy

```
ng generate component shared/footer
```

2. The generate command will create 4 files:

```
PS C:\projects\angular-tutorial>  ng generate component shared\footer
installing component
  create src\app\shared\footer\footer.component.scss
  create src\app\shared\footer\footer.component.html
  create src\app\shared\footer\footer.component.spec.ts
```

   - scss (styles)
   - html (view)
   - spec file (test)
   - component (controller)
   - add reference to the component in the app.module.ts file.

3. Open the src\app\shared\footer\footer.component.html and replace the contents with

Copy

```
<footer>
    <div class="pull-center">
        &copy;Angular WS
```

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

## Exercise: Styling the Footer

We want to postion the footer at bottom of the page and change the background color. With Angular 2, we now have the ablity to have css constrained to an individual component like the footer instead of being global for the whole site to use. We still have the ability to have global css for those styles that should be applied to the whole site by using the src\styles.scss file.

Since the footer is in the src\app\shared\footer\footer.component.html file we need to add the footer styles to the src\app\footer\footer.component.scss file and add the following content

Copy

```
@import "../../../assets/bootstrap/variables.scss";

footer {
    position: fixed;
    height: 50px;
    bottom: 0;
    left: 0;
    right: 0;
    padding: 10px 5px;
    border-top: 1px solid $gray-light;
    background-color: $gray-lighter;
    font-size: 0.8em;
```

```
    color: $dark-blue;
    div {
        margin-left: 25px;
        margin-right: 25px;
    }
}
```

## 10.6 Display Footer

### Exercise: Update Main page

Now we are ready to add our footer to the Main page

1. Open the src\app\app.component.ts file and import the footer component below the HeaderComponent import statement (note: order is not important)

Copy

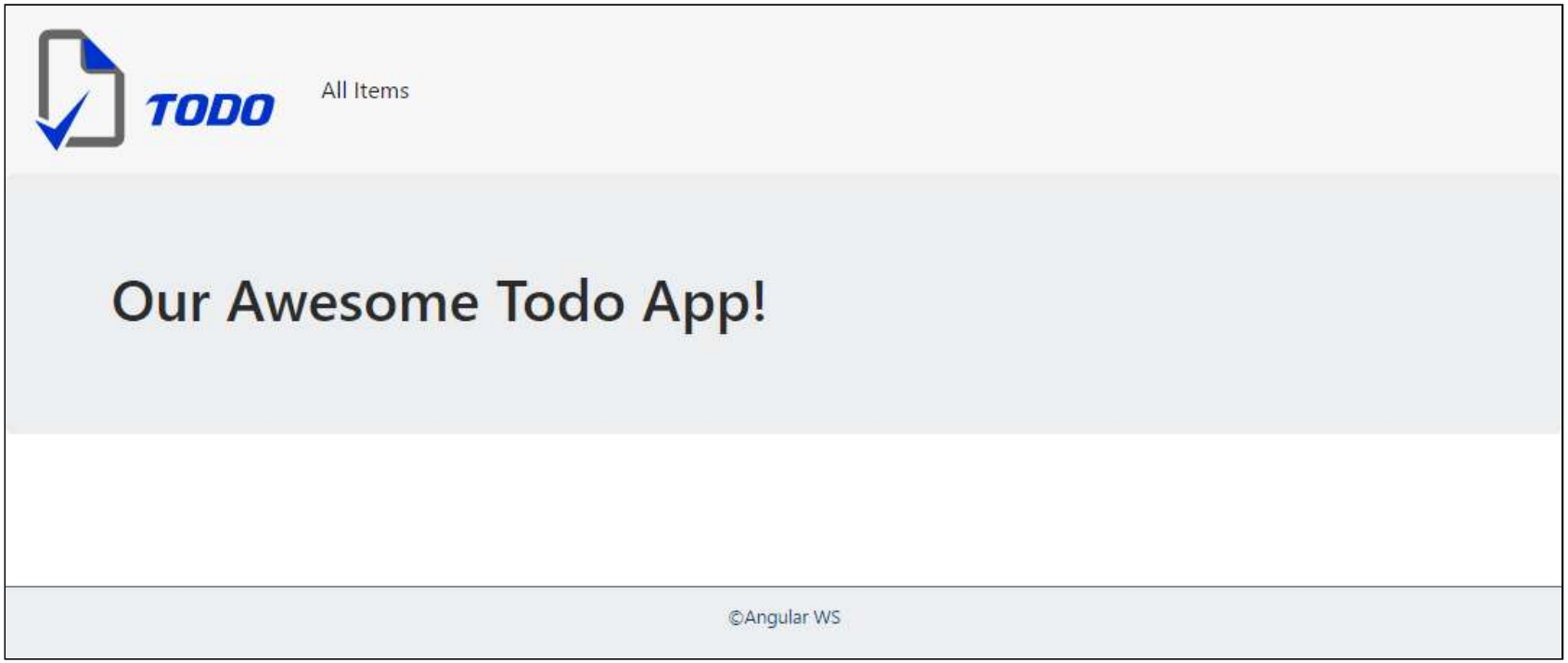
```
import { FooterComponent } from './shared/footer/footer.component';
```

2. Open the src\app\app.component.html file and replace the contents of this file with the following html

```
<app-header></app-header>
<div class="jumbotron">
    <div class="container">
        <h1>{{title}}</h1>
    </div>
</div>
<div class="container clearfix">
    <router-outlet></router-outlet>
</div>
<app-footer></app-footer>
```

Copy

3. If you view the web page you should see the footer




## Exercise: Fixing body height

Since we have a static footer, will need to change the body height to account for the footer height. Without doing this the body content will be hidden for the last 50px of the screen.

So far we have seen how to use component level css but Angular still has the ability to do global styles that will apply to the whole by adding them into the src\style.scss file.

1. Open the src\style.scss file and add the following css

```
html {
    position: relative;
    height: auto;
    min-height: 100%;
}

body {
    position: static;
    height: auto;
    margin: 0 0 60px; /* bottom = footer height */
}
```

Copy

## 10.7 Review

In this chapter will learned:

1. How to create new components
2. How to import and use components within other components
3. Learned about component level styles so that they only apply to a single component
4. Learned how to do global styles that apply to the whole site

5. Learned that if we have a static footer that we need to adjust the body margin height to stop content for being hidden behind the footer.
6. Learned that the `<router-outlet></router-outlet>` tag is used to tell Angular the location within the html code to render the routed component's template (html)

# 11. Deploying

## 11.1 Overview

Before deploy you will need to run an a build with the Angular CLI. With the build you can tell it the environment to use and if it is a production build or not. A production will minify everything.

## 11.2 Goals

- Learn how to create a build that is ready to deploy.

## 11.3 Non-Production Build

> Note the build create a dist folder to hold the output. This directory is removed before each build

Select which type of build you want to run: Non-Production or Production

### Exercise: Running Non-Production Build

Run the following command to run a build that uses the environment.ts file and is not a production build

Copy

```
ng build
```

## 11.4 Production Build

### Exercise: Running a Production Build

Note that this will remove the dist directory before build.

Run the following command to run a build that uses the environment.prod.ts file and is a production build. The environment name must match the file name for it to be valid.

Copy

```
ng build --target=production --environment=prod
```

# 12. Bonus: Additional Todo Features

## 12.1 Overview

Right now the Todo UI is fully functional but there is some nice usability things that could be immplemented.

- The todo list is sorted by when it was created, it would be much nicer to sort alphabetically and by completion status.
- There is no way to see a count of how many items you have open. The form also
- The add form is not cleared out on successful save

## 12.2 Goals

- Show how to implement additional usability features such as running open item count, sorting in the UI and resetting the form.

## 12.3 Displaying Open Item Count

It would be nice to know how many todo items that the user has and display that in the UI. You will need to make sure to update the open number when pulling, updating, and deleting todo items.

### Exercise: Calculating Open Items

1. Open the src\app\todo\todo.component.ts file

2. Create a variable to hold the open count in the TodoCompoment class that is of type number

Copy

```
openItemCount: number = 0;
```

3. Add the following function to get the count of open todo items by using the filter function to look for items where the completed value is false.

Copy

```
calculateOpenItems(): void {
    this.openItemCount = this.todoList.filter(item => item.completed === false).length;
}
```

### Exercise: Updating Open Item Count on Fetch

1. In the getTodoListAll function, add a call to the calculateOpenItems function after setting the todoList variable to data.

Copy

```
getTodoListAll(): void {
    this.todoService.getAll()
    .subscribe(
    data => {
        this.todoList = data;
        this.calculateOpenItems();
    },
    error => {
        this.errorMessage = <any>error;
    });
}
```

## Exercise: Update Open Item Count on Add

1. In the todo.component.ts file increment the openItemCount in the save function

```
save(): void {
    this.todoService.save(this.addForm.value.item)
        .subscribe(result => {
            console.log('save result', result);
            this.todoList.push(result);
            this.openItemCount++;
        },
        error => {
            this.errorMessage = <any>error;
        });
}
```

Copy

Since we are just adding a new record and are not pulling a new todo list array from the service, we can just increment the openItemCount variable.

### Exercise: Update Open Item on Complete

When we toggle the completion status of a Todo item we also need to update the openItemCount value.

In the completeTodo, upon returning the data from the TodoService.updateTodo call, we need to update the openItemCount. To do this, we will call the calculateOpenItems function.

The reason that we did not just subtract from the openItemCount is that we are able to toggle completion status so sometime you will need to add and other times you will need to subtract. Instead of having to create this logic, we can just call the calculateOpenItems function to do the work for us.

1. In the todo.component.ts file, find the completeTodo function and add the call to calculateOpenItems in the returned data

Copy

```
completeTodo(todo: Todo): void {
    todo.completed = !todo.completed;
    this.todoService.updateTodo(todo)
        .subscribe(
        data => {
         this.calculateOpenItems();
        },
        error => {
            todo.completed = !todo.completed;
            this.errorMessage = <any>error;
            console.log('complete error', this.errorMessage);
        });
}
```

## Exercise: Update Open Item on Delete

The last thing we need to do is to re-calculate the openItemCount when an item is deleted. We don't just decrement since a user can delete a completed it and there is no sense in adding in additional logic since the calculateOpenItems already has the logic.

1. In the todo.component.ts file, find the deleteTodo function and add the call to the calculateOpenItems function after removing the deleted item from the todoList.

Copy

```
deleteTodo(todo: Todo): void {
    this.todoService.deleteTodo(todo)
        .subscribe(
        data => {
            let index = this.todoList.indexOf(todo);
            this.todoList.splice(index, 1);
            this.calculateOpenItems();
        },
        error => {
            todo.completed = !todo.completed;
            this.errorMessage = <any>error;
            console.log('complete error', this.errorMessage);
        });
}
```

### Exercise: Add Open Item Count to UI

The last thing we need to do is add the open item count to the UI. It should automatically update as items are added, updated, and deleted.

1. Open the todo.component.html file

2. Inside of the page-header div after the "Todo List" title, add the following html

Copy

```
<p class="lead">You've got <em>{{openItemCount}}</em> things to do</p>
```

3. The UI should now look like

# 12.4 Sort in the UI

Right now the data from the API comes back unsorted. It is ordered based on the order that they were added. Instead it would be much better if we sorted the todo items based on the text and completion status. The completed items should be at the bottom. As well the sort should be case insensitive.

With ES6 and TypeScript we can use a map and reduce function to accomplish this task.

### Exercise: Sorting in the Client

1. Open the todo.component.ts file

2. Add the following code to create a generic array sort function

Copy

```
fieldSorter(fields, ignoreCase) {
    return (a, b) => fields.map(o => {
        let dir = 1;
        if (o[0] === '-') { dir = -1; o = o.substring(1); }
        if (ignoreCase === true && typeof a[o] === 'string' && typeof b[o] === 'string') {
        a[0] = a[o].toLocaleLowerCase();
        b[o] = b[o].toLocaleLowerCase();
        }
```

```
            return a[o] > b[o] ? dir : a[o] < b[o] ? -(dir) : 0;
        }).reduce((p, n) => p ? p : n, 0);
    }
```

- We can even have it sort if needed in decending order by appending on a - sign to the field name.

> The fieldSorter function is generic and can be reused on any array sort call.

1. Now that we have our generic sorting method, we can create a function to do the actual sorting that can be called in the save, completedTodo and deleteTodo functions

Copy

```
sortItems(): void {
    this.todoList.sort(this.fieldSorter(['completed', 'name'], true));
}
```

2. The last thing we need to do is to call the sortItems function in the save, completeTodo and getTodoListAll functions.  Make sure to place the call after any updates to the todoList have occurred

Copy

```
this.sortItems();
```

- We do not need to call the sort in the deleteTodo since the list will already be sorted and we are only removing a single item.

3. Now when you use interact with the UI the list will stay sorted by name and complete state. The sort is also case insensitive and the open items should be on the top of the list.


## 12.5 Reset Form After Save

Right now after you add a new todo item, the form does not clear out the input box. Then when you go to clear it out, the required field validator will be fired.

Instead, we can clear out the form on a successful save call by calling `this.addForm.reset();`

```
this.addForm.reset();
```

# 13. Bonus: Additional Login Features

## 13.1 Goals

- Understand how to create a cookie with user information in it
- Understand how to logout the user

## 13.2 Caching User

Right now when you refresh the page the current user information in the AuthService is lost. We can cache the data using cookies. To implement the cookie storage we are going to use the ngx-cookie library.

**Exercise: Install ngx-cookie**

1. Open terminal and add/install the ngx-cookie library

Copy

```
npm install --save ngx-cookie
```

2. Open src\app\app.module.ts

3. Import the ngx-cookie library

Copy

```
import { CookieModule } from 'ngx-cookie';
```

4. Add the ngx-cookie library to the @Ngmodule imports sections

Copy

```
imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    HttpModule,
    AppRoutingModule,
    CookieModule.forRoot()
],
```

### Exercise: Add Cookie Get/Set Functions

1. Open the auth.service.ts file

2. Add the following functions to get/set the cookie

Copy

```
getUser(): User {
    return <User>this.cookieService.getObject(this.cookieKey)
}

private setUser(value: User): void {
    this.cookieService.putObject(this.cookieKey, value);
}

private clearUser() : void {
    this.cookieService.remove(this.cookieKey);
}
```

## Exercise: Setting Cookie

1. In the login and signup functions, inside the do statement remove the existing code that is setting the this.currentUser variable and replace it with the following.

```
if (res) {
      this.setUser(<User>res.json());
}
```

2. In the login and isAuthenticated functions, in the catch section, add a call to clearUser

```
this.clearUser();
```

3. In the isAuthenticated function, in the map section, if res is false call clearUser

```
this.clearUser();
```

4. You can remote the class variable currentUser as it will no longer be used.

## 13.3 Logout User

Before being able to signup for an account, we need the user to be logged out first.

There are a number of ways that you could implement this such as giving a logout button in the header or showing user info in header with link to profile page with a logout button.

We are going to implement the logout button in the header.

### Exercise: Create AuthService Logout

1. open the auth.service.ts file

2. Add the logout function below that will call the API logout function and clear out the cookie

Copy

```
logout(): Observable<boolean> {
    return this.http.get(`${this.url}/logout`, this.options)
    .map((res: Response) => {
      if (res.ok) {
        this.clearUser();
        return Observable.of(true);
      }

      return Observable.of(false);
    })
    .catch(error => {
        console.log('logout error', error)
        this.clearUser();
        return Observable.of(false);
```

```
      });
  }
```

### Exercise: Add Logout Button

1. Open the src\app\shared\header\header.component.html

2. Inside of the `<div class="collapse navbar-collapse"....` tag add the following after the closing `</ul>`

Copy

```
 <ul class="nav navbar-nav">
     <li class="nav-item">
         <span class="nav-link">Welcome {{(authService.getUser())?.email}}  <a
[hidden]="!authService.getUser()" (click)="logout()"> | Logout</a></span>
     </li>
 </ul>
```

   - This code will display a Welcome along with the email if it is populated.

### Exercise: Add Compoment Logging Out Function

1. Open header.component.ts

2. Import the AuthService and Router

Copy

```
import { Router } from '@angular/router';
import { AuthService } from '../services/auth.service';
`
```

3. Add the AuthService and Router to the constructor

Copy

```
constructor(private authService: AuthService, private router: Router) { }
```

4. Add the logout function

Copy

```
logout() {
    this.authService.logout().subscribe(() => {
        this.router.navigate(['/login']);
    });
}
```

5. You are now ready to test it.

# 14. Thank you

I hope that you enjoyed going through this tutorial. If you have any questions, please hit me up on twitter @digitaldrummerj.

# 15. Tags For Creating Tutorial

## 15.1 Exercises

**Exercise: Title**

Details Here

```
<h4 class="exercise-start">
  <b>Exercise</b>:
</h4>

<div class="exercise-end"></div>
```

## 15.2 Alerts

Bootstrap based alerts

## Danger

Danger

```
<div class="alert alert-danger" role="alert">

</div>
```

## Warning

Warning

```
<div class="alert alert-warning" role="alert">

</div>
```

## Info

Info

```
<div class="alert alert-info" role="alert">
```

```
</div>
```